



Serenity

Developer Guide

---

# Table of Contents

Introduction	1.1
Getting Started	1.2
Installing Serene From Visual Studio Marketplace	1.2.1
Installing Serene Directly From Visual Studio	1.2.2
Instaling Serene Asp.Net Core Version with Serin	1.2.3
Starting Serene	1.2.4
A Tour Of Serene Features	1.3
Theming	1.3.1
Localization	1.3.2
User and Role Management	1.3.3
Listing Pages	1.3.4
Edit Dialogs	1.3.5
Tutorials	1.4
Movie Database	1.4.1
Creating Movie Table	1.4.1.1
Generating Code For Movie Table	1.4.1.2
Customizing Movie Interface	1.4.1.3
Handling Movie Navigation	1.4.1.4
Customizing Quick Search	1.4.1.5
Adding a Movie Kind Field	1.4.1.6
Adding Movie Genres	1.4.1.7
Updating Serenity Packages	1.4.1.8
Allowing Multiple Genre Selection	1.4.1.9
Filtering with Multiple Genre List	1.4.1.10
The Cast and Characters They Played	1.4.1.11
Listing Movies in Person Dialog	1.4.1.12
Adding Primary and Gallery Images	1.4.1.13
Multi Tenancy	1.4.2
Adding Tenants Table and TenantId Field	1.4.2.1
Generating Code for Tenants Table	1.4.2.2

---

Tenant Selection in User Dialog	1.4.2.3
Filtering Users By TenantId	1.4.2.4
Removing Tenant Dropdown From User Form	1.4.2.5
Securing Tenant Selection At Server Side	1.4.2.6
Setting TenantId For New Users	1.4.2.7
Preventing Edits To Users From Other Tenants	1.4.2.8
Hiding the Tenant Administration Permission	1.4.2.9
Making Roles Multi-Tenant	1.4.2.10
Using Serenity Service Behaviors	1.4.2.11
Extending Multi-Tenant Behavior To Northwind	1.4.2.12
Handling Lookup Scripts	1.4.2.13
Meeting Management	1.4.3
Creating Lookup Tables	1.4.3.1
How To Guides	1.5
How To: Remove Northwind & Other Samples From Serene	1.5.1
How To: Update Serenity NuGet Packages	1.5.2
How To: Upgrade to Serenity 2.0 and Enable TypeScript	1.5.3
How To: Authenticate With Active Directory or LDAP	1.5.4
How To: Use a SlickGrid Formatter	1.5.5
How To: Add a Row Selection Column	1.5.6
How To: Setup Cascaded Editors	1.5.7
How To: Use Recaptcha	1.5.8
How To: Register Permissions in Serene	1.5.9
How To: Use a Third Party Plugin With Serenity	1.5.10
How To: Enable Script Bundling	1.5.11
How To: Debugging with Serenity Sources	1.5.12
Frequently Asked Questions	1.6
Troubleshooting	1.7
Service Locator & Initialization	1.8
Dependency Static Class	1.8.1
IDependencyResolver Interface	1.8.2
IDependencyRegistrar Interface	1.8.3
MunqContainer Class	1.8.4
CommonInitialization Static Class	1.8.5

---

---

Authentication & Authorization	1.9
IAuthenticationService Interface	1.9.1
IAuthorizationService Interface	1.9.2
IPermissionService Interface	1.9.3
IUserDefinition Interface	1.9.4
IUserRetrieveService Interface	1.9.5
Authorization Static Class	1.9.6
Configuration System	1.10
Defining Configuration Settings	1.10.1
IConfigurationRepository Interface	1.10.2
AppSettingsJsonConfigRepository	1.10.3
Config Static Class	1.10.4
Localization	1.11
LocalText Class	1.11.1
Language Identifiers	1.11.2
Language Fallbacks	1.11.3
ILocalTextRegistry Interface	1.11.4
LocalTextRegistry Class	1.11.5
Pending Approval Mode	1.11.5.1
Registering Translations	1.11.6
Manually Registering Translations	1.11.6.1
Nested Local Texts	1.11.6.2
Enumeration Texts	1.11.6.3
JSON Local Texts	1.11.6.4
Caching	1.12
Local Caching	1.12.1
ILocalCache Interface	1.12.1.1
LocalCache Static Class	1.12.1.2
User Profile Caching Sample	1.12.1.3
Distributed Caching	1.12.2
WEB Farms and Caching	1.12.2.1
IDistributedCache Interface	1.12.2.2
Distributed Cache Static Class	1.12.2.3

---

---

DistributedCacheEmulator Class	1.12.2.4
CouchbaseDistributedCache Class	1.12.2.5
RedisDistributedCache Class	1.12.2.6
Two Level Caching	1.12.3
Using Local Cache and Distributed Cache In Sync	1.12.3.1
TwoLevelCache Class	1.12.3.2
Entities (Row)	1.13
Mapping Attributes	1.13.1
FieldFlags Enumeration	1.13.2
Fluent SQL	1.14
SqlQuery Object	1.14.1
Criteria Objects	1.14.2
Connections and Transactions	1.15
SQL Database Types	1.16
Working with Other Databases	1.16.1
Setting Connection Dialect	1.16.2
Dialect Based Expressions	1.16.3
PostgreSQL	1.16.4
MySQL	1.16.5
Sqlite	1.16.6
Oracle	1.16.7
Services	1.17
Service Endpoints	1.17.1
List Request Handler	1.17.2
Widgets	1.18
ScriptContext Class	1.18.1
Widget Class	1.18.2
Widget With Options	1.18.3
TemplatedWidget Class	1.18.4
TemplatedDialog Class	1.18.5
Attributes	1.19
Grids	1.20
Formatter Types	1.20.1
Persisting Settings	1.20.2

---

---

Code Generator (Sergen)	1.21
Used Tools & Libraries	1.22

---

# Introduction

## What is Serenity Platform

Serenity is an ASP.NET Core / MVC / TypeScript application platform which has been built on open source technologies.

It aims to make development easier while reducing maintenance costs by avoiding boiler-plate code, reducing the time spent on repetitive tasks and applying the best software design practices.

## Who/What This Platform Is For

Serenity is best suited to business applications with many data entry forms or administrative interface of public facing web sites. It's features can be useful for other kinds of web applications as well.

## Where To Look For Information

After reading this guide and its tutorials, follow resources below for more information about Serenity.

**Home Page:**

<http://serenity.is>

**Blog:**

<http://serenity.is/blog>

**Github Repository:**

<https://github.com/volkanceylan/Serenity>

**Issues / Questions**

<https://github.com/volkanceylan/Serenity/issues>

**Wiki (FAQ, Troubleshooting and Other Community Content)**

<https://github.com/volkanceylan/Serenity/wiki>

**Change Log:**

<https://github.com/volkanceylan/Serenity/blob/master/CHANGELOG.md>

### **Serene Application Template:**

[https://marketplace.visualstudio.com/items?  
itemName=VolkanCeylan.SereneSerenityApplicationTemplate](https://marketplace.visualstudio.com/items?itemName=VolkanCeylan.SereneSerenityApplicationTemplate)

## **What's In The Name**

Serenity has dictionary meanings of *peace*, *comfort* and *calmness*.

This is what we are trying to achieve with Serenity. We hope that after installing and using it you will feel this way too...

## **What Features It Provides**

- A modular, service based web application model
- Code generator to produce initial services / user interface code for an SQL table
- T4 based code generation on server to reference script widgets with intellisense / compile time validation
- T4 based code generation to provide compile time type safety and intellisense while calling AJAX services from script side.
- An attribute based form definition system (prepare UI in server side with a simple C# class)
- Automatic seamless data-binding through form definitions (form <-> entity <-> service).
- Caching Helpers (Local / Distributed)
- Automatic cache validation
- Configuration System (storage medium independent. store settings in database, file, whatever...)
- Simple Logging
- Reporting (reports just provide data, has no dependency on rendering, similar to MVC)
- Script bundling, minification (making use of Node / UglifyJS / CleanCSS) and content versioning (no more F5 / clear browser cache)
- Fluent SQL Builder (SELECT/INSERT/UPDATE/DELETE)
- Micro ORM (also Dapper is integrated)
- Customizable handlers for REST like services that work by reusing information in entity classes and do automatic validation.
- Attribute based navigation menu
- UI Localization (store localized texts in json files, embedded resource, database, in memory class, anywhere)
- Data Localization (using an extension table mechanism helps to localize even data entered by users, like lookup tables)



- Script widget system (inspired by jQueryUI but more suitable for C# code)
- Client side and server side validation (based on jQuery validate plugin, but abstracts dependency)
- Audit logging (where CDC is not available)
- System for data based integration tests
- Dynamic scripts
- Script side templates

## Background

This part was originally written for a CodeProject article as an introduction to Serenity. The article was rejected with the reason that it didn't contain code but was an ad for code. They were right, as i did put a link to Movie tutorial in this guide, instead of copy pasting code.

You can safely skip to next chapter, if you don't like reading history :)

We, developers, are all solving the same sets of problems everyday. Just like college students working on their problem books.

Even though we know that they are already solved and have answers somewhere, it doesn't stop us from working on them. Actually, it helps us improve our skills, and hey you can't learn without making some mistakes, can you? But we should learn where to draw a line between training and wasting time.

When you start a new project, you have several decisions to make on platform, architecture and set of libraries. Today you have so many choices for every single topic. Yes, having some options is good, as long as they are limited, as our time is not infinite.

Here is a short history about *Serenity*, which aims to handle common tasks you deal with business applications, and let you spare your precious time focusing on features specific to your application domain.

My first real job in web technologies was in a web agency designing country-specific web sites of some of big names in industry, e.g. automative companies (btw, we are talking about 10+ years past, time flows fast).

As I had a software architect career in desktop applications before I signed there, I was asked to design a ASP.NET WebForms platform for them. They explained that they have many shared modules, like news, galleries, navigation at each site, but as requirements are different, they had to copy/paste then customize code specific to every customer. When they wanted to add a common feature, they had to repeat it for every site.

At that time, there weren't so many CMS systems in market, and I designed one for them, without even knowing it was called a CMS. For me, it wasn't perfect, not even good enough as I just had a few weeks to design it. But they were very pleased with the result, as it took development of new sites down to days/weeks from months. Also resulting code was more manageable than before.

Learning from that experience, and mistakes, that poor-mans CMS became something better. Later, that platform is evolved to be used by applications in varying domains, like a help-desk system, a CRM, ERP, personnel management, electronic document management, university student information system and more.

To be compatible with different kinds of applications, systems and even legacy databases, it had to be flexible and went through many architectural changes.

Now it takes us to Serenity. Even though it is an open source project for about 2 years, it has a much older background. But it is also young, energetic, and is not afraid of change. It can adapt to new technologies as they became popular and stable. This might mean breaking changes from time to time, but we strive to keep them to a minimum without being paranoid about backwards compability.

# Getting Started

The best and fastest way to get your hands dirty on Serenity is *SERENE*, which is a sample application template.

You have three options to install *SERENE* template:

Please check prerequisites below before trying to install Serene.

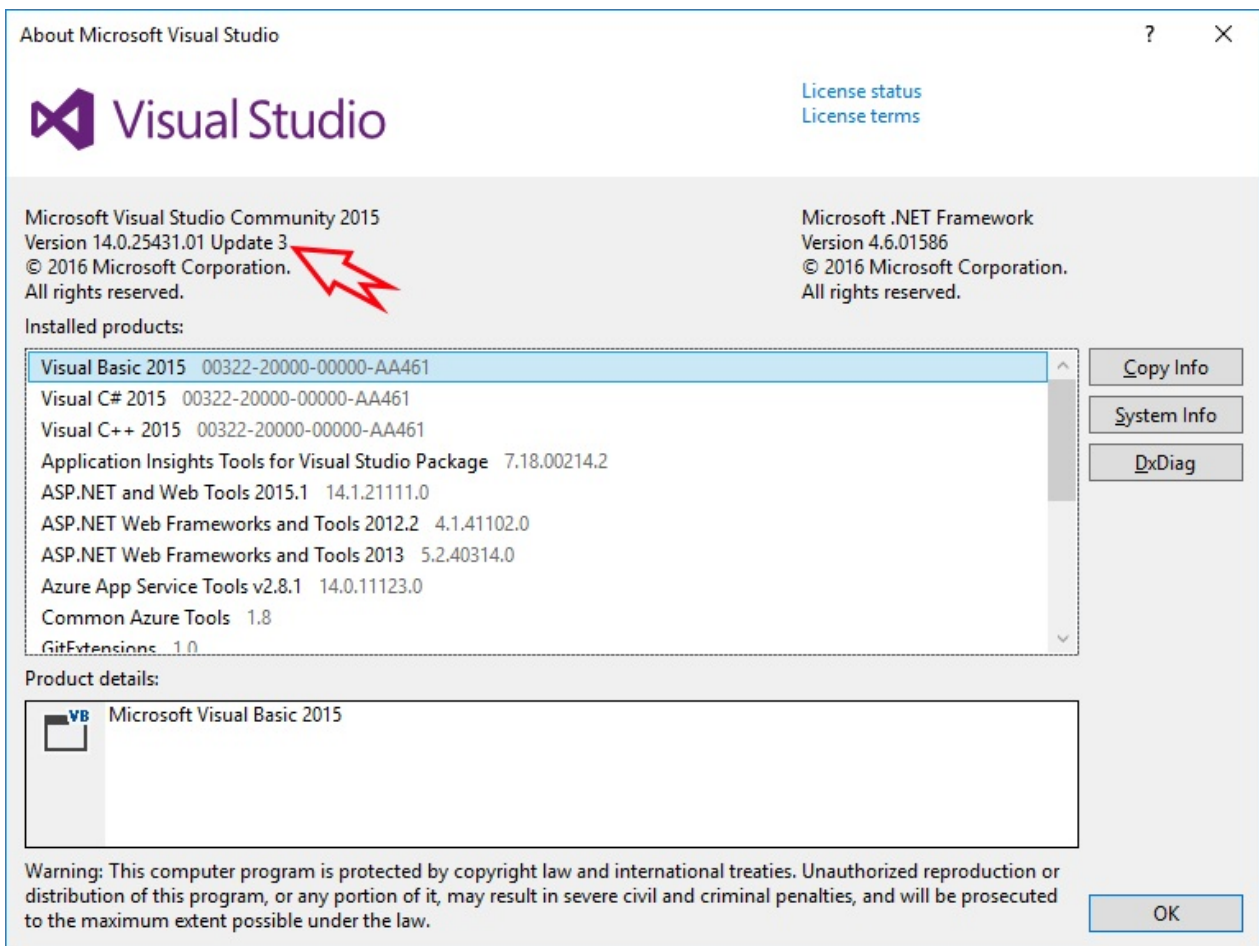
- [Installing SERENE from Visual Studio Marketplace \(Windows\)](#)
- [Installing SERENE directly from Visual Studio \(Windows\)](#)
- [Installing SERENE for Asp.Net Core with SERIN \(Linux, OSX, Windows\)](#)

## Prerequisites

### Visual Studio Version

Serene .NET Framework version (ASP.NET MVC 4) requires *Visual Studio 2017* or *Visual Studio 2015* with **Update 3** installed.

If you have Visual Studio 2015, please make sure that you have **Update 3** installed by looking at Help => About



It might be possible to work with Visual Studio 2013 as well but you'll have many intellisense errors as TypeScript 2.5.2 can't be installed in VS2013.

Serene ASP.NET Core 2.0 version only works in Visual Studio 2017, or using command line.

Microsoft recently obsoleted *project.json* based projects and replaced them with a lighter version of *MsBuild* based *CSPROJ* projects. This new project system only works in Visual Studio 2017, so if you want to work with .NET Core version of Serene, either you need to use Visual Studio 2017 or go lighter with Visual Studio Code / Command Line.

## .NET Core SDK

If you are going to use ASP.NET Core version of Serene, please install .NET Core 2.0 SDK from:

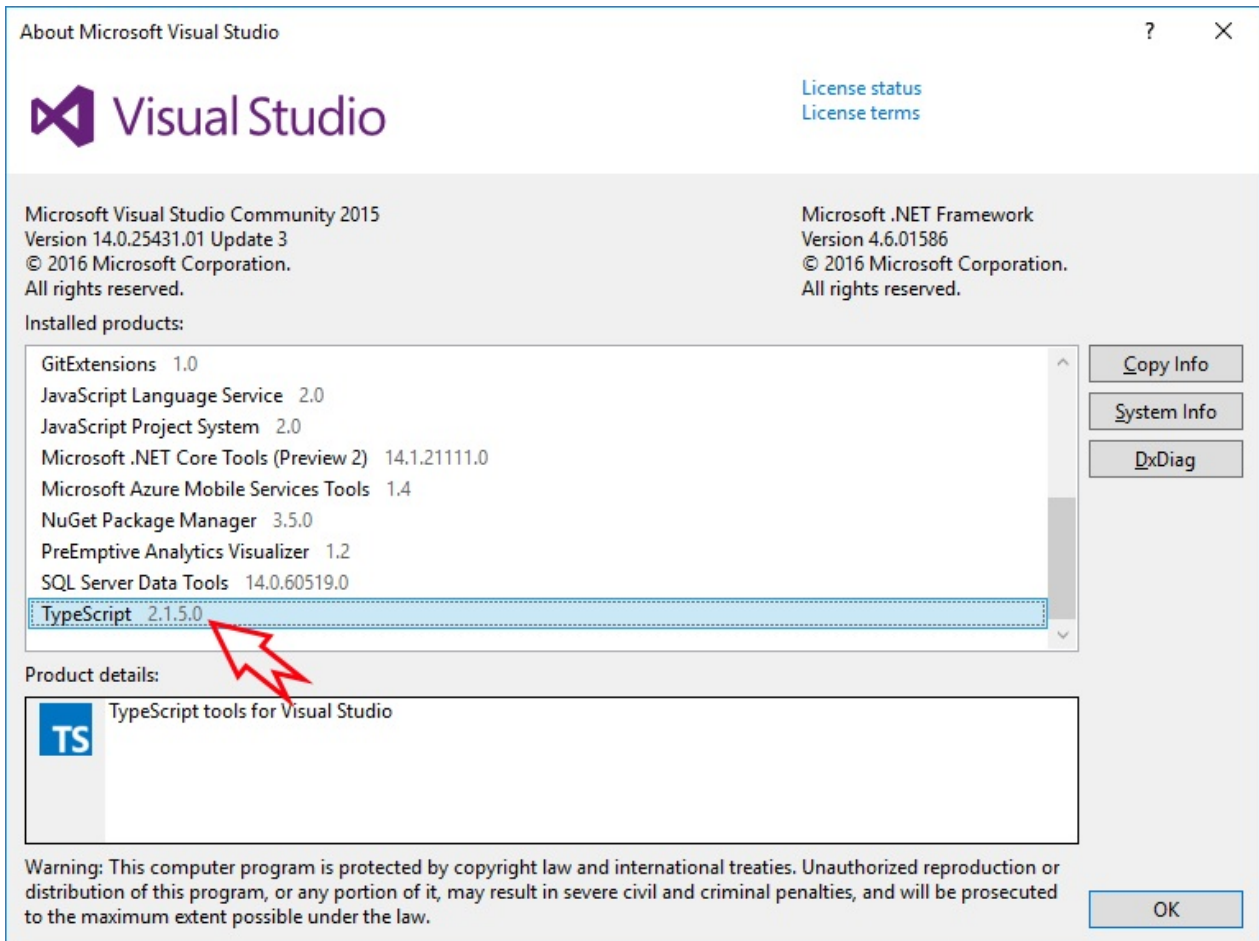
<https://www.microsoft.com/net/download/core>

## Visual Studio TypeScript Extension

As of writing, the recommended version of TypeScript is 2.5.2.

Even though Serene uses NodeJS based TypeScript compiler (tsc) on build, Visual Studio still uses its own version of TypeScript for intellisense and refactoring etc. If you have an older version of that extension, you'll be greeted with many errors as soon as you open a Serene project.

To check what version of TypeScript Visual Studio Extension you have, again see Help => About:



Visual Studio 2017 comes with TypeScript 2.1.5 by default, but Visual Studio 2015 might include older versions.

If you have something lower than 2.3.4 there, you might need to install TypeScript for Visual Studio extension.

TypeScript version you see in Control Panel / Add Remove Programs doesn't matter at all. What matters is the one that is enabled in Visual Studio.

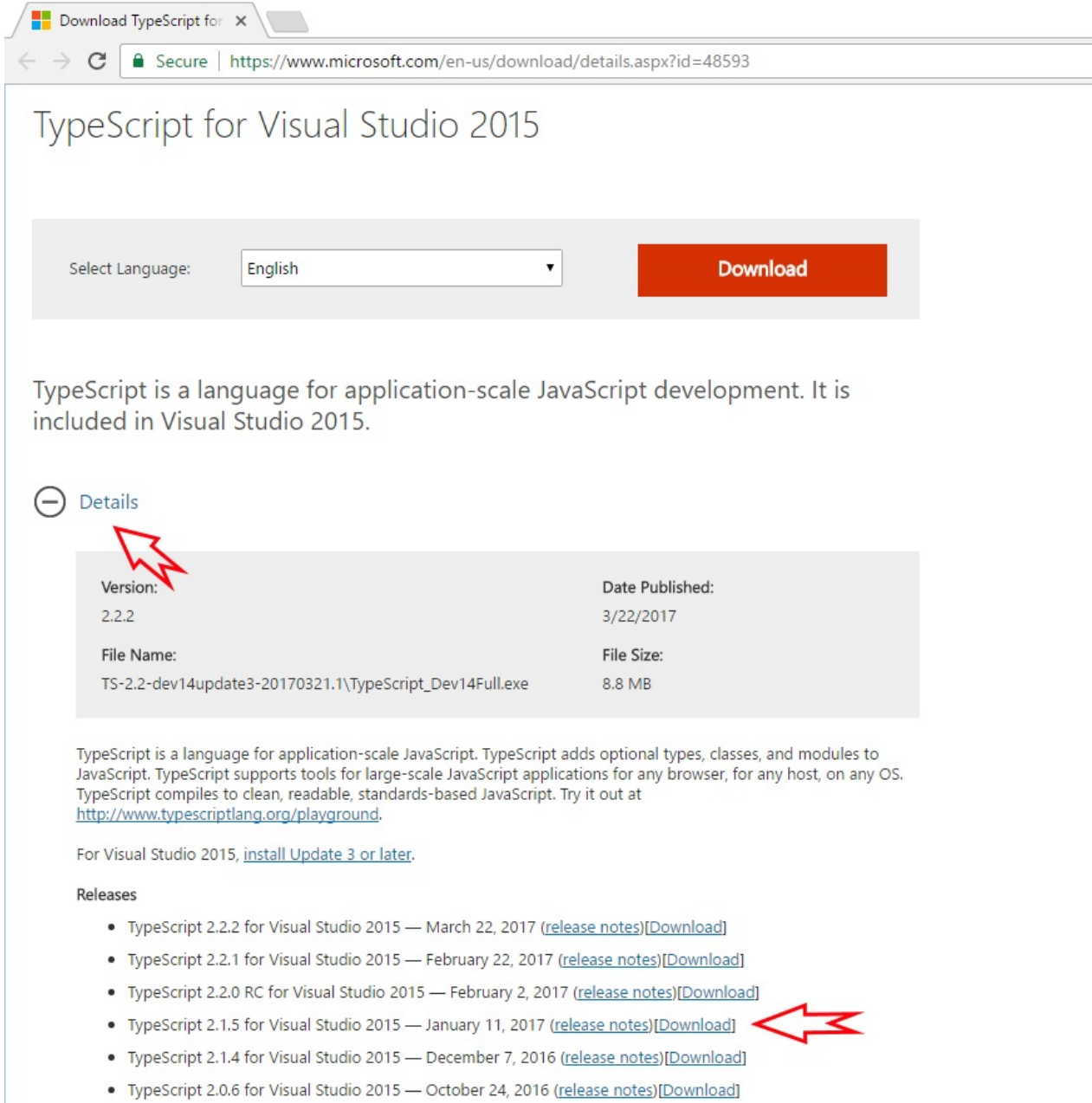
TypeScript versions after 1.8.6 requires Visual Studio 2015 Update 3 to be installed first, so even if you try to install the extension it will raise an error, so please first install Update 3.

You can get TypeScript extension for your Visual Studio version from <http://www.typescriptlang.org/#download-links>.

Here is the link for Visual Studio 2015:

<https://www.microsoft.com/en-us/download/details.aspx?id=48593>

But **don't click the download button** right away. Expand **Details** section, and select the exact version you need (e.g. 2.5.2):



Latest version of TypeScript might probably work but keeping in sync with the version we currently use, can help you avoid compatibility problems that could come with them.

## NodeJS / NPM

Serene uses NodeJS / NPM for these:

- TypeScript typings (.d.ts) for libraries like jQuery, Bootstrap etc.

- TypeScript compiler itself (tsc)
- Less compilation (lessjs)
- T4 Code generation by parsing TypeScript sources

It requires NodeJS v6.9+ and NPM 3.10+

Serene will check their versions on project creation and ask for confirmation to download and install them. Anyway, please check your versions manually by opening a command prompt:

```
> npm -v  
3.10.10
```

```
> node -v  
6.9.4
```

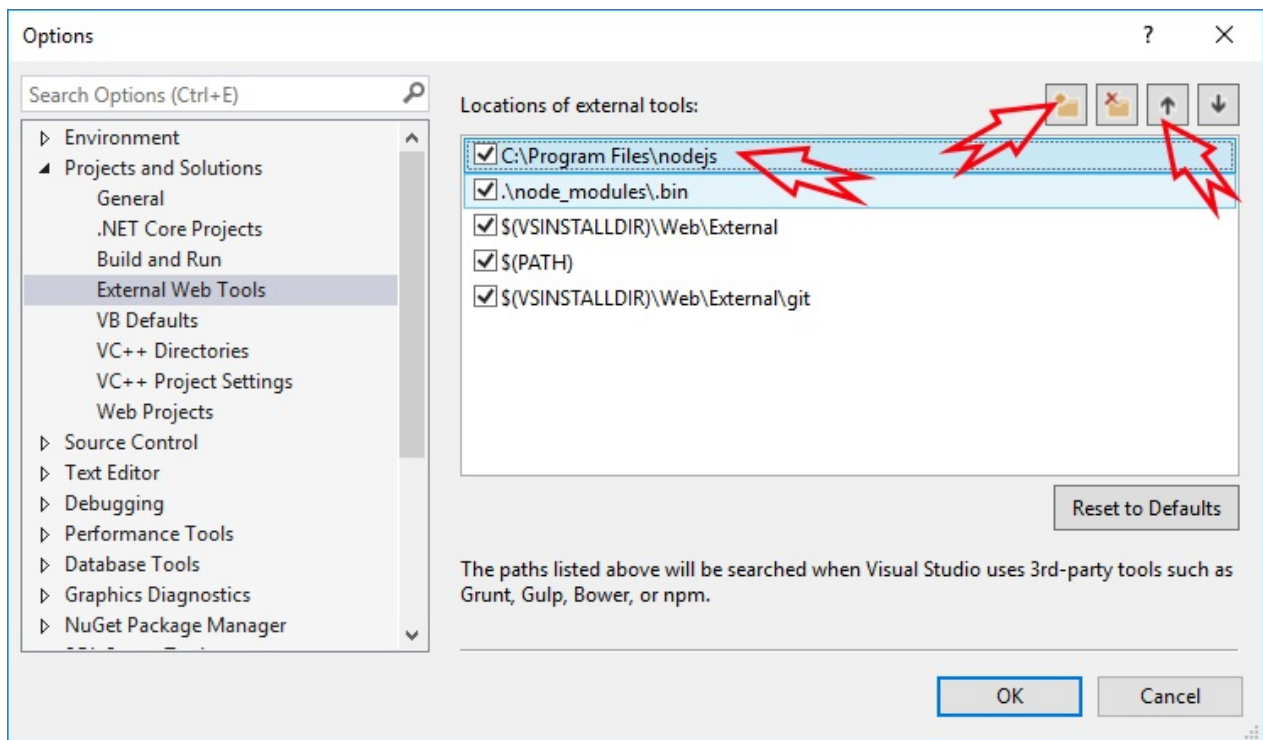
If you get an error, they might not be installed or not in path. Please install LTS (long term support) versions from <https://nodejs.org/en/>

Current version might also work but is not tested.

## Visual Studio and External Web Tool Paths

Even if you have correct Node / NPM installed, Visual Studio might still be trying to use its own integrated, and older version of NodeJS.

Click *Tools => Options*, and then under *Projects and Solutions => External Web Tools* add *C:\Program Files\nodejs* to the top of the list by clicking plus folder icon, typing *C:\Program Files\nodejs* and using *Up Arrow* to move it to the start:



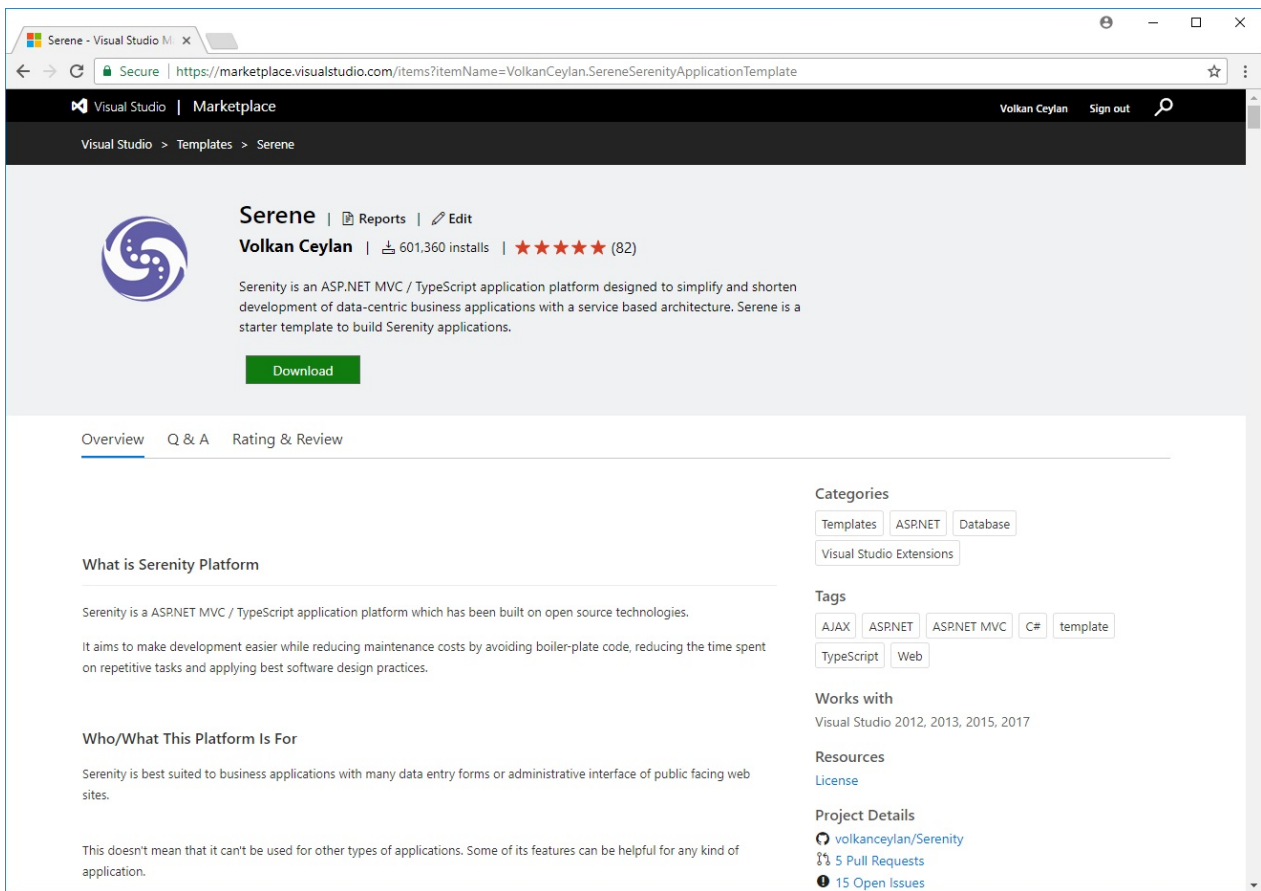


# Installing Serene From Visual Studio Marketplace

## Downloading Template

Open URL below in your browser:

<https://marketplace.visualstudio.com/items?itemName=VolkanCeylan.SereneSerenityApplicationTemplate>

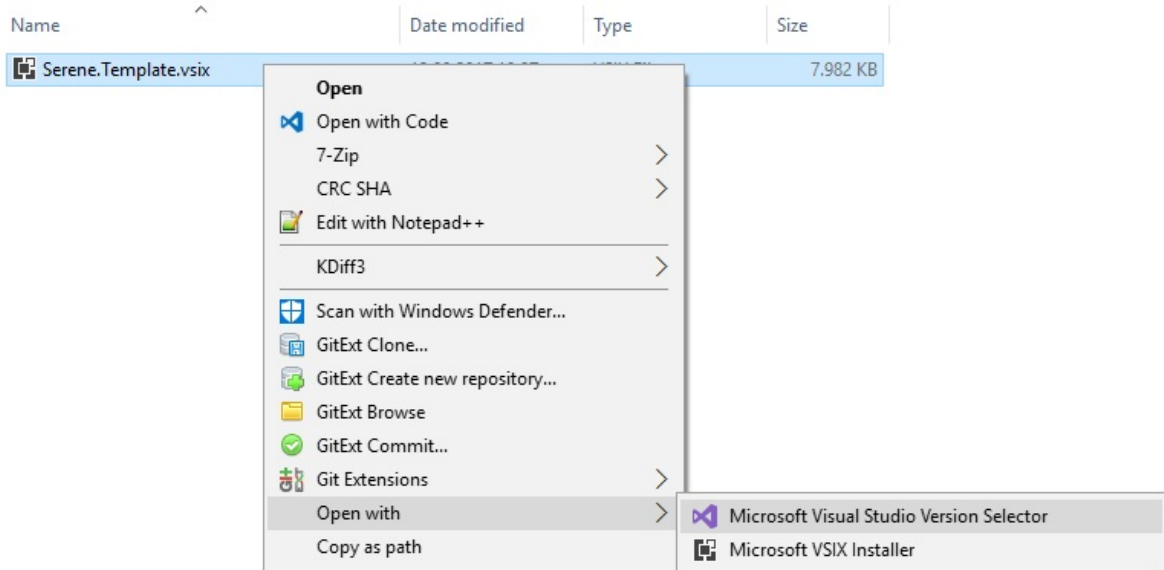


Click *Download* to transfer VSIX file to your computer.

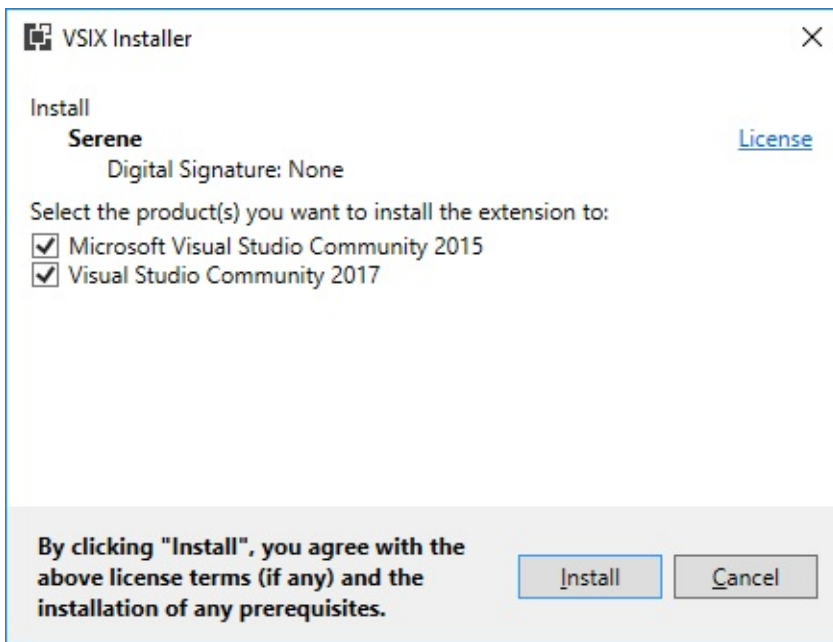
## Install Template into Visual Studio

After download is finished, double click the downloaded VSIX file to start Visual Studio extension installation dialog

If you have both Visual Studio 2017 and 2015 installed, sometimes Visual Studio 2015 installer might pick up VSIX file so it only installs in Visual Studio 2015. If you experience this issue, right click the file, click *Open With* and choose *Visual Studio Version Selector*.



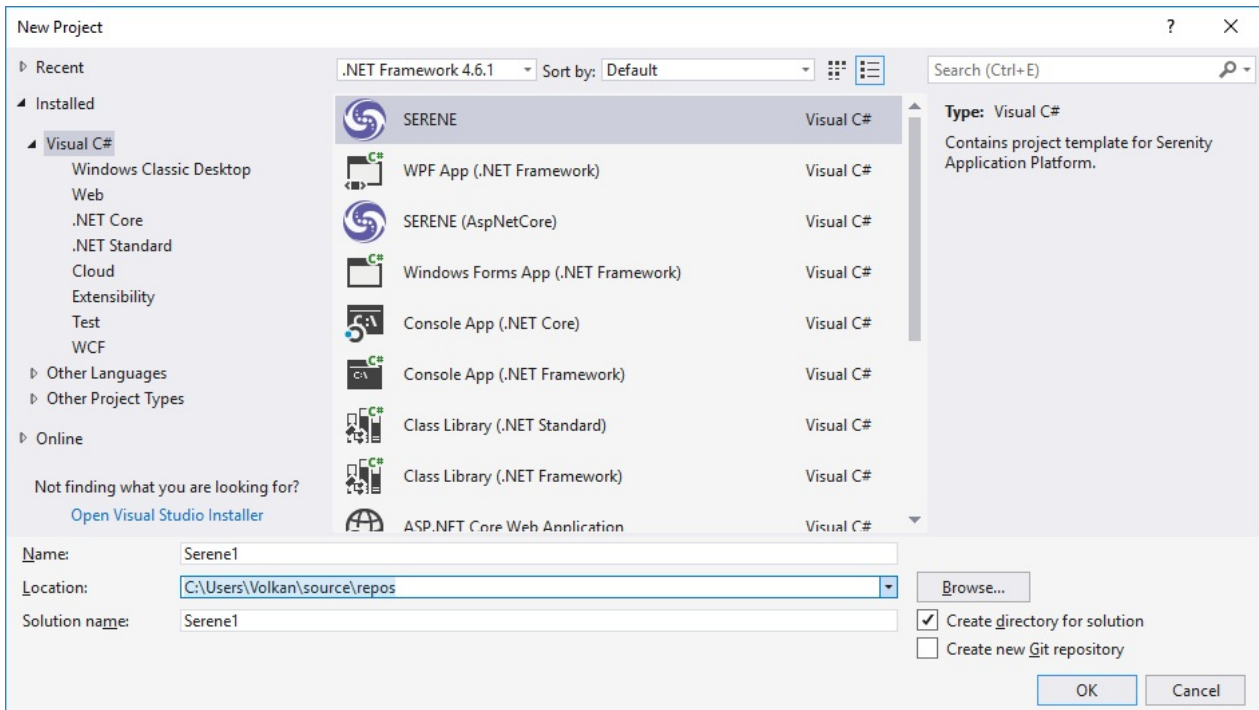
Click Install when prompted.



Note that this application template requires Visual Studio 2012 or higher. Make you sure you have the latest Visual Studio updates installed. ASP.NET MVC Core version requires Visual Studio 2017 with Update 3

## Creating a New Project in Visual Studio

Start Visual Studio (if it was already open, restart it). Click File => New Project. You should see Serene template under Templates => Visual C# section.



We have two versions of Serene template. One that uses classic ASP.NET MVC 4 (SERENE) and another one that works on ASP.NET CORE MVC 2.0 / .NET CORE 2.0.

ASP.NET Core is a recent technology and is platform independent (as long as you target .NET Core it also runs on Linux / OSX).

ASP.NET MVC only runs on Windows and .NET framework but more mature (latest version is dated 2/9/2015).

We can say both versions of Serene is pretty stable.

Here is a document from Microsoft that might help you choose between two frameworks:

<https://docs.microsoft.com/en-us/aspnet/core/choose-aspnet-framework>

Name your application something like *MyCompany*, *MyProduct*, *HelloWorld* or leave the default *Serene1*.

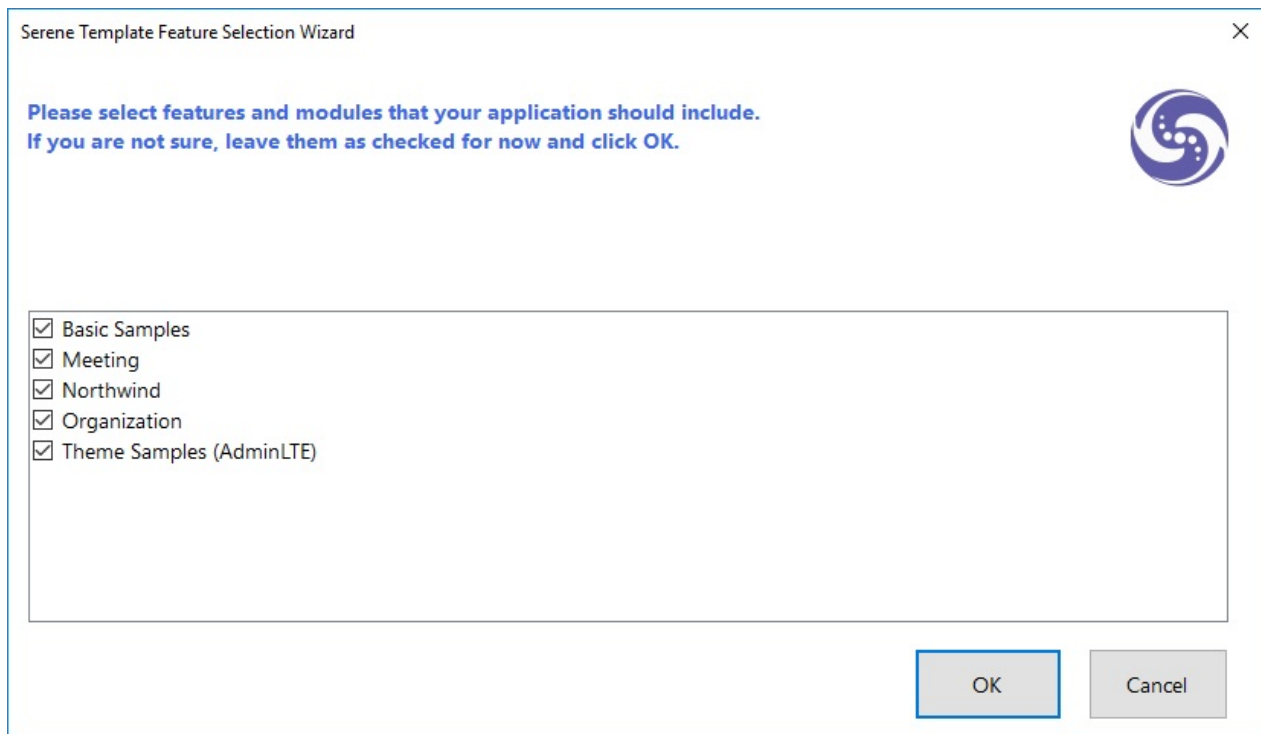
Please don't name it *Serenity*. It may conflict with other Serenity assemblies.

Please use Pascal casing, e.g. a name that starts with a Capital Letter. Don't name your project something like `myProject`.

Click OK.

## Feature Selection

Serene will prompt you to choose features you would like to see.



All of these features / samples are optional. Initially we recommend you to leave them all checked so that you might have a look at how they are implemented.

After having some experience with Serene, you might create a new application and clear all these checkboxes to have a bare minimum project.

Choose features you like, click OK and take a break while Visual Studio creates the solution.

# Installing Serene Directly From Visual Studio

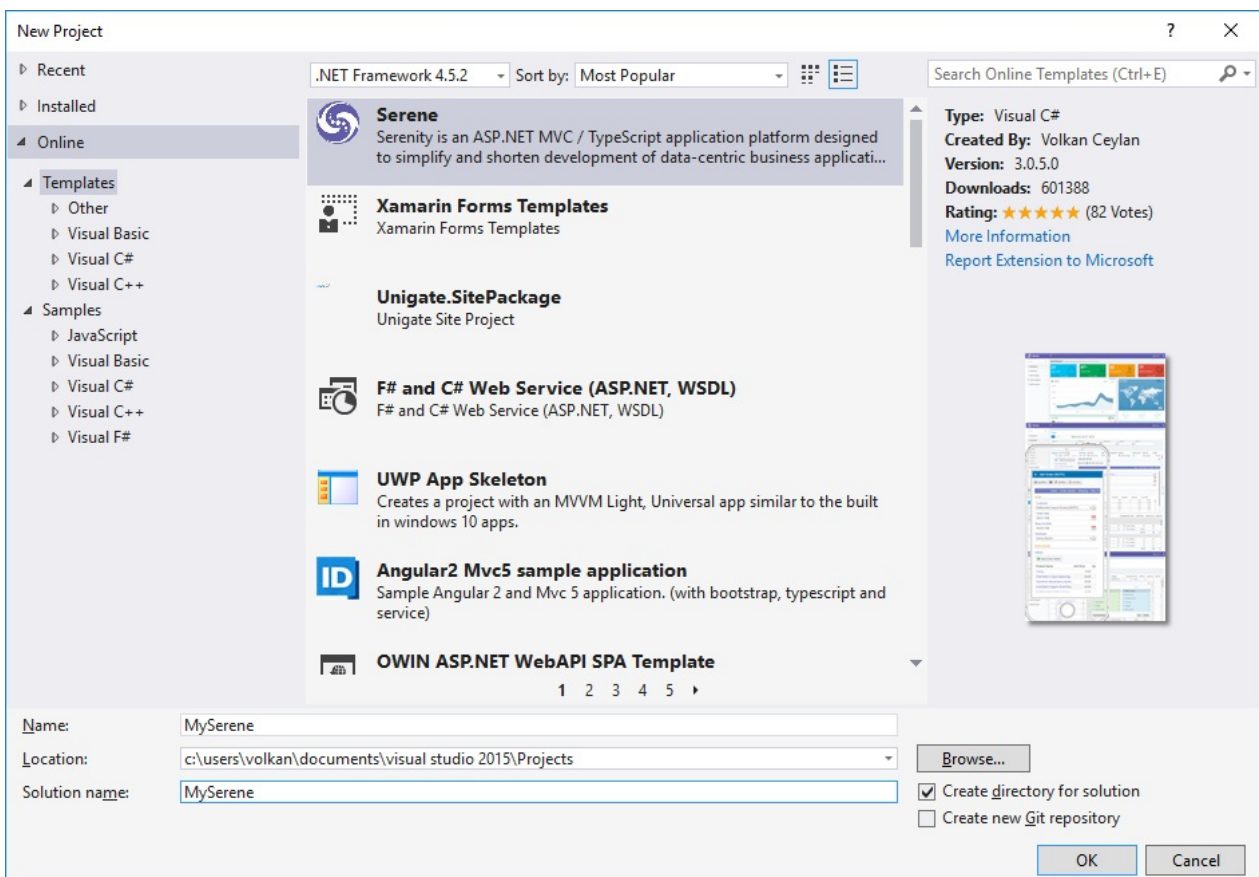
Start Visual Studio and Click *New => Project*.

Note that this application template requires Visual Studio 2012 or higher. Make sure you have the latest Visual Studio updates installed.

In the New Project dialog box *Recent*, *Installed* and *Online* sections will be shown on left and *Installed* is the active one.

Click the *Online* section and wait a bit while *Retrieving information* message is on screen.

Please wait while it is loading results.



Serene might be already showing on top of the list. If it is not, type *SERENE* into input box with *Search Online Templates* label and press ENTER.

You will see *Serene (Serenity Application Template)*:

## Creating a New Project in Visual Studio 2015 and Older

Name your application something like *MyCompany*, *MyProduct*, *HelloWorld* or leave the default *Serene1*.

Please don't name it *Serenity*. It may conflict with other Serenity assemblies.

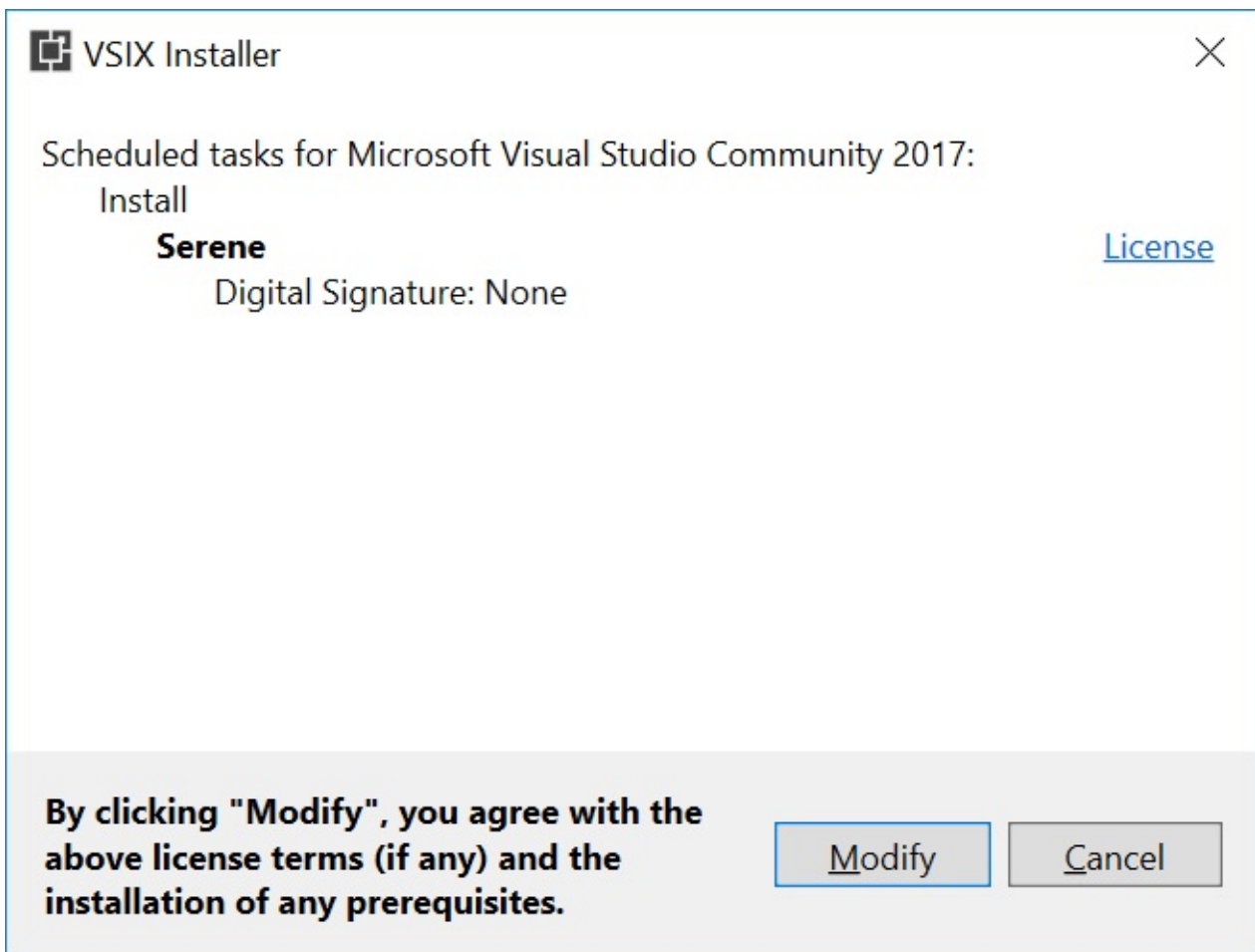
After you create your first project, Serene template is installed into Visual Studio, so you can use the *Installed* section in *New Project* dialog to create another Serenity application.

Click OK to download Serene and create your new project

Your project will use ASP.NET MVC 4 version of Serene, next time you may choose ASP.NET Core version from *New Project* dialog, *Installed* section.

## Creating a New Project in Visual Studio 2017

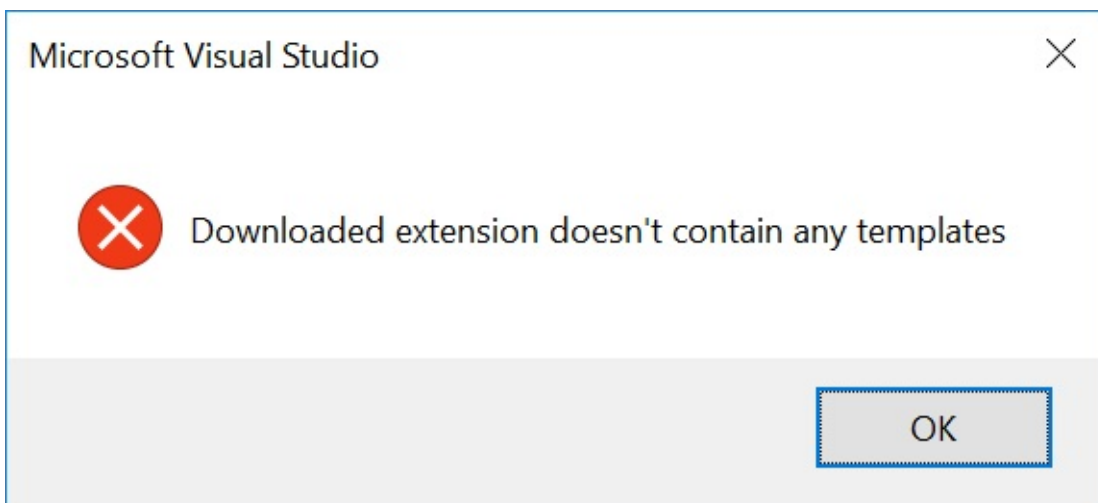
Unfortunately Visual Studio 2017 changed template installation process. When you select Serene from *Online* section and click OK, you'll see this dialog:



When you click Modify, you'll be asked to terminate Visual Studio and other related processes.

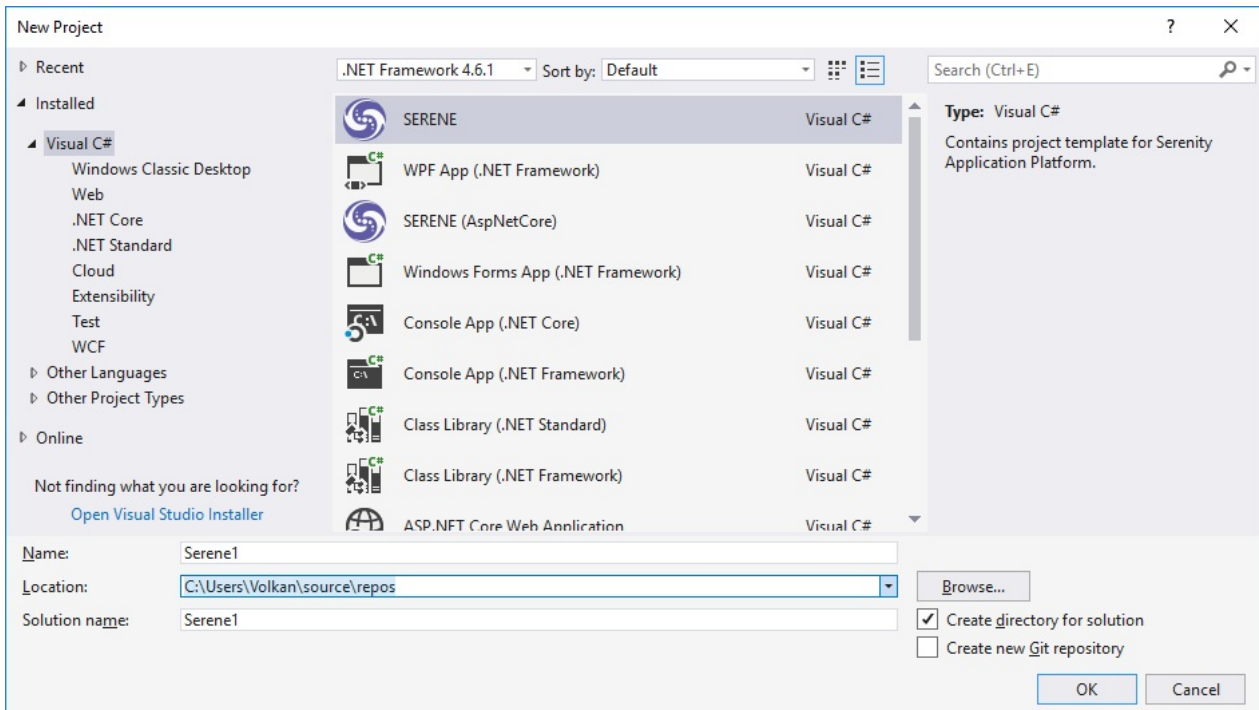
After installation, Visual Studio won't restart itself. You'll have to manually relaunch Visual Studio.

Now, if you again go to *Online* section and click *Serene*, you'll get this error message:



Unfortunately, this is a VS2017 bug.

Please close *Online* section, and find Serene under *New Project => Installed => Visual C#:*



After you create your first project, Serene template is installed into Visual Studio, so you can use the *Installed* section in *New Project* dialog to create another Serenity application.

Name your application something like *MyCompany*, *MyProduct*, *HelloWorld* or leave the default *Serene1*.

Please don't name it *Serenity*. It may conflict with other Serenity assemblies.

Please use Pascal casing, e.g. a name that starts with a Capital Letter. Don't name your project something like `myProject`.

Click OK.

## Choosing Between ASP.NET MVC / ASP.NET Core

We have two versions of Serene template. One that uses classic ASP.NET MVC 4 (SERENE) and another one that works on ASP.NET CORE 2.0 / .NET CORE 2.0.

ASP.NET Core is a recent technology and is platform independent (as long as you target .NET Core it also runs on Linux / OSX).

ASP.NET MVC only runs on Windows and .NET framework but more mature (latest version is dated 2/9/2015).

We can say both versions of Serene is pretty stable.



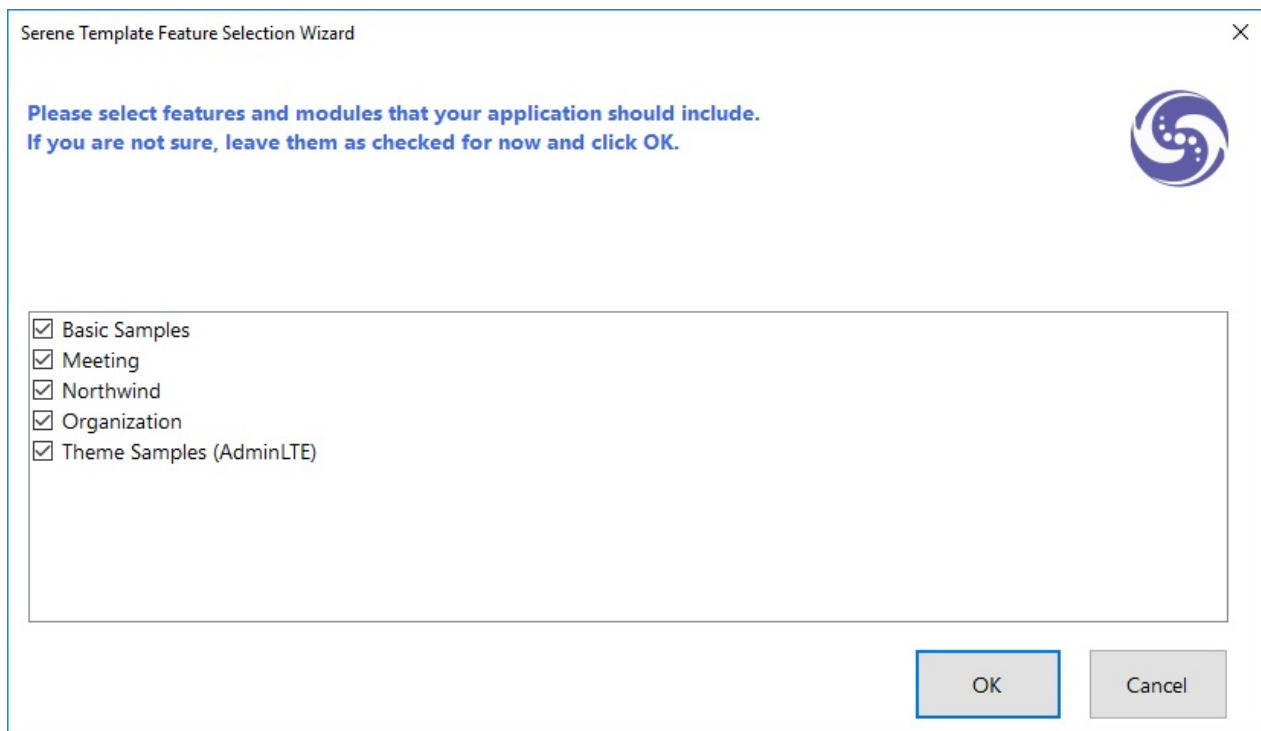
Here is a document from Microsoft that might help you choose between two frameworks:

<https://docs.microsoft.com/en-us/aspnet/core/choose-aspnet-framework>

Unfortunately Visual Studio 2015 doesn't let you choose which version to use on first install. But, you will see ASP.NET Core version after installation on New Project dialog.

## Feature Selection

Serene will prompt you to choose features you would like to see.



All of these features / samples are optional. Initially we recommend you to leave them all checked so that you might have a look at how they are implemented.

After having some experience with Serene, you might create a new application and clear all these checkboxes to have a bare minimum project.

Choose features you like, click OK and take a break while Visual Studio creates the solution.

# Installing Serene Asp.Net Core Version with SERIN

This section is for users who doesn't or can't use Visual Studio (in Linux / OSX).

Serene Asp.Net Core version supports Linux and OSX in addition to Windows.

We recommend Visual Studio Code for all platforms, but it is also possible to work with a basic text editor like Notepad / VIM. There are also other nice options e.g. Atom.

## Install .NET Core 2.0 SDK

Please go to address below and follow instructions for your specific platform:

<https://www.microsoft.com/net/download/core>

## Install NodeJS

As TypeScript (and our SERene project INitializer - SERIN) runs on NodeJS you need to install Node/NPM from:

<https://nodejs.org/en/download/>

or using your favorite package manager:

<https://nodejs.org/en/download/package-manager/>

## Install SERIN as a Global Tool

Install our project initializer, *SERIN* as a global tool using NPM:

**Linux / OSX:**

```
> sudo npm install -g serin
```

**Windows:**

```
> npm install -g serin
```

```
victor@vctor ~ $ sudo npm install -g serin
/usr/local/bin/serin -> /usr/local/lib/node_modules/serin/index.js
/usr/local/lib
├── serin@1.0.3
│   ├── xml2js@0.4.17
│   ├── sax@1.2.1
│   ├── xmlbuilder@4.2.1
│   └── lodash@4.17.4
victor@vctor ~ $
```

Thanks to Victor (@vctor) for Linux screenshots

## Create Folder for New Project

Create an empty *MySerene* (or a name you like) folder.

### Linux / OSX:

```
> cd ~
> mkdir MySerene
> cd MySerene
```

### Windows:

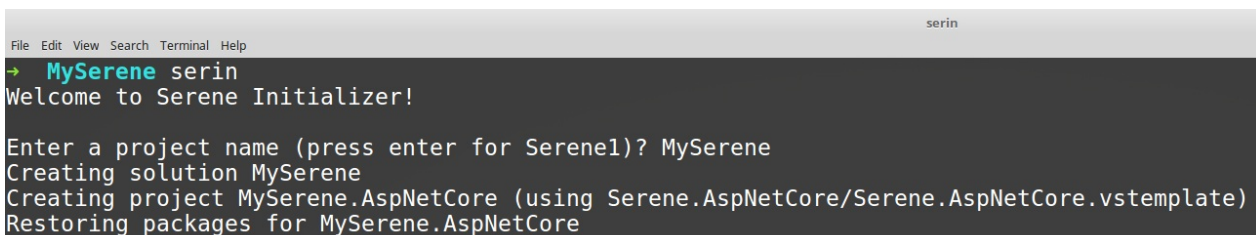
```
> cd c:\Projects
> mkdir MySerene
> cd MySerene
```

Serin has to be run from a completely empty directory

## Run Serin to Create a New Project

While inside an empty directory, run *serin*:

```
> serin
```



```
File Edit View Search Terminal Help
serin
→ MySerene serin
Welcome to Serene Initializer!

Enter a project name (press enter for Serene1)? MySerene
Creating solution MySerene
Creating project MySerene.AspNetCore (using Serene.AspNetCore/Serene.AspNetCore.vstemplate)
Restoring packages for MySerene.AspNetCore
```

Type an application name, e.g. *MySerene* and press enter. Take a break while Serin creates your project, initializes static content and restores packages etc.

After Serin creates your project, you will have a *MySerene.Web* folder under current directory. Enter that directory:

```
> cd MySerene.Web
```

## Running Serene

For OSX / Linux, first restore packages:

```
> dotnet restore
```

Make sure you run this command under *MySerene.AspNetCore* folder.

Then type:

```
> dotnet run
```

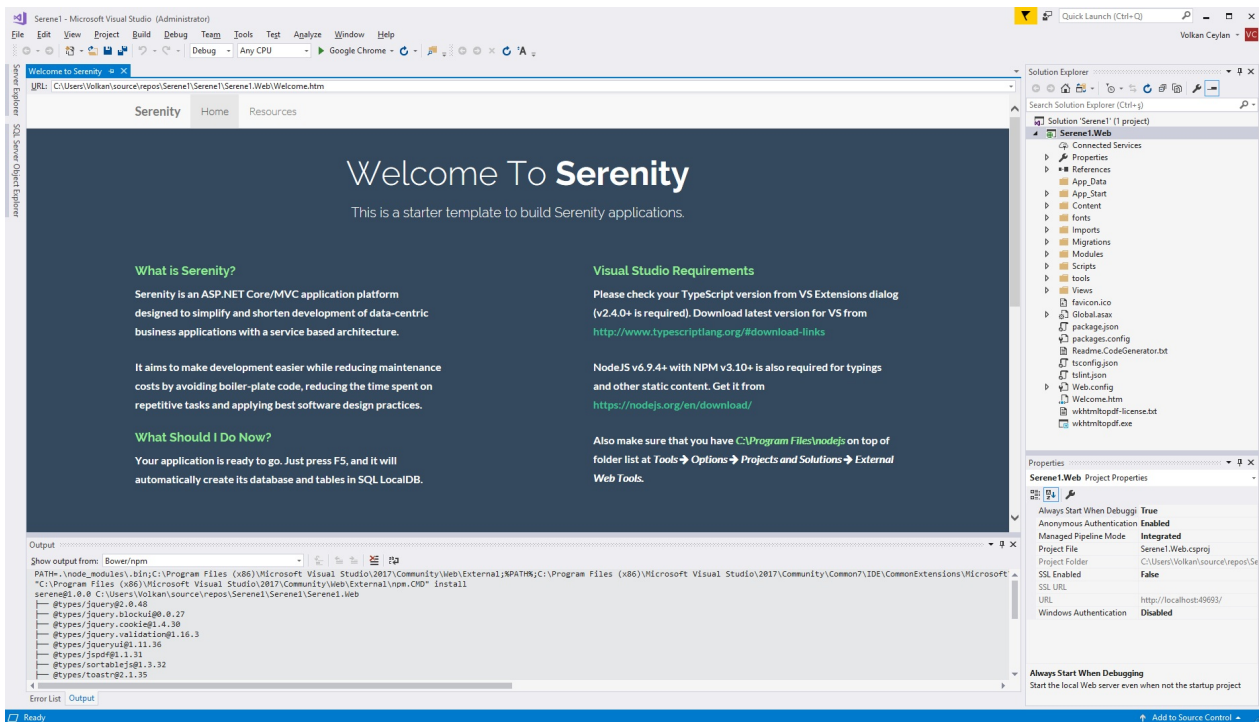
Now open a browser and navigate to `http://localhost:5000` .

Actual port may vary. You'll see it on console after executing *dotnet run*.

# Starting Serene

After your first project is created in Visual Studio using Serene template, you will see a solution like this:

Asp.Net Core users don't have to use Visual Studio, but we'll use Visual Studio in this guide as we think most of our users will.



Your solution contains Serene1.Web project, which is an ASP.NET MVC (or ASP.NET Core) application.

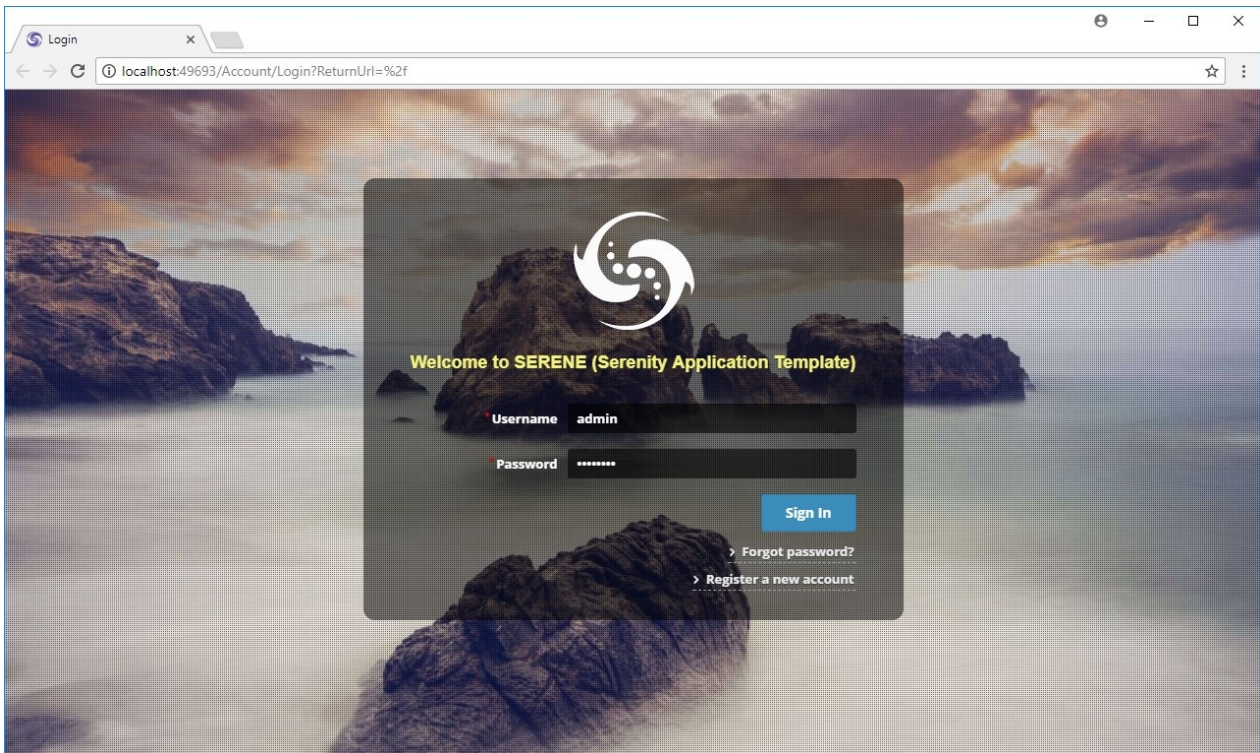
It includes server side code written in C# (.cs) and client side code that is written in TypeScript (.ts).

Serene.Web has references to Serenity NuGet packages, so you can update it using package manager console anytime necessary.

Asp.Net Core version can also be updated by hand editing .CSPROJ file.

Serene automatically creates its database in SQL local db at first run, so just press F5 and you are ready to go.

When application launches use `admin` user and `serenity` password to login. You can change password or create more users later, using *Administration / User Management* page.



The sample application includes old and famous Northwind data along with services and user interface to edit it, which is mostly produced by Serenity Code Generator.

## Troubleshooting Connection Problems

If you are getting a connection error like the following while starting Serene for first time:

- > A network-related or instance-specific error occurred
- > while establishing a connection to SQL Server.
- > The server was not found or was not accessible.
- > Verify that the instance name is correct...

This error might mean that you don't have SQL Server Local DB 2012 installed. This server comes preinstalled with Visual Studio 2012 and 2013.

## ASP.NET MVC

In `web.config` file there are *Default* and *Northwind* connection entries:

```
<connectionStrings>
  <add name="Default" connectionString="Data Source=(LocalDb)\v11.0;
    Initial Catalog=Serene_Default_v1; Integrated Security=True"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

## ASP.NET Core

In `appsettings.json` file you'll find *Default* and *Northwind* connection entries:

```
"Data": {
  "Default": {
    "ConnectionString": "Server=(localdb)\\MsSqlLocalDB;Database=Serene2_Default_v1;
Integrated Security=true",
    "ProviderName": "System.Data.SqlClient"
  },
  "Northwind": {
    "ConnectionString": "Server=(localdb)\\MsSqlLocalDB;Database=Serene2_Northwind_v
1;Integrated Security=true",
    "ProviderName": "System.Data.SqlClient"
  }
}
```

## Fixing Connection Strings

`(localdb)\v11.0` corresponds to default SQL Server 2012 LocalDB instance, while `(localdb)\MsSqlLocalDB` is an instance of SQL 2014+ LocalDB.

If you don't have SQL LocalDB 2012, you can install it from:

<http://www.microsoft.com/en-us/download/details.aspx?id=29062>

Visual Studio 2015 comes with SQL Server 2014 LocalDB. Its default instance name is renamed to `MsSqlLocalDB` by default. Thus, if you have VS2015, try changing connection strings from `(localdb)\v11.0` to `(localdb)\MsSqlLocalDB`.

```
<connectionStrings>
  <add name="Default" connectionString="Data Source=(LocalDb)\MsSqlLocalDB;
  Initial Catalog=Serene_Default_v1; Integrated Security=True"
  providerName="System.Data.SqlClient" />
</connectionStrings>
```

If you still have an error, open an administrative command prompt and type

```
> sqllocaldb info
```

This will list localdb instances like:

```
MSSqlLocalDB
test
```

If you don't have MsSqlLocalDB listed, you can create it:

```
> sqllocaldb create MsSqlLocalDB
```

If you have another SQL server instance, for example SQL Express, change data source to

```
.\SqlExpress :
```

```
<connectionStrings>
  <add name="Default" connectionString="Data Source=.\SqlExpress;
    Initial Catalog=Serene_Default_v1; Integrated Security=True"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

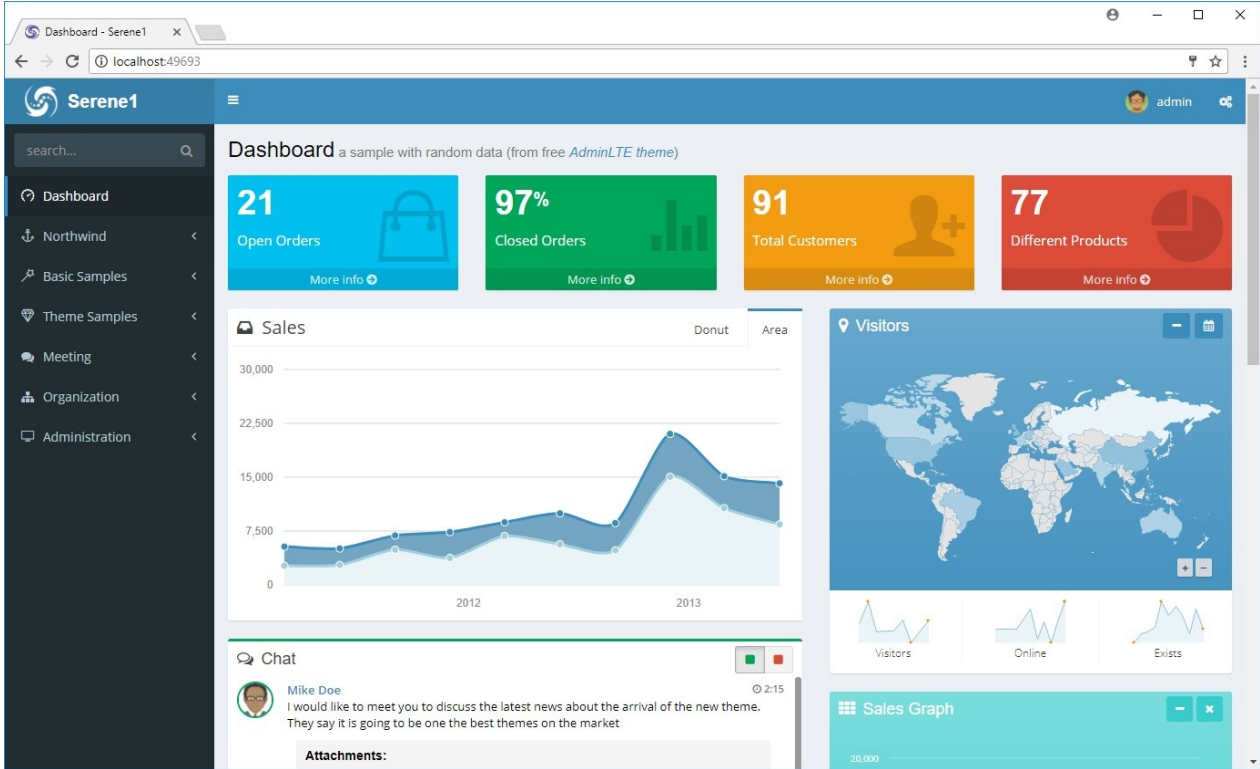
You can also use another SQL server. Just change the connection string.

Perform these steps for both Default and Northwind databases.



# A Tour Of Serene Features

After logging to Serene, you are greeted with the dashboard page.



This page is taken as a sample from free AdminLTE (<https://almsaeedstudio.com/themes/AdminLTE/index.html>) theme.

The page content, except some numbers calculated from Northwind tables, contains random data.

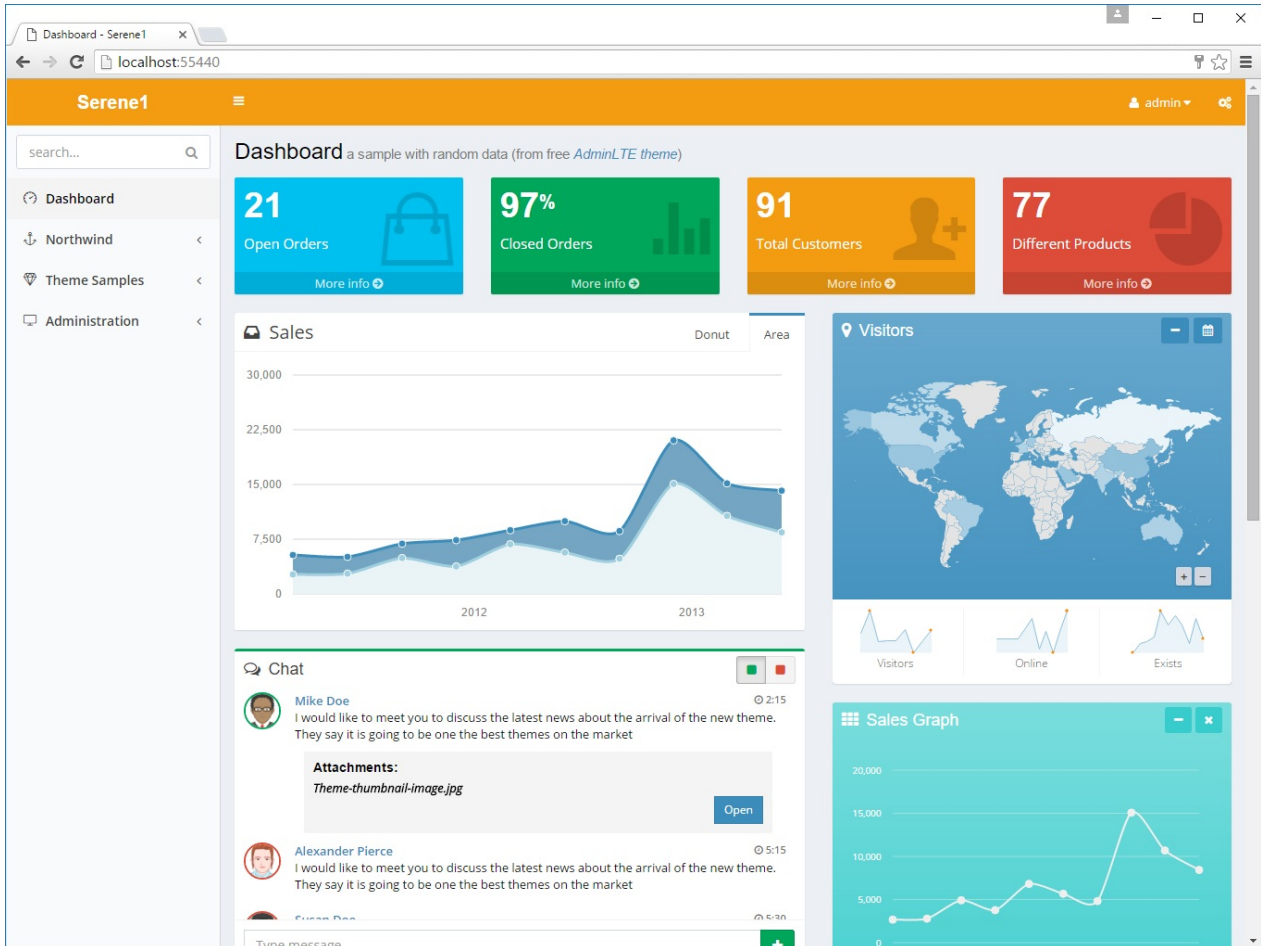
There is an accordion navigation menu on left which has a search feature by input above it. We'll talk about how to customize items there in later sections.

On top navigation, there is site name on left, along with a dropdown containing current user name on right, and a settings button through which we change change theme or active language.

- [Theming](#)
- [Localization](#)

# Theming

Serene initially starts with a dark/blue theme. On top right of the screen, next to username, click the settings button and change theme to another one.



This feature is implemented by replacing a body CSS class.

If you look at the source, you may spot a skin class like below inside `<body>` tag:

```
<body id="s-DashboardPage" class="fixed sidebar-mini hold-transition
  skin-blue has-layout-event">
```

When you select the light yellow skin, it actually changes to this:

```
<body id="s-DashboardPage" class="fixed sidebar-mini hold-transition
  skin-yellow-light has-layout-event">
```

This is done in memory so no page reload is required.

Also cookie, "*ThemePreference*" with the content "*yellow-light*" is added to your browser. So next time you launch Serene, it will remember your preference and start with a light yellow theme.

These skin files are located under "Content/adminlte/skins/" of the Serene.Web project. If you look there you can see files with names:

```
_all-skins.less  
skin.black-light.less  
site.blue.less  
site.yellow-light.less  
site.yellow.less
```

We are using LESS for CSS generation so you should try editing LESS files, not CSS. Next time you build your project, LESS files will be compiled to CSS (using *Less.js* compiler for *Node*).

This operation is configured with a build step in Serene.Web.csproj file:

```
...  
<Target Name="CompileSiteLess" AfterTargets="AfterBuild">  
  <Exec Command="&quot;$(ProjectDir)tools\node\lessc.cmd&quot;  
    &quot;$(ProjectDir)Content\site\site.less&quot; &gt;  
    &quot;$(ProjectDir)Content\site\site.css&quot;">  
  </Exec>  
</Target>  
...
```

Here *site.less* file is compiled to its corresponding css file in the same directory.

See <http://lesscss.org/> for more information on LESS compiler and its syntax.

# Localization

Serene allows you to change the active language from top right settings menu .

Try changing active language to Spanish.

The screenshot shows the Serene application interface. The top navigation bar is blue with the 'SERENE' logo on the left and a user profile 'admin' on the right. A dark sidebar on the left contains a search bar and a navigation menu with items like 'Salpicadero', 'Northwind', 'Clientes', 'Órdenes', 'Productos', 'Proveedores', 'Transportistas', 'Categorías', 'Regiones', 'Territorios', 'Theme Samples', and 'Administración'. The main content area is titled 'Customers' and features a search bar, a '+ Nueva Cliente' button, and 'Refrescar' and 'Excel' buttons. Below this are filters for 'País' and 'Ciudad'. A table displays a list of customer records with columns: ID, Nombre de Empresa, Nombre de Contacto, Contacto Título, Ciudad, Región, Código Postal, and País. The table contains 19 rows of data. At the bottom of the table, there is a pagination control showing 'Página 1 / 1' and 'Mostrando 1 de 91 de registros totales'. The footer of the page includes 'Copyright (c) 2015. All rights reserved.' and 'Serenity Platform'.

ID	Nombre de Empresa	Nombre de Contacto	Contacto Título	Ciudad	Región	Código Postal	País
ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Berlin		12209	Germany
ANATR	Ana Trujillo Emparedados y helados	Ana Trujillo	Owner	México D.F.		05021	Mexico
ANTON	Antonio Moreno Taquería	Antonio Moreno	Owner	México D.F.		05023	Mexico
AROUT	Around the Horn	Thomas Hardy	Sales Representative	London		WA1 1DP	UK
BERGS	Berglunds snabbköp	Christina Berglund	Order Administrator	Luleå		S-958 22	Sweden
BLAUS	Blauer See Delikatessen	Hanna Moos	Sales Representative	Mannheim		68306	Germany
BLONP	Blondesd'sl père et fils	Frédérique Citeaux	Marketing Manager	Strasbourg		67000	France
BOLID	Bóolido Comidas preparadas	Martín Sommer	Owner	Madrid		28023	Spain
BONAP	Bon app'	Laurence Leblhan	Owner	Marseille		13008	France
BOTTM	Bottom-Dollar Markets	Elizabeth Lincoln	Accounting Manager	Tsawassen	BC	T2F 8M4	Canada
BSBEV	B's Beverages	Victoria Ashworth	Sales Representative	London		EC2 5NT	UK
CACTU	Cactus Comidas para llevar	Patricio Simpson	Sales Agent	Buenos Aires		1010	Argentina
CENTC	Centro comercial Moctezuma	Francisco Chang	Marketing Manager	México D.F.		05022	Mexico
CHOPS	Chop-suey Chinese	Yang Wang	Owner	Bern		3012	Switzerland
COMMI	Comércio Mineiro	Pedro Afonso	Sales Associate	Sao Paulo	SP	05432-043	Brazil
CONSH	Consolidated Holdings	Elizabeth Brown	Sales Representative	London		WX1 6LT	UK

I don't speak Spanish and used machine translation, so sorry for errors...

When you changed the language, page is reloaded, unlike the theme selection where no page reload is required.

Serene, also added a cookie, *"LanguagePreference"* with content *"es"* to your browser, so next time you visit the site, it will remember your last selection and start with Spanish.

When you launched Serene first time, you might have seen the site in English, but it is also possible that it started in Spanish, Turkish or Russian (these are currently available sample languages) if you have an operating system or browser of that language.

This is controlled by a web.config setting:

```
<globalization culture="en-US" uiCulture="auto:en-US" />
```

Here we set UI culture to automatic, while en-US is a fallback (if system can't determine your browser language).

It is also possible to set another language as fallback:

```
<globalization culture="en-US" uiCulture="auto:tr-TR" />
```

Or set a language as default, whatever visiting users browser language is:

```
<globalization culture="en-US" uiCulture="es" />
```

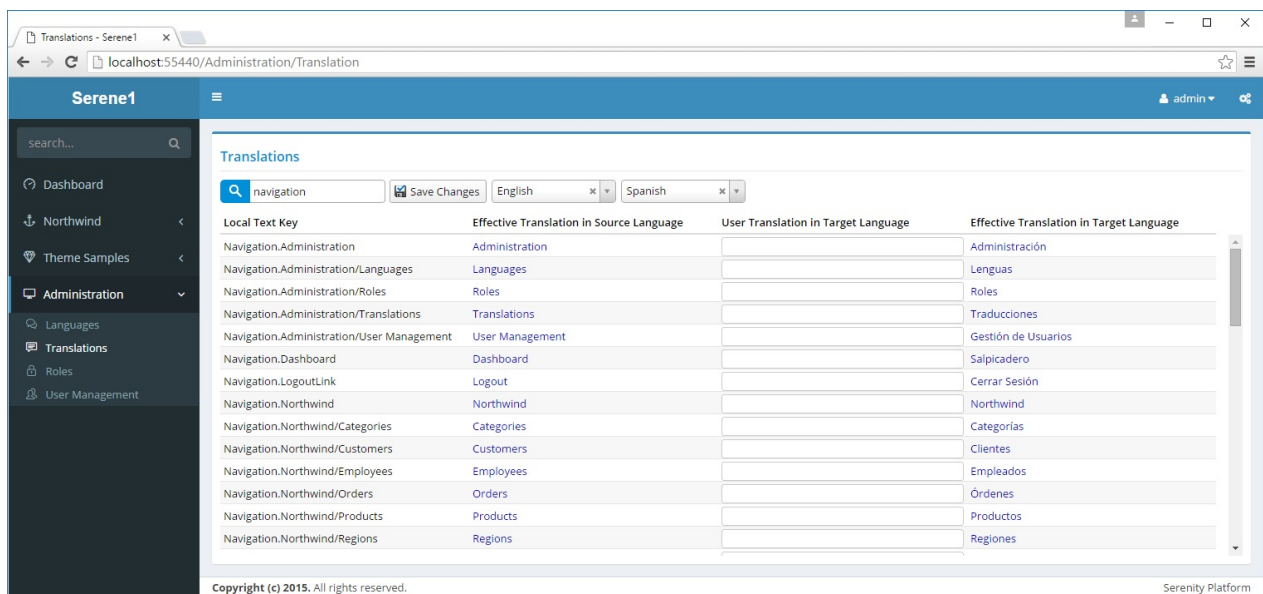
If you don't want to let users to change UI language, you should remove the language selection dropdown.

You may add more languages to the language selection dropdown by using Languages page under Administration menu.

## Localizing UI Texts

Serene includes ability to translate its text resources live.

Click *Administration* then *Translations* link in navigation.



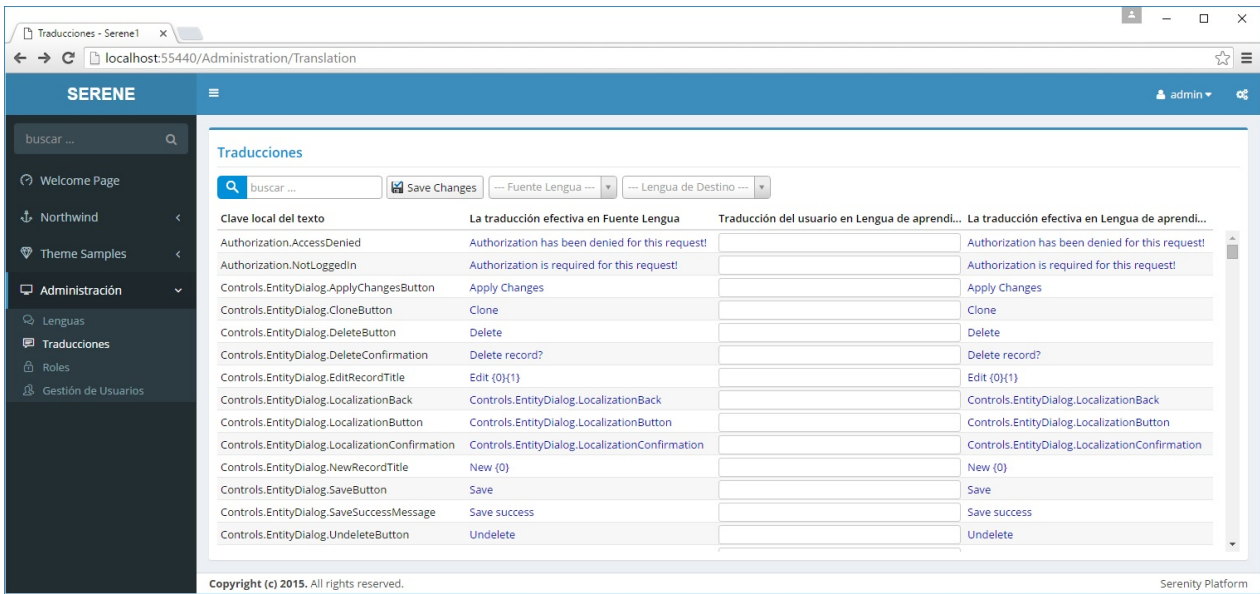
Type *navigation* into top left search box to see list of texts related to navigation menu.

Choose *English* as source language and *Spanish* as target language.

Type *Welcome Page* into line with *Navigation.Dashboard* local text key.

Click *Save Changes*.

When you switch to Spanish language, Dashboard menu item will be changed to *Welcome Page* instead of *Salpicadero*.



When you saved changes, Serene created a `user.texts.es.json` file in folder `App_Data/texts` with content like below:

```
{
  "Navigation.Dashboard": "Welcome Page"
}
```

In the `~/scripts/site/texts` folder, there are also other similar JSON files with default translations for Serene interface:

- `site.texts.es.json`
- `site.texts.invariant.json`
- `site.texts.tr.json`

It is recommended to transfer your translations from `user.texts.xx.json` files to `site.texts.xx.json` files before publishing. You can also keep them under version control this way, if `App_Data` folder is ignored.

# User and Role Management

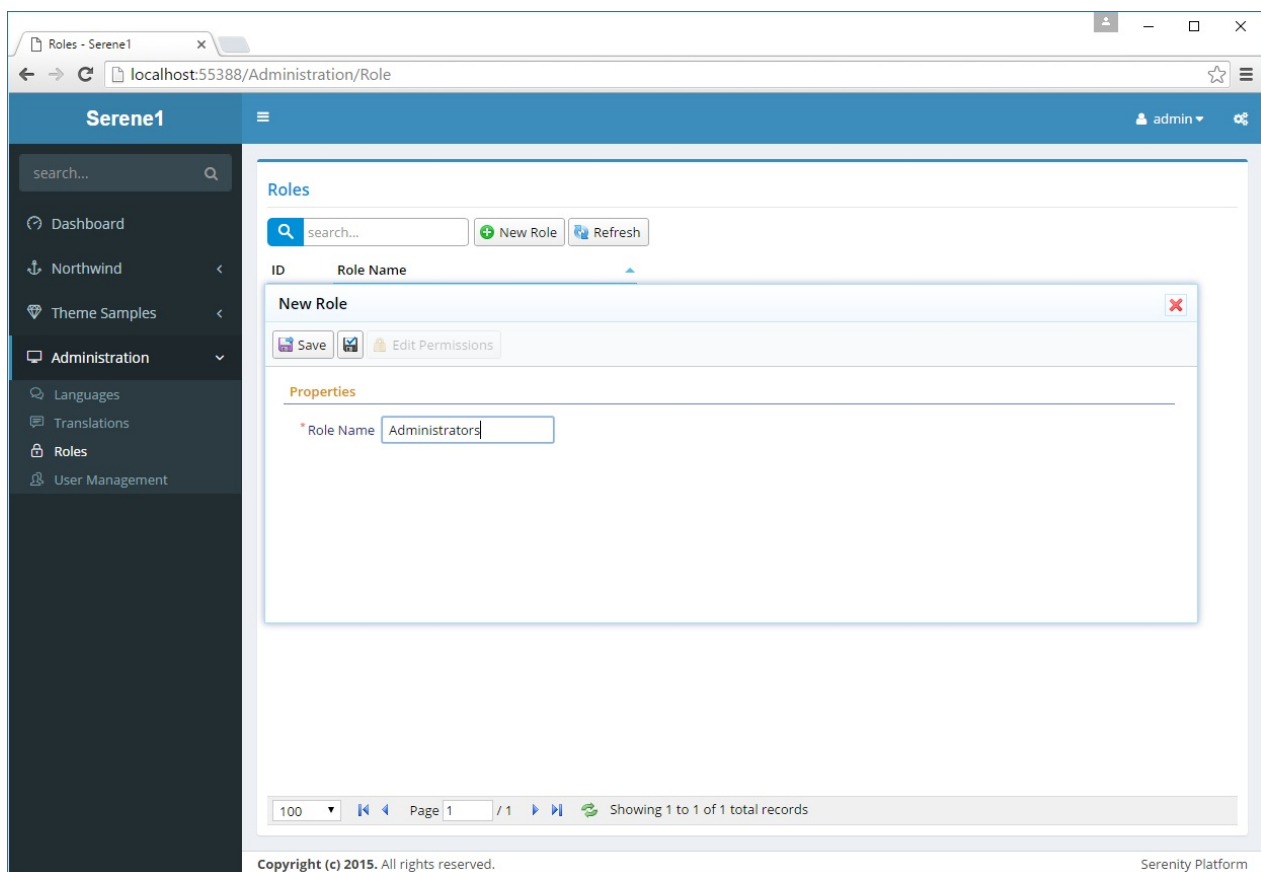
Serene has user, role and rights management built in.

This feature is not embedded in Serenity itself. It is just a sample, so you can always implement and use your user management of choice. We'll take a look at how in following chapters.

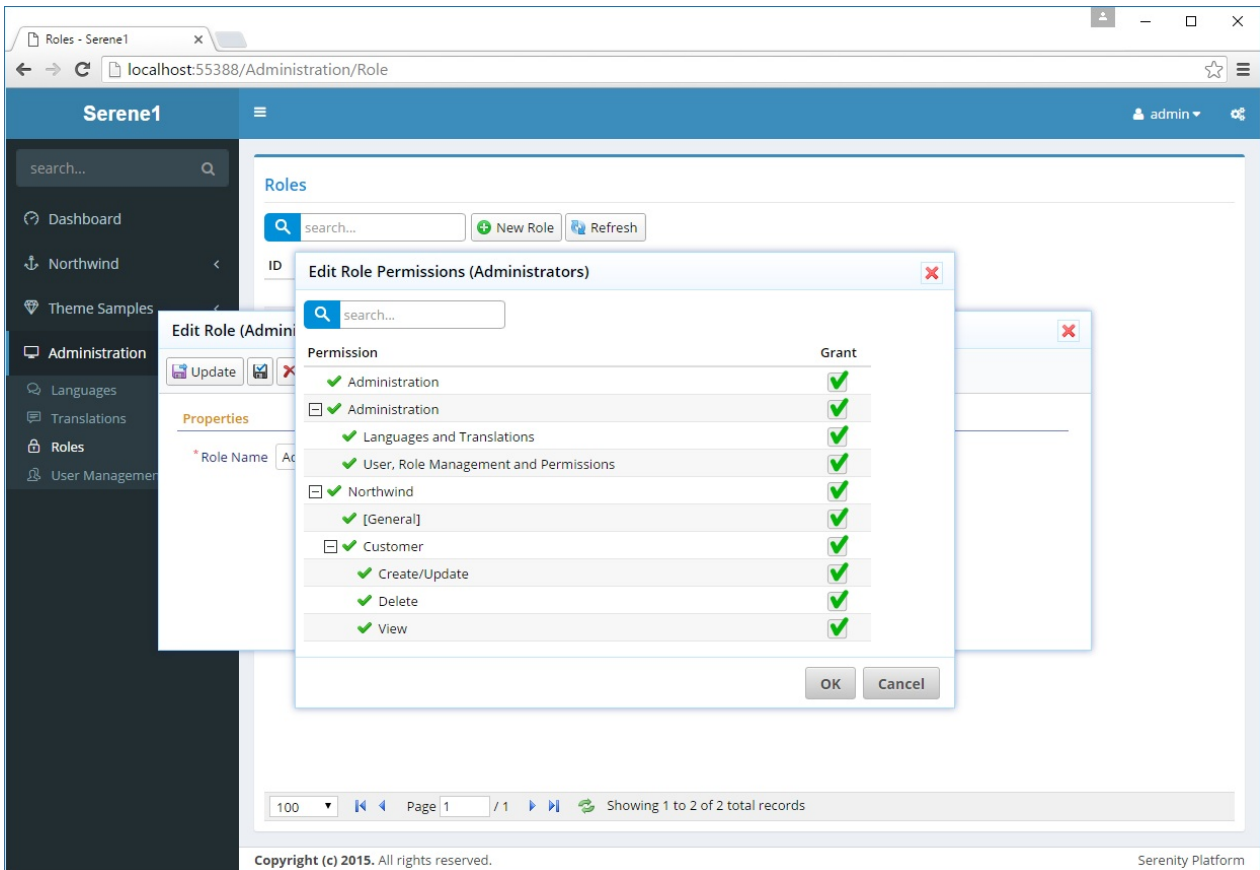
Open Administration / Roles to create roles *Administrators* and *Translators*.

Click *New Role* and type *Administrators*, then click *Save*.

Repeat it for *Translators*.



Then click role *Administrators* to open edit form, and click *Edit Permissions* button to modify its permissions. Check all boxes to grant every permission to this role, then click *OK*.

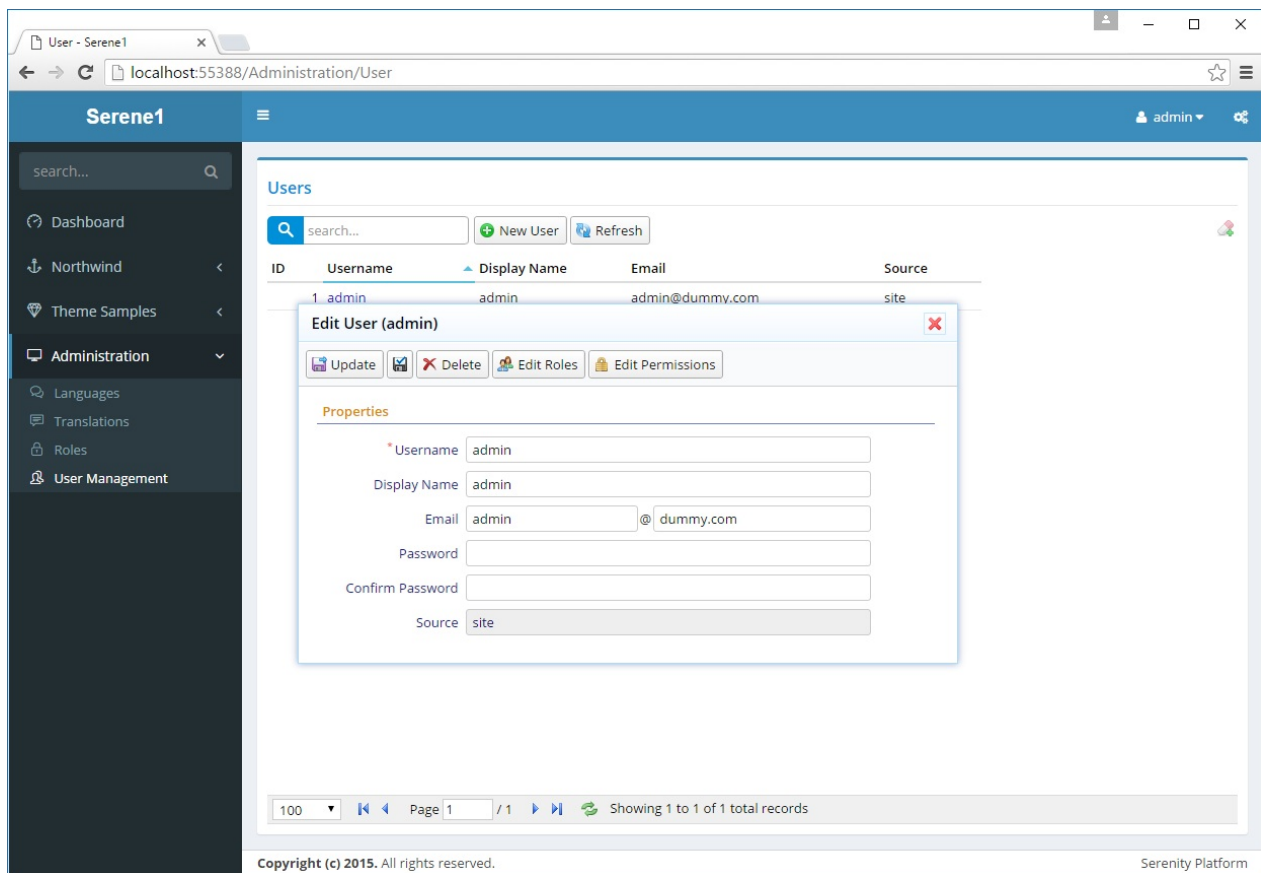


Repeat same steps for the *Translations* role but this time grant only the *Administration: Languages and Translations* permission.

Navigate to *Administration / User Management* page to add more users.

Click *admin* user to edit its details.





Here you can change admin details like username, display name, email.

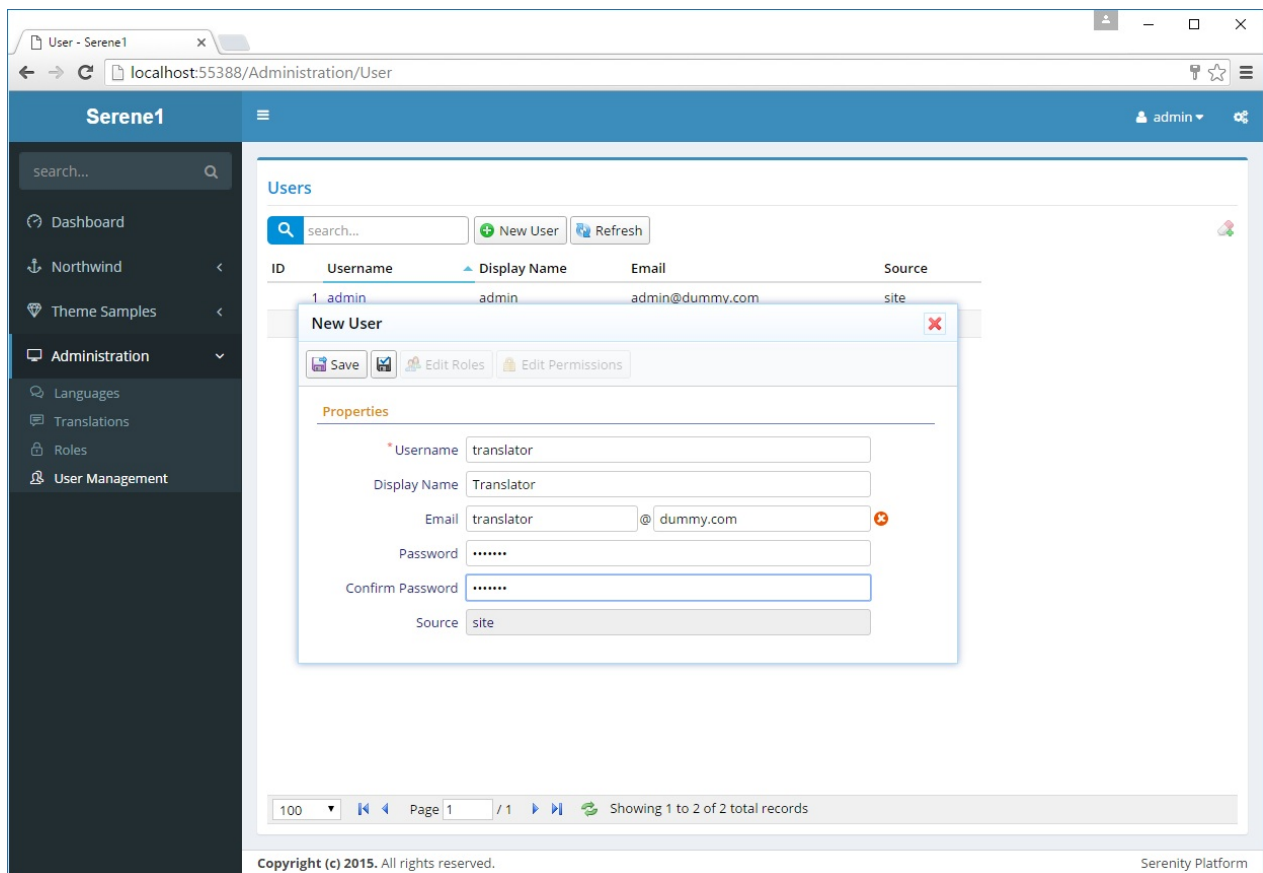
You can also change its password (which is *serenity* by default) by typing into *Password* and *Confirm Password* inputs and clicking *Update*.

You can also delete it but this would make your site unusable as you wouldn't be able to login.

*admin* is a special user in Serene, as it has all permissions even if none is explicitly granted to him.

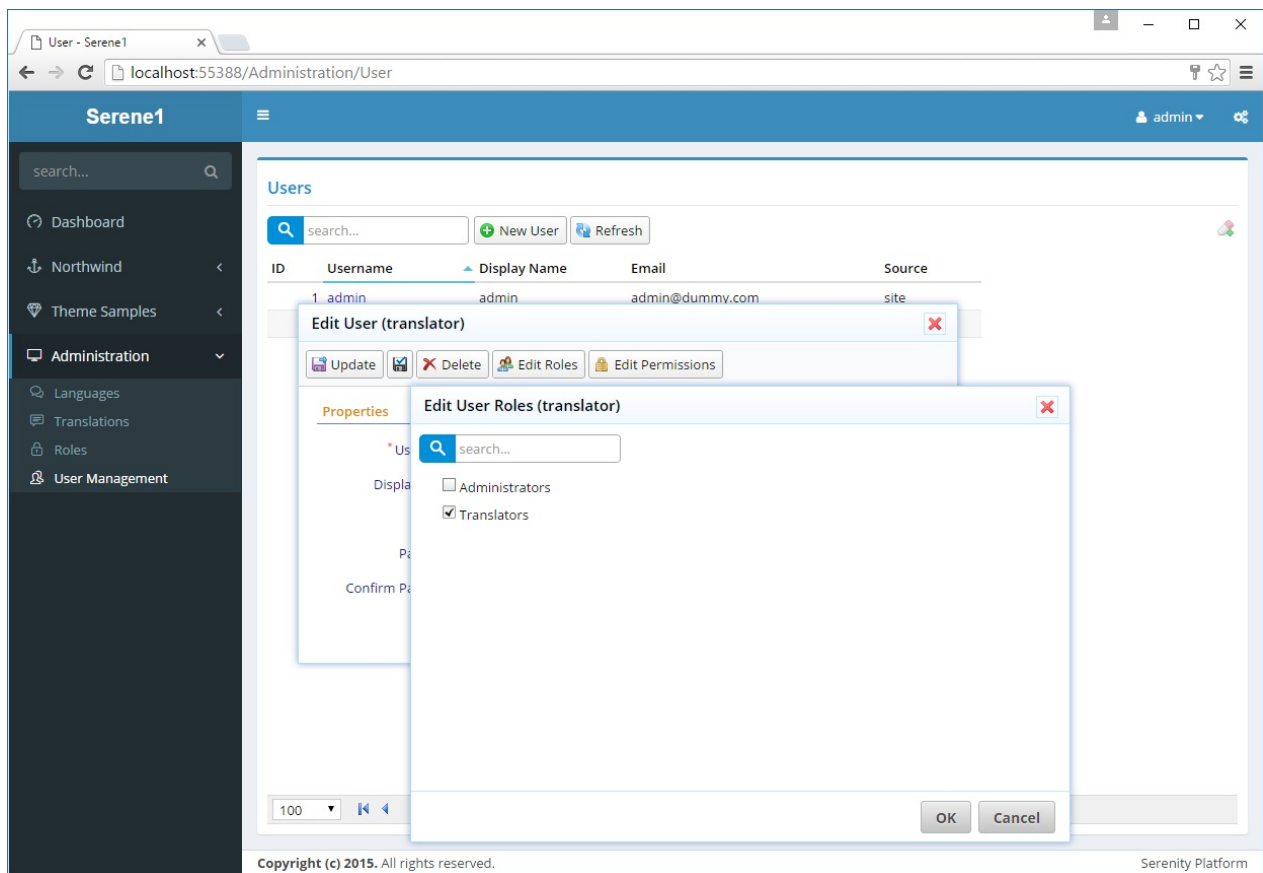
Lets create another one and grant roles / permissions to it.

Close this dialog, click new user and type *translator* as username. Fill in other fields as you'd like, then click *Update*.



You may have noticed there is a *Apply Changes* button with a black disk icon without title, next to *Save*. Unlike *Save*, when you use it, the form stays open, so you can see how your record looks like after saving, also you can edit roles and permissions before closing the form.

Now click *Translator* role to open its edit form and click *Edit Roles*. Grant him *Translators* role and click *OK*.



When you grant a role to a user, he gets all permissions granted to the role automatically. By clicking Edit Permissions and you can also grant extra permissions explicitly. You can also revoke a role permission from a user explicitly.

Now close all dialogs and logout by clicking *admin* on top right of site and clicking *Logout*.

Try logging in with *translator* and the password you set.

Translator user will only have access to Dashboard, Theme Samples, Languages and Translations pages.

The screenshot shows the Serenity1 Administration/Translation interface. The page title is "Translations" and the user is logged in as "translator". The interface includes a search bar, a "Save Changes" button, and dropdown menus for "Source Language" and "Target Language". The main content is a table with the following columns: "Local Text Key", "Effective Translation in Source Language", "User Translation in Target Language", and "Effective".

Local Text Key	Effective Translation in Source Language	User Translation in Target Language	Effective
Authorization.AccessDenied	Authorization has been denied for this request!		Autho
Authorization.NotLoggedIn	Authorization is required for this request!		Autho
Controls.EntityDialog.ApplyChangesButton	Apply Changes		Apply
Controls.EntityDialog.CloneButton	Clone		Clone
Controls.EntityDialog.DeleteButton	Delete		Delete
Controls.EntityDialog.DeleteConfirmation	Delete record?		Delete
Controls.EntityDialog.EditRecordTitle	Edit {0}{1}		Edit (C
Controls.EntityDialog.LocalizationBack	Controls.EntityDialog.LocalizationBack		Contr
Controls.EntityDialog.LocalizationButton	Controls.EntityDialog.LocalizationButton		Contr
Controls.EntityDialog.LocalizationConfirmation	Controls.EntityDialog.LocalizationConfirmation		Contr
Controls.EntityDialog.NewRecordTitle	New {0}		New {0
Controls.EntityDialog.SaveButton	Save		Save
Controls.EntityDialog.SaveSuccessMessage	Save success		Save s
Controls.EntityDialog.UndeleteButton	Undelete		Undel
Controls.EntityDialog.UndeleteConfirmation	Undelete record?		Undel
Controls.EntityDialog.UpdateButton	Update		Updat
Controls.EntityGrid.IncludeDeletedToggle	display inactive records		displa
Controls.EntityGrid.NewButton	New {0}		New {0
Controls.EntityGrid.RefreshButton	Refresh		Refres

Copyright (c) 2015. All rights reserved. Serenity Platform

# Listing Pages

Serene has listing pages and editing interface for Northwind database. Let's have a look at the Products page under Northwind module.

The screenshot shows the Serene1 application interface. The main content area displays a table of products. The table has the following columns: ID, Product Name, Dis..., Supplier, Category, Quantity Per Unit, Unit Price, Units In S..., Units On ..., and Reorder L... The products are sorted by Product Name. The table contains 15 rows of data. The interface also includes a search bar, a navigation menu on the left, and pagination controls at the bottom of the table.

ID	Product Name	Dis...	Supplier	Category	Quantity Per Unit	Unit Price	Units In S...	Units On ...	Reorder L...
17	Alice Mutton	<input checked="" type="checkbox"/>	Pavlova, Ltd.	Meat/Poultry	20 - 1 kg tins	39	0	0	0
3	Aniseed Syrup	<input type="checkbox"/>	Exotic Liquids	Condiments	12 - 550 ml bottles	10	13	70	25
40	Boston Crab Meat	<input type="checkbox"/>	New England Seafood Cannery	Seafood	24 - 4 oz tins	18.4	123	0	30
60	Camembert Pierrot	<input type="checkbox"/>	Gai pâturage	Dairy Products	15 - 300 g rounds	34	19	0	0
18	Carnarvon Tigers	<input type="checkbox"/>	Pavlova, Ltd.	Seafood	16 kg pkg.	62.5	42	0	0
1	Chai	<input type="checkbox"/>	Exotic Liquids	Beverages	10 boxes x 20 bags	18	39	0	10
2	Chang	<input type="checkbox"/>	Exotic Liquids	Beverages	24 - 12 oz bottles	19	17	40	25
39	Chartreuse verte	<input type="checkbox"/>	Aux joyeux ecclésiastiques	Beverages	750 cc per bottle	18	69	0	5
4	Chef Anton's Cajun Seasoning	<input type="checkbox"/>	New Orleans Cajun Delights	Condiments	48 - 6 oz jars	22	53	0	0
5	Chef Anton's Gumbo Mix	<input checked="" type="checkbox"/>	New Orleans Cajun Delights	Condiments	36 boxes	21.35	0	0	0
48	Chocolade	<input type="checkbox"/>	Zaanse Snoepfabriek	Confections	10 pkgs.	12.75	15	70	25
38	Côte de Blaye	<input type="checkbox"/>	Aux joyeux ecclésiastiques	Beverages	12 - 75 cl bottles	263.5	17	0	15
58	Escargots de Bourgogne	<input type="checkbox"/>	Escargots Nouveaux	Seafood	24 pieces	13.25	62	0	20
52	Filo Mix	<input type="checkbox"/>	G'day, Mate	Grains/Cereals	16 - 2 kg boxes	7	38	0	25
71	Flotemysost	<input type="checkbox"/>	Norske Meierier	Dairy Products	10 - 500 g pkgs.	21.5	26	0	0

Here we see list of products sorted by product name (initial sort order).

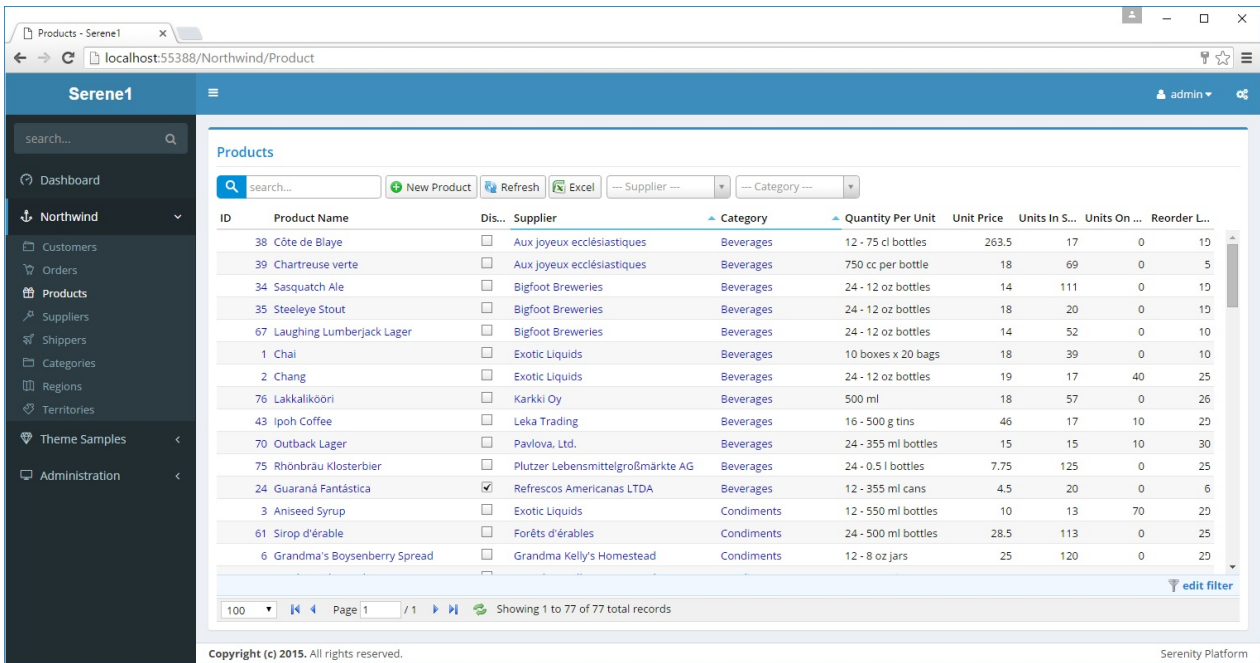
Grid component is SlickGrid with a customized theme.

<https://github.com/mleibman/SlickGrid>

You can change order by clicking column headers. To sort descending, click the same column header again.

To sort by multiple columns, you can use Shift+Click.

Here is what it looks like after sorting by Category then Supplier columns:



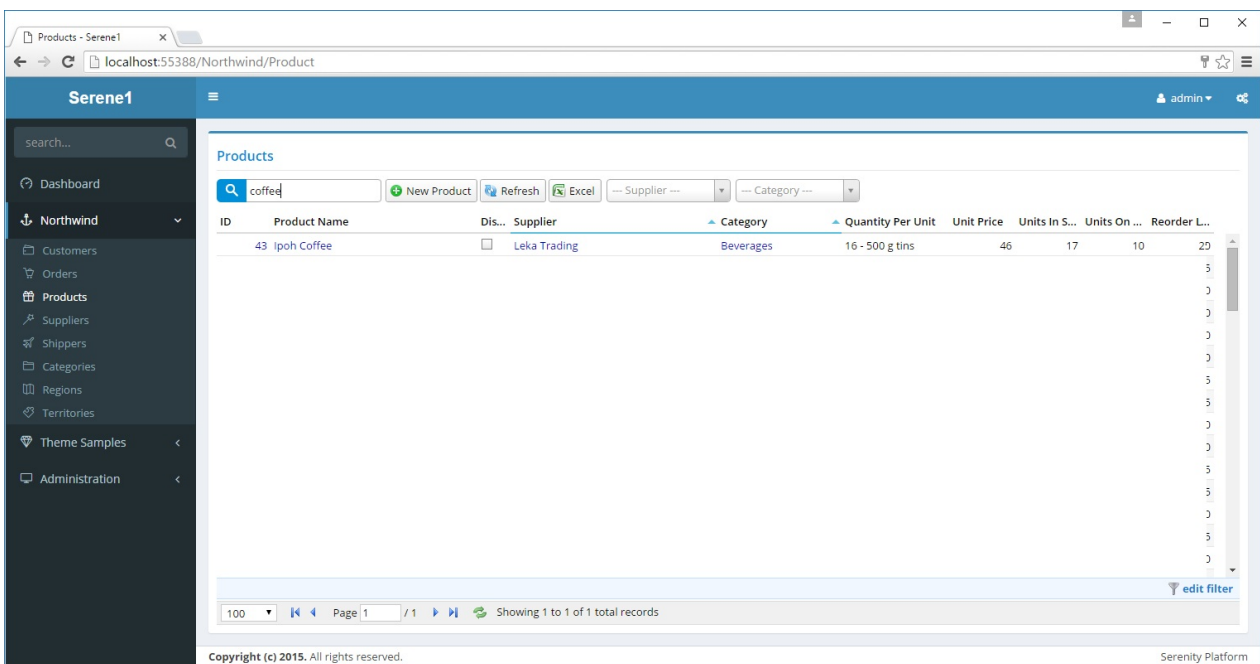
When you changed sort order, grid loaded data from a service with an AJAX request.

When you open the page first time, initial records were also loaded by an AJAX call.

By default grid loads records by 100 page size. Only records in current page are loaded from server. In the sample image, i changed page size to 20 (bottom left of grid) to show paging in effect.

On top left of the grid, you can type something to do a simple search.

Type *coffee* for example to see products containing it in their names.



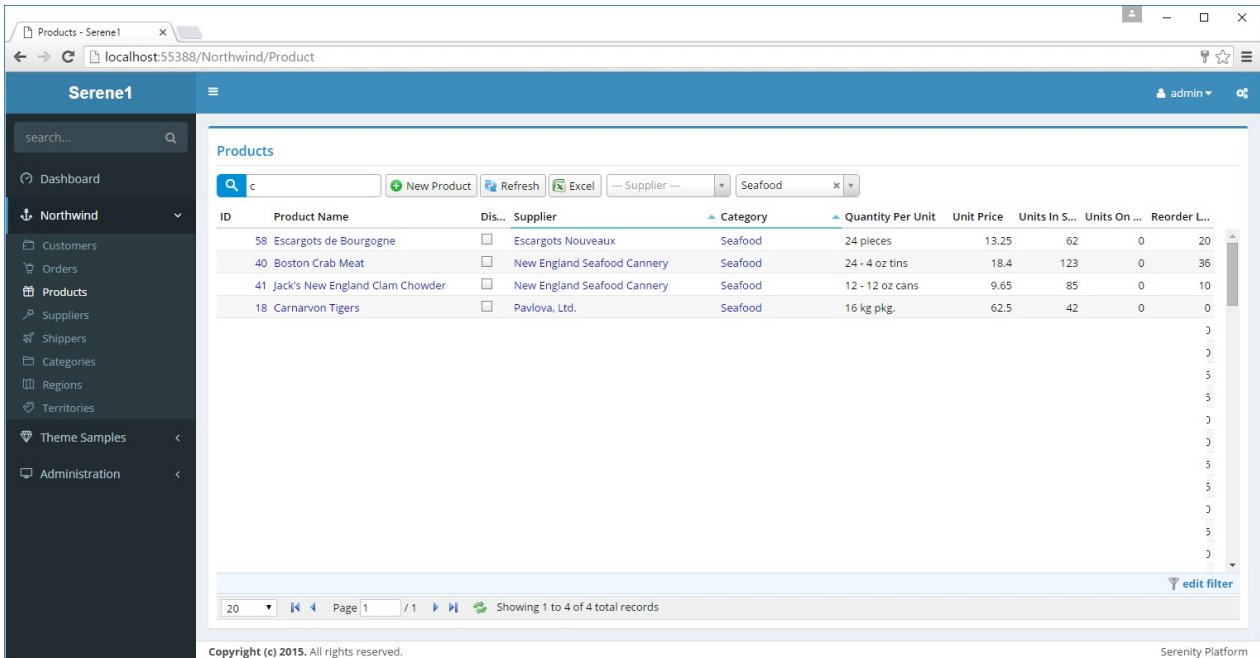
It searched in product name field. It is also possible to use another, or multiple fields for quick search. We'll see how in later chapters.

On top right of the grid, there are quick filtering dropdowns for *Supplier* and *Category* fields.

Dropdown component used is Select2

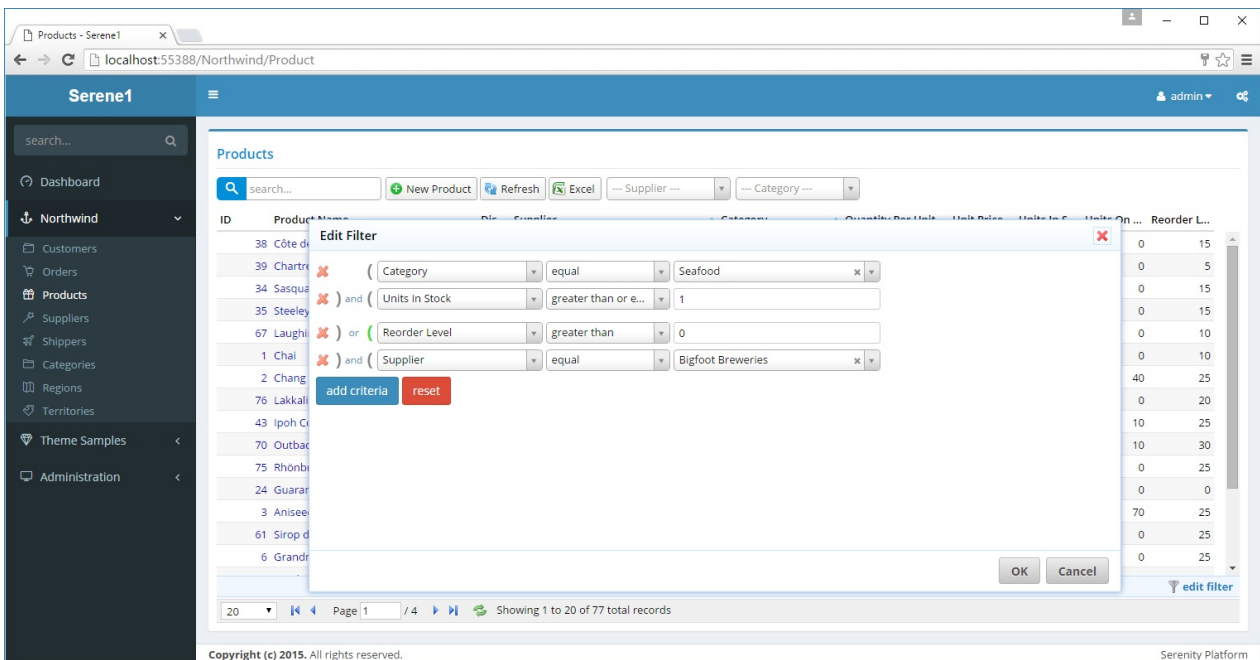
<https://github.com/select2/select2>

Choose *Seafood* as *Category* and it will show only products in that category.



All sorting, paging and filtering is done on server side with dynamic SQL queries generated by Serenity service handlers.

It is also possible to filter by any column by clicking *edit filter* link at bottom right of the grid.



Here you can add criteria by any column by clicking *add criteria* and choosing column name, choosing comparison operator from next dropdown, and setting a value.

Some value editors are simple textboxes while some others may have dropdowns and other custom editors depending on column type.

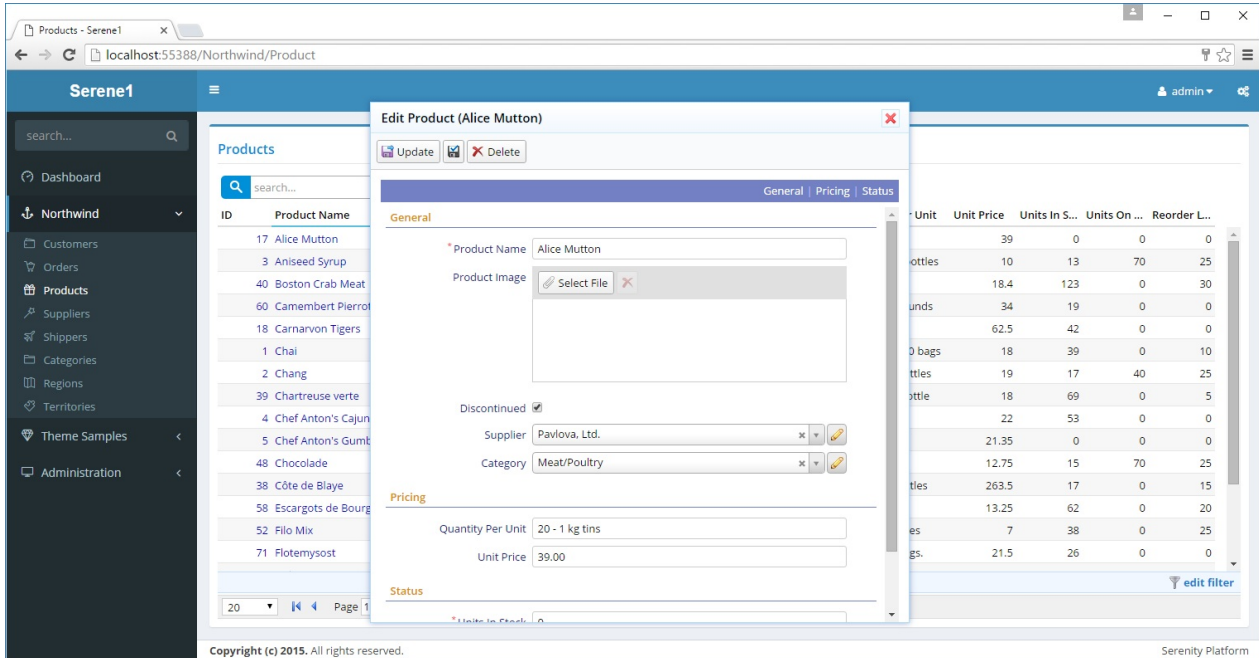
It is also possible to change *and* to *or* by clicking on it.

You can also group criteria by clicking parenthesis. Groups will have a bit more space between them than ordinary lines.



# Edit Dialogs

When you click a product name in Products page, an editing dialog for that row is displayed.



This dialog is shown on client side, there is no post-back happening. Data for the clicked entity is loaded from server side using an AJAX request (only data, not markup). Dialog itself is a customized version of jQuery UI dialog.

In this form we have three categories for fields: *General*, *Pricing* and *Status*. By clicking category links on top blue bar you can navigate to start of that category.

Each form field occupies a row with label and editor. You may choose to show more than one field in a row if required (with CSS).

Fields marked with "\*" are required (cannot be empty).

Each field has a specific type of editor tailored to its data type like *string*, *image upload*, *checkbox*, *select* etc.

We would see such an HTML code if we looked at the source (simplified for brevity):

```
<div class="field ProductName">
  <label>Product Name</label>
  <input type="text" class="editor s-StringEditor" />
</div>

<div class="field ProductImage">
  <label class="caption"> Product Image</label>
  <div class="editor s-ImageUploadEditor">
    ...
  </div>
</div>

...
```

Every field has a separate "div" of its own with class "field". Inside this div, there is a "label" element and another element (input, select, div) that changes with the editor type for that field.

We can look at the class names of these elements to identify their editor types (e.g. *s-StringEditor*, *s-ImageUploadEditor*)

In the toolbar we have a button to save current entity and close dialog (*Update*), next to it a smaller one that keeps dialog open after save and another one to delete current entity (obviously).

Most Serenity editing dialogs has this familiar interface, though you can always customize buttons, fields, add tabs, and other interface elements.

# Tutorials

- [Movie Database \(similar to IMDB\)](#)
- [Multi Tenancy](#)

# Tutorial: Movie Database

Let's create editing interface for some site similar to IMDB with Serenity.

You can find source code for this tutorial at:

<https://github.com/volkanceylan/MovieTutorial>

## Create a new project named *MovieTutorial*

In Visual Studio click File -> New Project. Make sure you choose *Serene* template. Type *MovieTutorial* as name and click *OK*.

You might also choose Serene (AspNetCore) version. A few things will be different. We'll try to list those differences.

In Solution explorer, you should see one project with name *MovieTutorial.Web*.

*MovieTutorial.Web* project is an ASP.NET MVC (or Core) application that contains server side code plus static resources like css files, images etc.

*MovieTutorial.Web* also has a *tsconfig.json* file at root, which specifies that it is also a TypeScript project. All *.ts* files under *Modules/* and *Scripts/* directories are compiled on save, and their outputs are combined into a script file under *scripts/site/* folder with name *MovieTutorial.Web.js*.

Please make sure that you have TypeScript 2.5.2+ installed. Check your version from Visual Studio Extensions dialog.

To install TypeScript 2.5.2+ in Visual Studio, you'll need to install Visual Studio 2015 Update 3.

Download its latest version from <http://www.typescriptlang.org/#download-links> for your Visual Studio.

Also check prerequisites in Getting Started section.

## Creating *Movie* Table

To store list of movies we need a *Movie* table. We could create this table using old-school techniques like SQL Management Studio but we prefer creating it as a *migration* using *Fluent Migrator*:

Fluent Migrator is a migration framework for .NET much like Ruby on Rails Migrations. Migrations are a structured way to alter your database schema and are an alternative to creating lots of sql scripts that have to be run manually by every developer involved. Migrations solve the problem of evolving a database schema for multiple databases (for example, the developer's local database, the test database and the production database). Database schema changes are described in classes written in C# that can be checked into a version control system.

See <https://github.com/schambers/fluentmigrator> for more information on FluentMigrator.

### Please Note

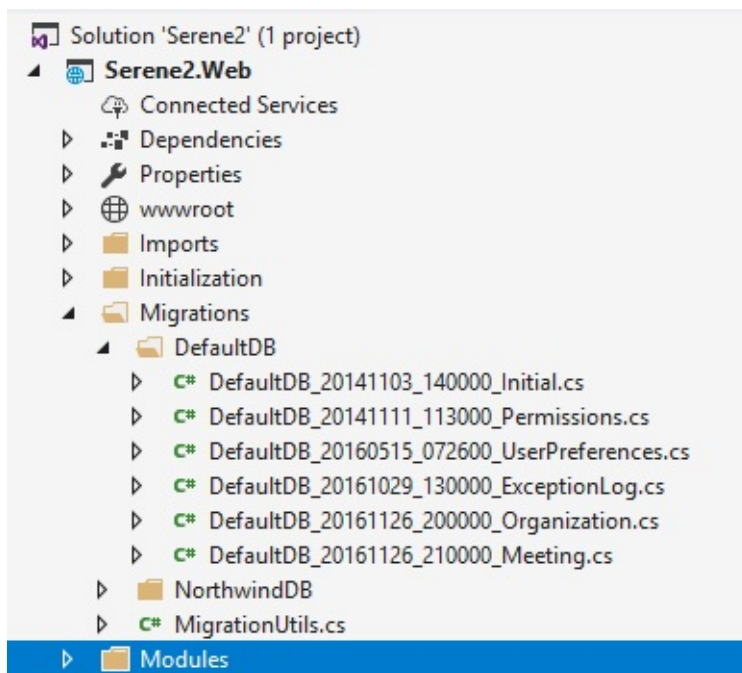
As we are using FluentMigrator in our samples, some users assume Serenity doesn't work without it. That's not correct. You don't have to use migrations. Serenity has no direct dependency on migrations.

If you like, instead of using these migrations you may manually create tables in SQL Management Studio.

You could also work with an existing database.

## Locating Migration Folder

Using *Solution Explorer* navigate to *Migrations / DefaultDB*.



Here we already have several migrations. A migration is like a DML script that manipulates database structure.

*DefaultDB\_20141103\_140000\_Initial.cs* for example, contains our initial migration that created *Northwind* tables and *Users* table.

Create a new migration file in the same folder with name *DefaultDB\_20160519\_115100\_MovieTable.cs*. You can copy and change one of the existing migration files, rename it and change contents.

Migration file name / class name is actually not important but recommended for consistency and correct ordering.

*20160519\_115100* corresponds to the time we are writing this migration in *yyyyMMdd\_HHmms* format. It will also act as a unique key for this migration.

Our migration file should look like below:

```
using FluentMigrator;
using System;

namespace MovieTutorial.Migrations.DefaultDB
{
    [Migration(20160519115100)]
    public class DefaultDB_20160519_115100_MovieTable : Migration
    {
        public override void Up()
        {
            Create.Schema("mov");

            Create.Table("Movie").InSchema("mov")
                .WithColumn("MovieId").AsInt32()
                    .Identity().PrimaryKey().NotNullable()
                .WithColumn("Title").AsString(200).NotNullable()
                .WithColumn("Description").AsString(1000).Nullable()
                .WithColumn("Storyline").AsString(Int32.MaxValue).Nullable()
                .WithColumn("Year").AsInt32().Nullable()
                .WithColumn("ReleaseDate").AsDateTime().Nullable()
                .WithColumn("Runtime").AsInt32().Nullable();
        }

        public override void Down()
        {
        }
    }
}
```

Make sure you use the namespace *MovieTutorial.Migrations.DefaultDB* as Serene template applies migrations only in this namespace to the default database.

In *Up()* method we specify that this migration, when applied, will create a schema named *mov*. We will use a separate schema for movie tables to avoid clashes with existing tables. It will also create a table named *Movie* with "MovieId, Title, Description..." fields on it.

We could implement *Down()* method to make it possible to undo this migration (drop movie table and mov schema etc), but for the scope of this sample, lets leave it empty.

Inability to undo a migration might not hurt much, but deleting a table by mistake could do more damage.

On top of our class we applied a Migration attribute.

```
[Migration(20160519115100)]
```

This specifies a unique key for this migration. After a migration is applied to a database, its key is recorded in a special table specific to FluentMigrator (`[dbo].[VersionInfo]`), so same migration won't be applied again.

Migration key should be in sync with class name (for consistency) but without underscore as migration keys are Int64 numbers.

Migrations are executed in the key order, so using a sortable datetime pattern like `yyyyMMdd` for migration keys looks like a good idea.

Please make sure you always use same number of characters for migration keys e.g. 14 (20160519115100). Otherwise your migration order will get messed up, and you will have migration errors, due to migrations running in unexpected order.

## Running Migrations

By default, Serene template runs all migrations in `MovieTutorial.Migrations.DefaultDB` namespace. This happens on application start automatically.

The code that runs migrations are in `App_Start/SiteInitialization.cs` and `App_Start/SiteInitialization.Migrations.cs` files:

In Asp.Net Core version, they are under `Initialization/Startup.cs` and `Initialization/DataMigrations.cs` files as there is no `App_Start` folder in ASP.NET Core.

### SiteInitialization.Migrations.cs (or DataMigrations.cs):

```
namespace MovieTutorial
{
    //...

    public static partial class SiteInitialization
    {
        private static string[] databaseKeys = new[] { "Default", "Northwind" };

        //...
        private static void EnsureDatabase(string databaseKey)
        {
            //...
        }

        public static bool SkippedMigrations { get; private set; }

        private static void RunMigrations(string databaseKey)
        {
            // ...
            // safety check to ensure that we are not modifying an
```



```
// arbitrary database. remove these two lines if you want
// MovieTutorial migrations to run on your DB.
if (cs.ConnectionString.IndexOf(typeof(SiteInitialization).Namespace +
    @"_" + databaseKey + "_v1",
    StringComparison.OrdinalIgnoreCase) < 0)
{
    SkippedMigrations = true;
    return;
}

// ...

using (var sw = new StringWriter())
{
    // ...
    var runner = new RunnerContext(announcer)
    {
        Database = databaseType,
        Connection = cs.ConnectionString,
        Targets = new string[] {
            typeof(SiteInitialization).Assembly.Location },
        Task = "migrate:up",
        WorkingDirectory = Path.GetDirectoryName(
            typeof(SiteInitialization).Assembly.Location),
        Namespace = "MovieTutorial.Migrations." + databaseKey + "DB"
    };

    new TaskExecutor(runner).Execute();
}
}
}
```

There is a safety check on database name to avoid running migrations on some arbitrary database other than the default Serene database (MovieTutorial\_Default\_v1). You can remove this check if you understand the risks. For example, if you change Northwind connection in web.config to your own production database, migrations will run on it and you will have Northwind etc tables even if you didn't mean to.

Now press F5 to run your application and create Movie table in default database.

## Verifying That the Migration is Run

Using Sql Server Management Studio or Visual Studio -> Connection To Database, open a connection to MovieTutorial\_Default\_v1 database in server (*localdb*)\MsSqlLocalDB or (*localdb*)\v11.0 depending on version you use.

(localdb)\v11.0 is a LocalDB instance created by SQL Server 2012 LocalDB.

(localdb)\MsSqlLocalDB is an instance created by SQL 2014+ LocalDB.

If you didn't install LocalDB yet, download it from <https://www.microsoft.com/en-us/download/details.aspx?id=29062>.

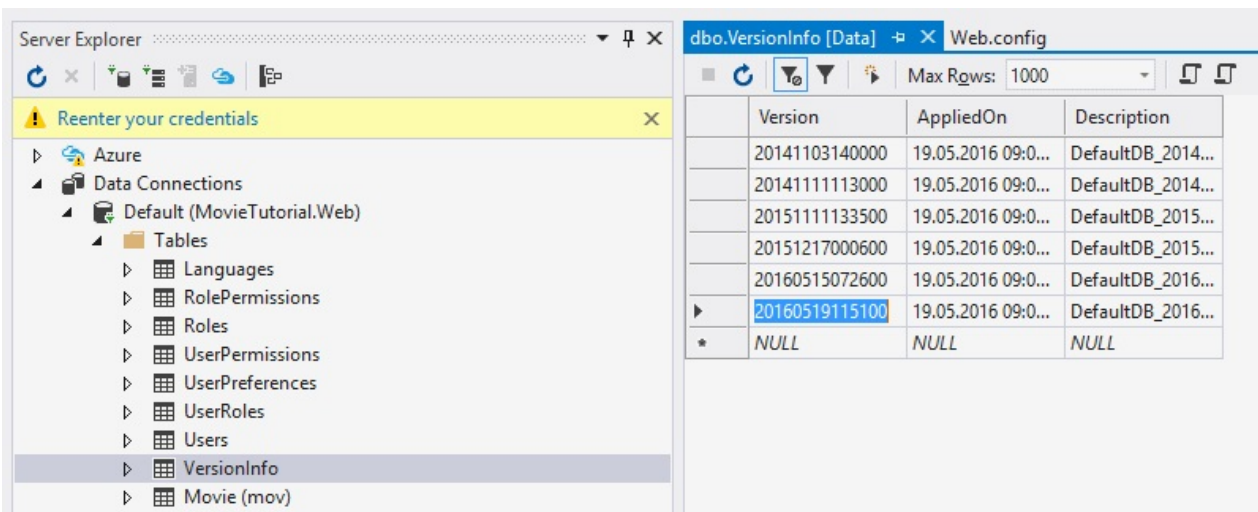
If you have SQL Server 2014 LocalDB, your server name would be (localdb)\MSSqlLocalDB or (localdb)\v12.0, so change connection string in web.config file.

You could also use another SQL server instance, just change the connection string to target server and remove the migration safety check.

You should see `[mov].[Movies]` table in SQL object explorer.

Also when you view data in `[dbo].[VersionInfo]` table, Version column in the last row of the table should be `20160519115100`. This specifies that the migration with that version number (migration key) is already executed on this database.

So, even if you change migration source code, that migration won't ever run again in this database. Try to avoid modifying migrations after they run on your DB. Create a new migration if possible.



Usually, you don't have to do these checks after every migration. Here we show these to explain where to look, in case you'll have any trouble in the future.

# Generating Code For Movie Table

## Serenity Code Generator (ASP.NET MVC)

These steps applies only to ASP.NET MVC version, not ASP.NET Core version. Keep reading to see how to run Sergen in ASP.NET Core version.

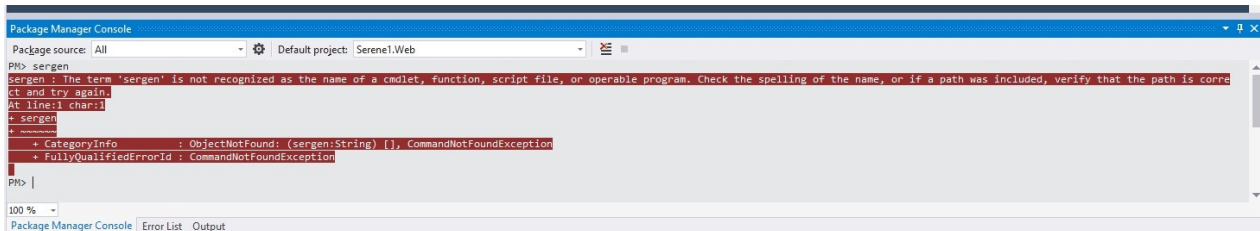
After making sure that our table exists in the database, we will use Serenity Code Generator (`sergen.exe`) to generate initial editing interface.

In Visual Studio, open *Package Manager Console* by clicking *View => Other Windows => Package Manager Console*.

Type `sergen` and press Enter.

### Resolving Sergen is not Recognized Issue

Sometimes NuGet package manager can't set PATH correctly and you may get an error like below while trying to execute Sergen.

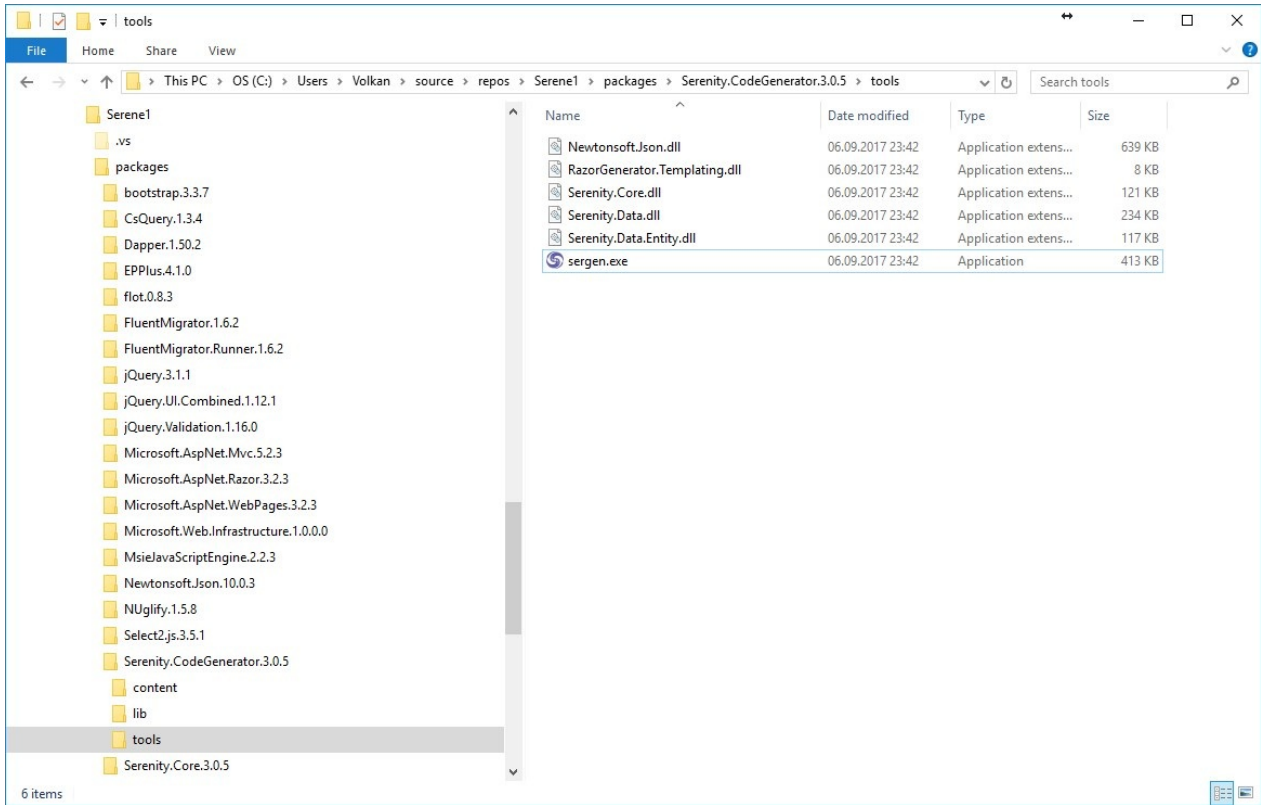


```
Package Manager Console
Package source: All Default project: Serene1.Web
PM> sergen
Sergen : The term 'sergen' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the spelling of the name, or if a path was included, verify that the path is correct and try again.
At line:1 char:1
+ sergen
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (sergen:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException
PM> |
100 %
Package Manager Console | Error List | Output
```

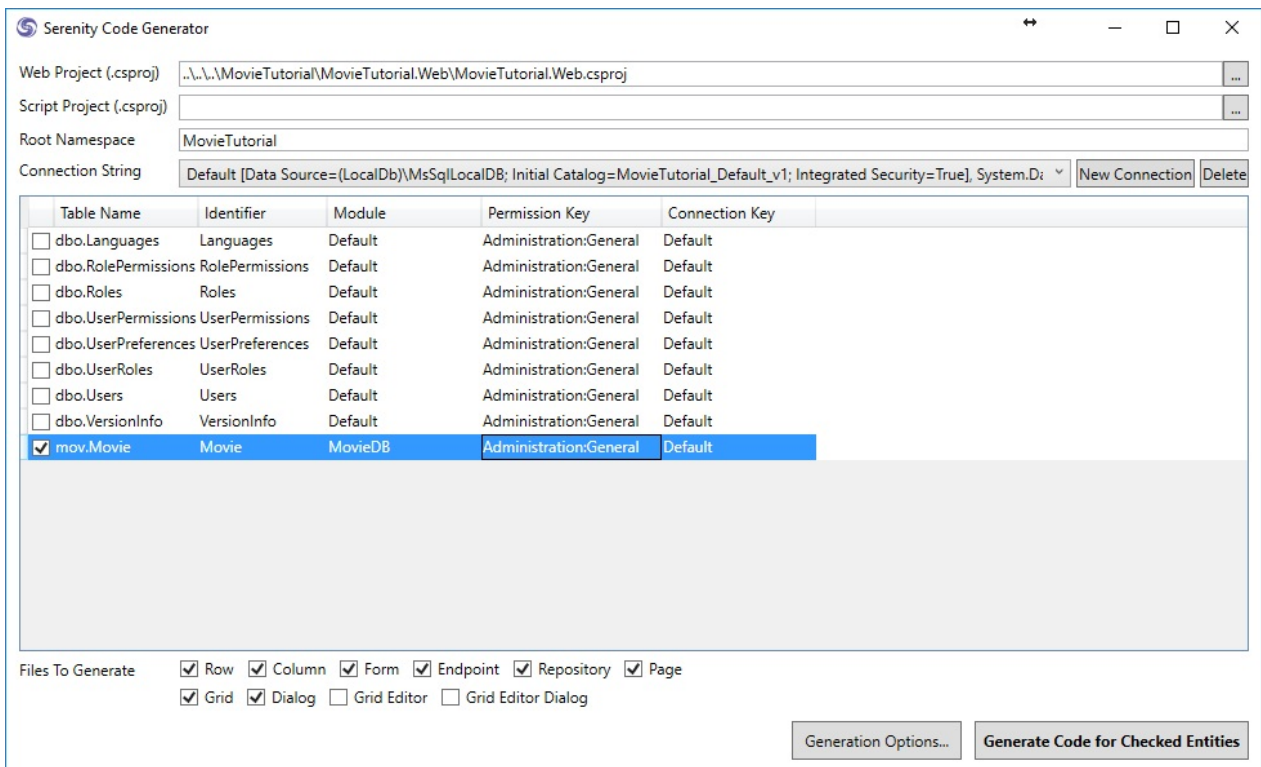
Unfortunately, this is a bug of Visual Studio / NuGet and is not related to Serenity or Sergen itself.

Most of the times, restarting Visual Studio might resolve the issue.

If it doesn't, you may open `Sergen.exe` from Windows Explorer. Right click on *MovieTutorial* solution in Solution Explorer, click *Open In File Explorer*. `Sergen.exe` is under `packages\Serenity.CodeGenerator.X.Y.Z\tools` directory.



## Sergen UI



## Setting Project Location

When you first run Sergen, Web Project field will be prefilled for you to:

- ..\..\..\MovieTutorial\MovieTutorial.Web\MovieTutorial.Web.csproj

If you change this value and other options, and generate your first page, you won't have to set them again. All these options will be saved in *Serenity.CodeGenerator.config* in your solution directory.

This value is required, as Sergen will automatically include generated files to your WEB project.

Script project field should be empty for v2.1+. This is for users of older Serene, who might still have code that was written with Saltaralle compiler, instead of TypeScript.

## Root Namespace Option

Your root namespace option is set to the Solution name you used, e.g. *MovieTutorial*. If your project name is *MyProject.Web*, your root namespace is *MyProject* by default.

This is critical so make sure you don't set it to anything different, as by default, Serene template expects all generated code to be under this root namespace.

It is also very important to understand that Root Namespace is case sensitive and must exactly match your project name, e.g. not *movietutorial* or *movieTutorial* but *MovieTutorial*.

This option is also saved, so next time you won't have to fill it in.

## Choosing Connection String

Once you set Web project name, Sergen populates connection dropdown with connection strings from your web.config file. We might have *Default* and *Northwind* in there. Choose *Default* one.

## Selecting Table To Generate Code For

Sergen can generate code for multiple tables, but we'll generate for only one now. Once we choose connection string, table grid is populated with table names from that database.

Mark checkbox next to *Movie* table.

## Identifier

This usually corresponds to the table name but sometimes table names might have underscores or other invalid characters, so you decide what to name your entity in generated code (a valid identifier name).

Our table name is *Movie* so it is also a valid and fine C# identifier, so let's leave *Movie* as the entity identifier. Our entity class will be named *MovieRow*.

This name is also used in other class names. For example our page controller will be named *MovieController*.

It also determines the page url, in this sample our editing page will be at URL */MovieDB/Movie*.

## Please Note!

Identifier must always be in Pascal case, e.g. something that starts with a CAPITAL letter.

`myTable` , `mycoolTable` , `aTable` are invalid module names. `MyCoolTable` is OK.

We'll add a validation to Sergen for this soon.

## Setting Module Name

In Serenity terms, a module is a logical group of pages, sharing a common purpose.

For example, in Serene template, all pages related to *Northwind* sample belongs to *Northwind* module.

Pages that are related to general management of site, like users, roles etc. belongs to *Administration* module.

A module usually corresponds to a database schema, or a single database but there is nothing that prevents you from using multiple modules in a single database / schema, or the opposite, multiple databases in one module.

For this tutorial, we will use *MovieDB* (analogous to IMDB) for all pages.

Module name is used in determining namespace and url of generated pages.

For example, our new page will be under *MovieTutorial.MovieDB* namespace and will use */MovieDB* relative url.

## Please Note!

Module names must also be in Pascal case, e.g. something that starts with a CAPITAL letter.

`myModule` , `mycoolmodule` , `aModule` are invalid module names. `MyCoolModule` is fine.

## Permission Key

In Serenity, access control to resources (pages, services etc.) are controlled by permission keys which are simple strings. Users or roles are granted these permissions.

Our Movie page will be only used by administrative users (or maybe later content moderators) so let's leave it as *Administration:General* for now. By default, in Serene template, only the *admin* user has this permission.

## ConnectionKey Parameter

Connection key is set to the connection key of selected connection string in web.config file. You usually don't have to change it, just leave default.

## Generating Code for First Page

After setting parameters as shown in the image above (you only have to set Module Name, others were prefilled), click *Generate Code for Entity* button.

Sergen will generate several files and include them in *MovieTutorial.Web* and *MovieTutorial.Script* projects.

Now you can close Sergen, and return to Visual Studio.

## Serenity Code Generator (ASP.NET Core)

These steps applies only to ASP.NET Core version, not ASP.NET MVC version.

As ASP.NET Core has cross-platform support, .NET Core version of Sergen also needs to run in OSX / Linux / Windows. Thus, its UI is currently console based.

We first need to open a command prompt at project folder. Right click *MovieTutorial.Web* project and click *Open Folder in File Explorer*.

Click *File* menu in file explorer, and click *Open Windows Powershell* or *Open Command Prompt*.

You may also install this extension (<https://marketplace.visualstudio.com/items?itemName=MadsKristensen.OpenCommandLine>) to easily open a command line next time. I can't understand why there is still not such an option in Visual Studio itself.

Make sure you are at `MovieTutorial.Web` directory.

Type `dotnet sergen g` to open Sergen code generation UI (console).

```
Administrator: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
PS C:\Users\Volkan\source\repos\MovieTutorial\MovieTutorial\MovieTutorial.Web> dotnet sergen g
=== Table Code Generation ===

Available Connections:
Default
Northwind

Enter a Connection: ('!' to abort)
_
```

If you receive an error, type `dotnet restore` before running `sergen`.

Sergen will list connections in `appsettings.json` file.

You can use TAB completion, e.g. type `D` and press `TAB` to complete `Default`.

After pressing `Enter` you'll get a list of tables in that database:

```
Administrator: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
PS C:\Users\Volkan\source\repos\MovieTutorial\MovieTutorial\MovieTutorial.Web> dotnet sergen g
=== Table Code Generation ===

Available Connections:
Default
Northwind

Enter a Connection: ('!' to abort)
Default

Available Tables:
dbo.Exceptions
dbo.Languages
dbo.RolePermissions
dbo.Roles
dbo.UserPermissions
dbo.UserPreferences
dbo.UserRoles
dbo.Users
dbo.VersionInfo
mov.Movie

Enter a Table: ('!' to abort)
dbo.
```

Clear `dbo.` using backspace, and type `mov.Movie` or type `m` and use TAB completion to select `mov.Movie`, then press `ENTER`.

Next, Sergen will ask for a module name, enter `MovieDB`.

When prompted, enter `Movie` as identifier.

Leave permission as `Administration:General` and press enter again.

```
Enter a Table: ('!' to abort)
mov.Movie

Enter a Module name for table: ('!' to abort)
MovieDB

Enter a class Identifier for table: ('!' to abort)
Movie

Enter a Permission Key for table: ('!' to abort)
Administration:General
```



Sergen will ask you which files to generate, leave default *RSU* option (e.g. Row, Service and User Interface) and press ENTER last time.

Now you can quit command prompt, and return back to Visual Studio (or Notepad :)

## After Generating Code

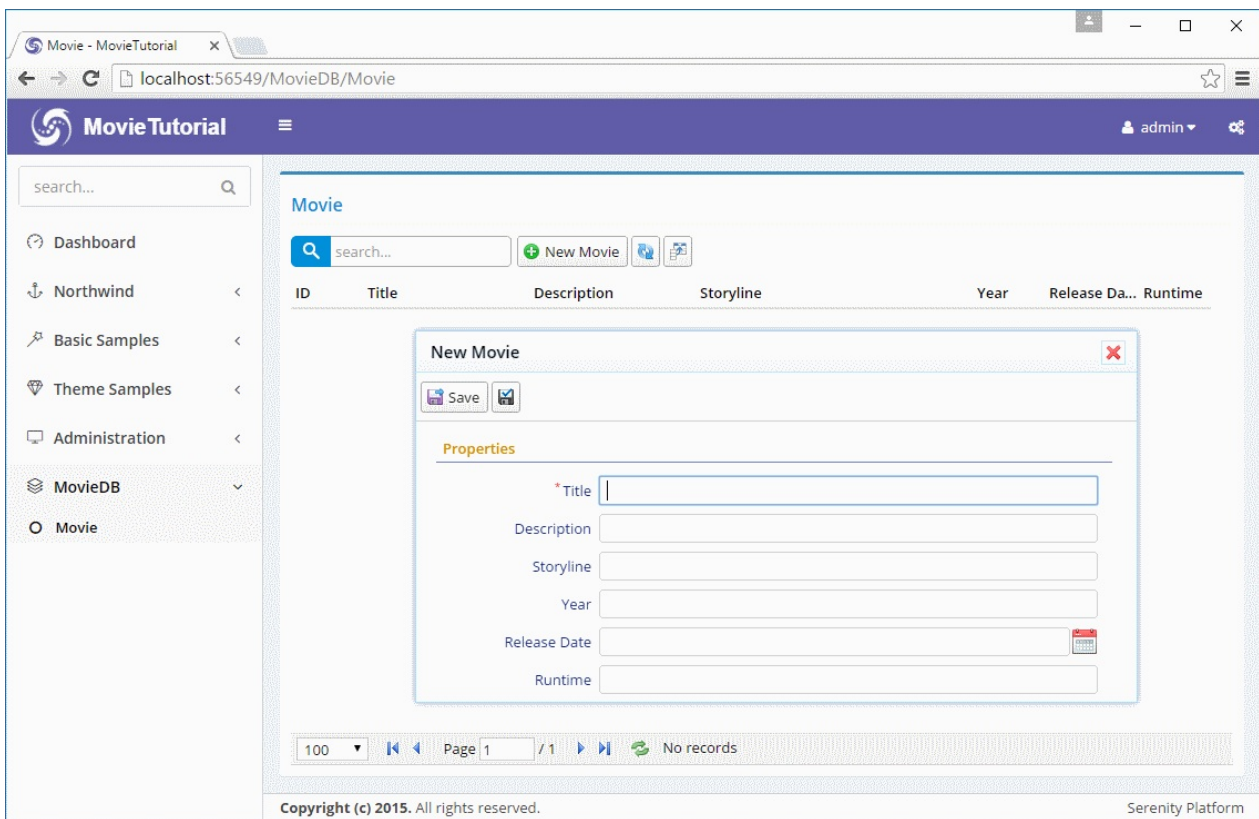
As project is modified, Visual Studio may ask if you want to reload changes, click Reload All.

*REBUILD* the Solution and then press *F5* to launch application.

Use *admin* as username, and *serenity* as password to login.

When you are greeted with Dashboard page, you will notice that there is a new section, *MovieDB* on the bottom of left navigation.

Click to expand it and click *Movie* to open our first generated page.



Now try adding a new movie, than try updating and deleting it.

Sergen generated code for our table, and it just works without writing a single line of code.

This doesn't mean i don't like writing code. In contrast, i love it. Actually i'm not a fan of most designers and code generators. The code they produce is usually unmanagable mess.

Sergen just helped us here for initial setup which is required for layered architecture and platform standards. We would have to create about 10 files for entity, repository, page, endpoint, grid, form etc. Also we needed to do some setup in a few other places.

Even if we did copy paste and replace code from some other page, it would be error prone and take about 5-10 mins.

The code files Sergen generates has minimum code with the absolute basics. This is thanks to the base classes in Serenity that handles the most logic. Once we generate code for some table, we'll probably never use Sergen again (for this table), and modify this generated code to our needs. We'll see how.

# Customizing Movie Interface

## Customizing Field Captions

In our movie grid and form, we have a field named *Runtime*. This field expects an integer number in *minutes*, but in its title there is no sign of this. Let's change its title to Runtime (mins).

There are several ways to do this. Our options include server side form definition, server side columns definition, from script grid code etc. But let's make this change in the central location, the entity itself, so its title changes everywhere.

When Sergen generated code for Movie table, it created an entity class named *MovieRow*. You can find it at *Modules/MovieDB/Movie/MovieRow.cs*.

Here is an excerpt from its source with our Runtime property:

```
namespace MovieTutorial.MovieDB.Entities
{
    // ...
    [ConnectionString("Default"), DisplayName("Movie"),
     InstanceName("Movie"), TwoLevelCached]
    public sealed class MovieRow : Row, IIdRow, INameRow
    {
        // ...
        [DisplayName("Runtime")]
        public Int32? Runtime
        {
            get { return Fields.Runtime[this]; }
            set { Fields.Runtime[this] = value; }
        }
        //...
    }
}
```

We'll talk about entities (or rows) later, let's now focus on our target and change its *DisplayName* attribute value to *\*Runtime (mins)*:

```

namespace MovieTutorial.MovieDB.Entities
{
    // ...
    [ConnectionKey("Default"), DisplayName("Movie"), InstanceName("Movie"),
    TwoLevelCached]
    public sealed class MovieRow : Row, IIdRow, INameRow
    {
        // ...
        [DisplayName("Runtime (mins)")]
        public Int32? Runtime
        {
            get { return Fields.Runtime[this]; }
            set { Fields.Runtime[this] = value; }
        }
        //...
    }
}

```

Now build solution and run application. You'll see that field title is changed in both grid and dialog.

Column title has "..." in it as column is not wide enough, though its hint shows the full title. We'll see how to handle this soon.

The screenshot shows a web application interface for 'MovieTutorial'. On the left is a navigation menu with items like Dashboard, Northwind, Basic Samples, Theme Samples, Administration, MovieDB, and Movie. The main area displays a 'Movie' entity with a search bar and a 'New Movie' button. A 'New Movie' dialog box is open, showing a 'Properties' section with input fields for Title, Description, Storyline, Year, Release Date, and Runtime (mins). Below the dialog is a data grid with columns: ID, Title, Description, Storyline, Year, Release Da..., and Runtime... The 'Runtime (mins)' column title is truncated. The footer contains 'Copyright (c) 2015. All rights reserved.' and 'Serenity Platform'.

## Overriding Column Title and Width

So far so good, what if we wanted to show another title in grid (columns) or dialog (form). We can override it corresponding definition file.

Let's do it on columns first. Next to *MovieRow.cs*, you can find a source file named *MovieColumns.cs*:

```
namespace MovieTutorial.MovieDB.Columns
{
    // ...

    [ColumnsScript("MovieDB.Movie")]
    [BasedOnRow(typeof(Entities.MovieRow))]
    public class MovieColumns
    {
        [EditLink, DisplayName("Db.Shared.RecordId"), AlignRight]
        public Int32 MovieId { get; set; }
        //...
        public Int32 Runtime { get; set; }
    }
}
```

You may notice that this columns definition is based on the *Movie* entity (*BasedOnRow* attribute).

Any attribute written here will override attributes defined in the entity class.

Let's add a *DisplayName* attribute to the *Runtime* property:

```
namespace MovieTutorial.MovieDB.Columns
{
    // ...

    [ColumnsScript("MovieDB.Movie")]
    [BasedOnRow(typeof(Entities.MovieRow))]
    public class MovieColumns
    {
        [EditLink, DisplayName("Db.Shared.RecordId"), AlignRight]
        public Int32 MovieId { get; set; }
        //...
        [DisplayName("Runtime in Minutes"), Width(150), AlignRight]
        public Int32 Runtime { get; set; }
    }
}
```

Now we set column caption to "Runtime in Minutes".

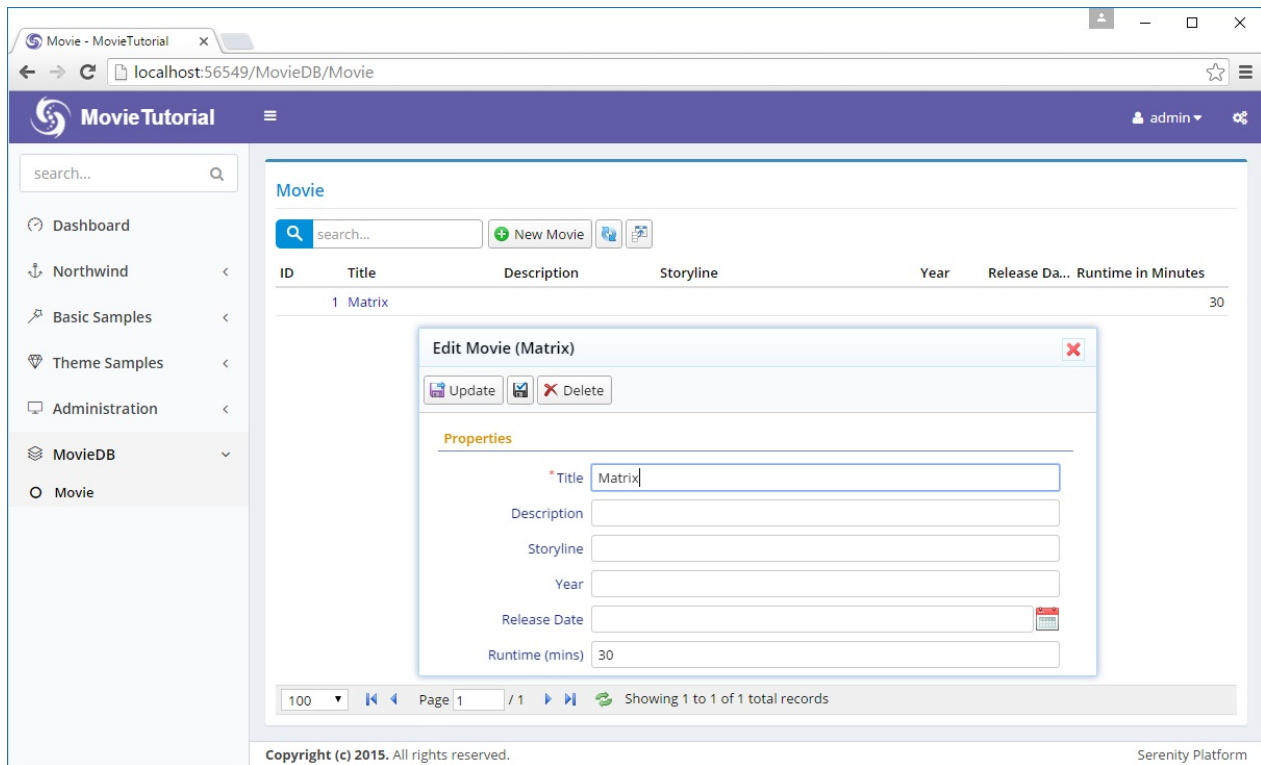
We actually added two more attributes.

One to override column width to 150px.

Serenity applies an automatic width to columns based on field type and character length, unless you set the width explicitly.

And another one to align column to right (AlignCenter, AlignLeft is also available).

Let's build and run again, than we get:



Form field title stayed same, while column title changed.

If we wanted to override form field title instead, we would do similar steps in `MovieForm.cs`

## Changing Editor Type For Description and Storyline

Description and Storyline fields can be a bit longer compared to Title field, so lets change their editor types to a textarea.

Open `MovieForm.cs` in the same folder with `MovieColumns.cs` and `MovieRow.cs`.

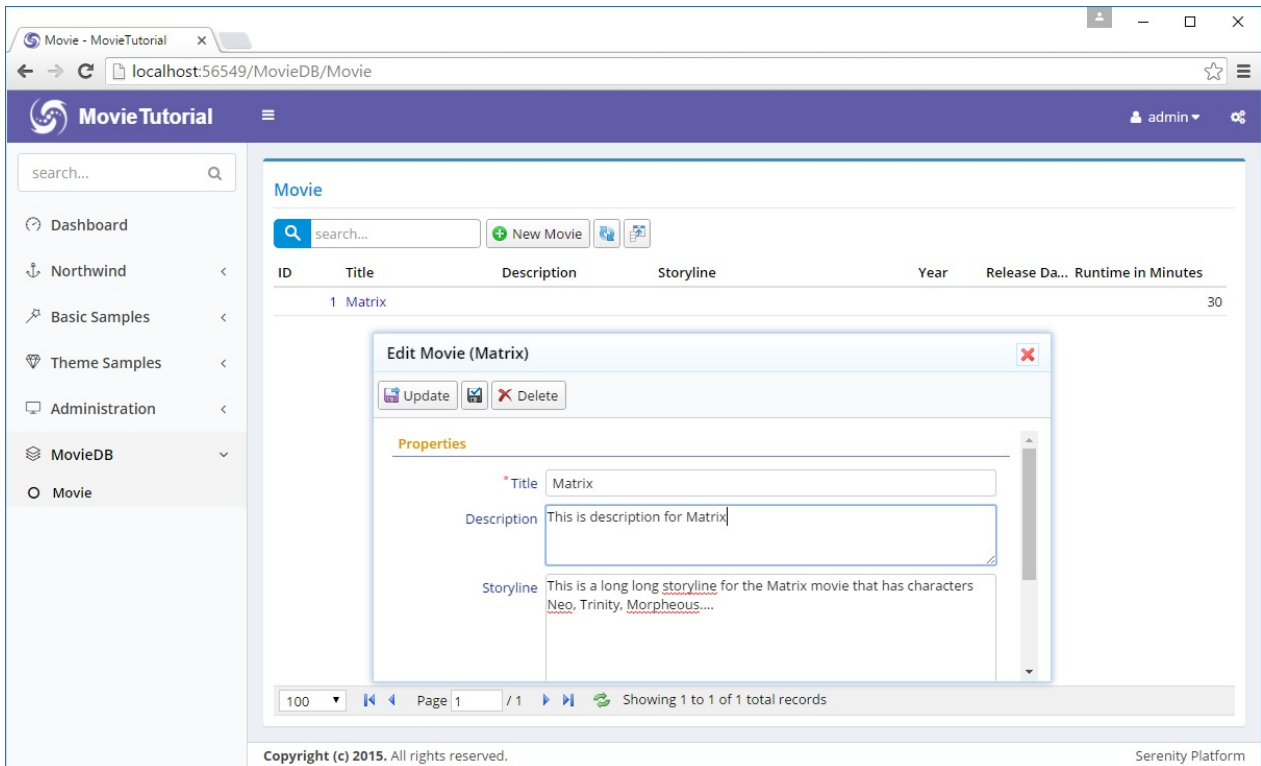
```
namespace MovieTutorial.MovieDB.Forms
{
    //...
    [FormScript("MovieDB.Movie")]
    [BasedOnRow(typeof(Entities.MovieRow))]
    public class MovieForm
    {
        public String Title { get; set; }
        public String Description { get; set; }
        public String Storyline { get; set; }
        public Int32 Year { get; set; }
        public DateTime ReleaseDate { get; set; }
        public Int32 Runtime { get; set; }
    }
}
```

and add `TextAreaEditor` attributes to both:

```
namespace MovieTutorial.MovieDB.Forms
{
    //...
    [FormScript("MovieDB.Movie")]
    [BasedOnRow(typeof(Entities.MovieRow))]
    public class MovieForm
    {
        public String Title { get; set; }
        [TextAreaEditor(Rows = 3)]
        public String Description { get; set; }
        [TextAreaEditor(Rows = 8)]
        public String Storyline { get; set; }
        public Int32 Year { get; set; }
        public DateTime ReleaseDate { get; set; }
        public Int32 Runtime { get; set; }
    }
}
```

I left more editing rows for Storyline (8) compared to Description (3) as Storyline should be much longer.

After rebuild and run, we have this:



Serene has several editor types to choose from. Some are automatically picked based on field data type, while you need to explicitly set others.

You can also develop your own editor types. You can take existing editor types as base classes, or develop your own from scratch. We'll see how in following chapters.

As editors became a bit higher, form height exceeded the default Serenity form height (which is about 260px) and now we have a vertical scroll bar. Let's remove it.

## Setting Initial Dialog Size With CSS (Less)

Sergen generated some CSS for our movie dialog in *MovieTutorial.Web/Content/site/site.less* file.

If you open it, and scroll to bottom, you will see this:

```

/* ----- */
/* APPENDED BY CODE GENERATOR, MOVE TO CORRECT PLACE AND REMOVE THIS COMMENT */
/* ----- */

.s-MovieDB-MovieDialog {
  > .size { width: 650px; }
  .caption { width: 150px; }
}
    
```



You can safely remove the 3 comment lines (appended by code generator...). This is just a reminder for you to move them to a better place like a *site.movies.less* file specific to this module (recommended).

These rules are applied to elements with *.s-MovieDB-MovieDialog* class. Our Movie dialog has this class by default, which is generated by "s-" + ModuleName + "-" + ClassName.

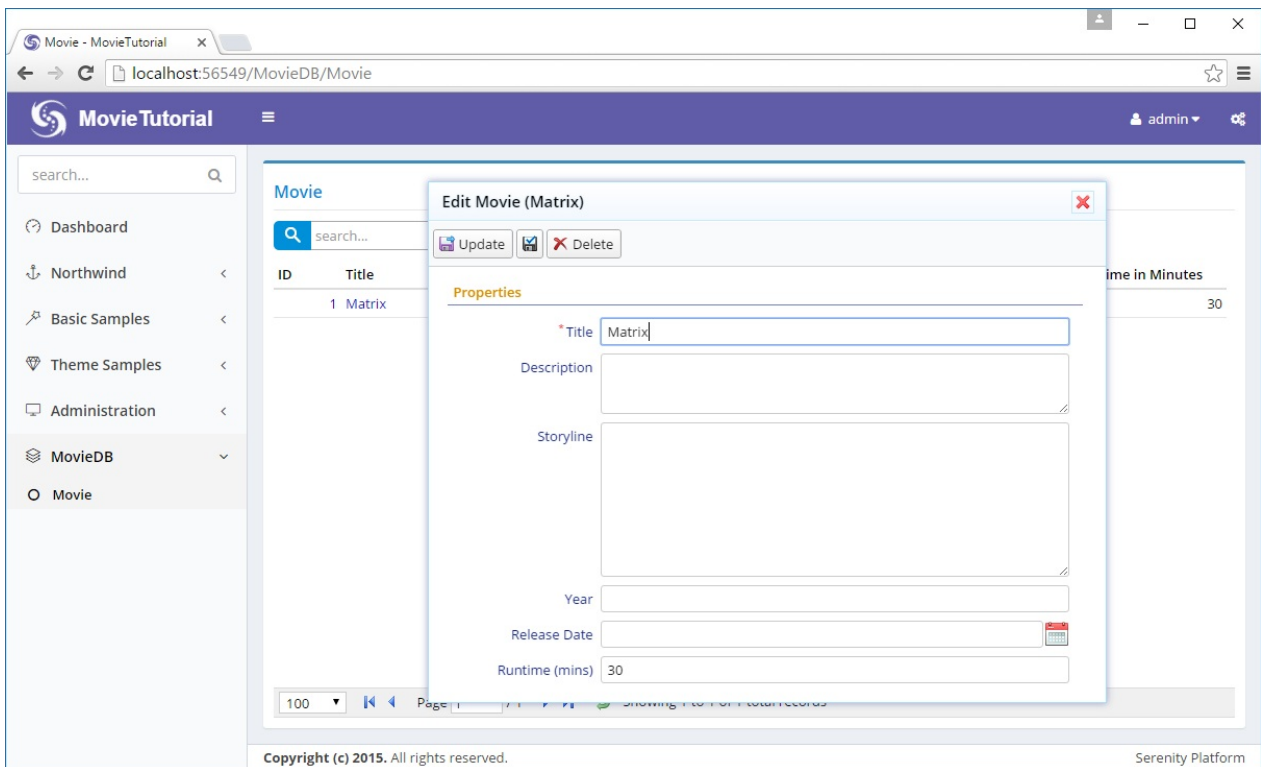
In the second line it is specified that this dialog is 650px wide by default.

In third line, we specify that field labels should be 150px (@l: 150px).

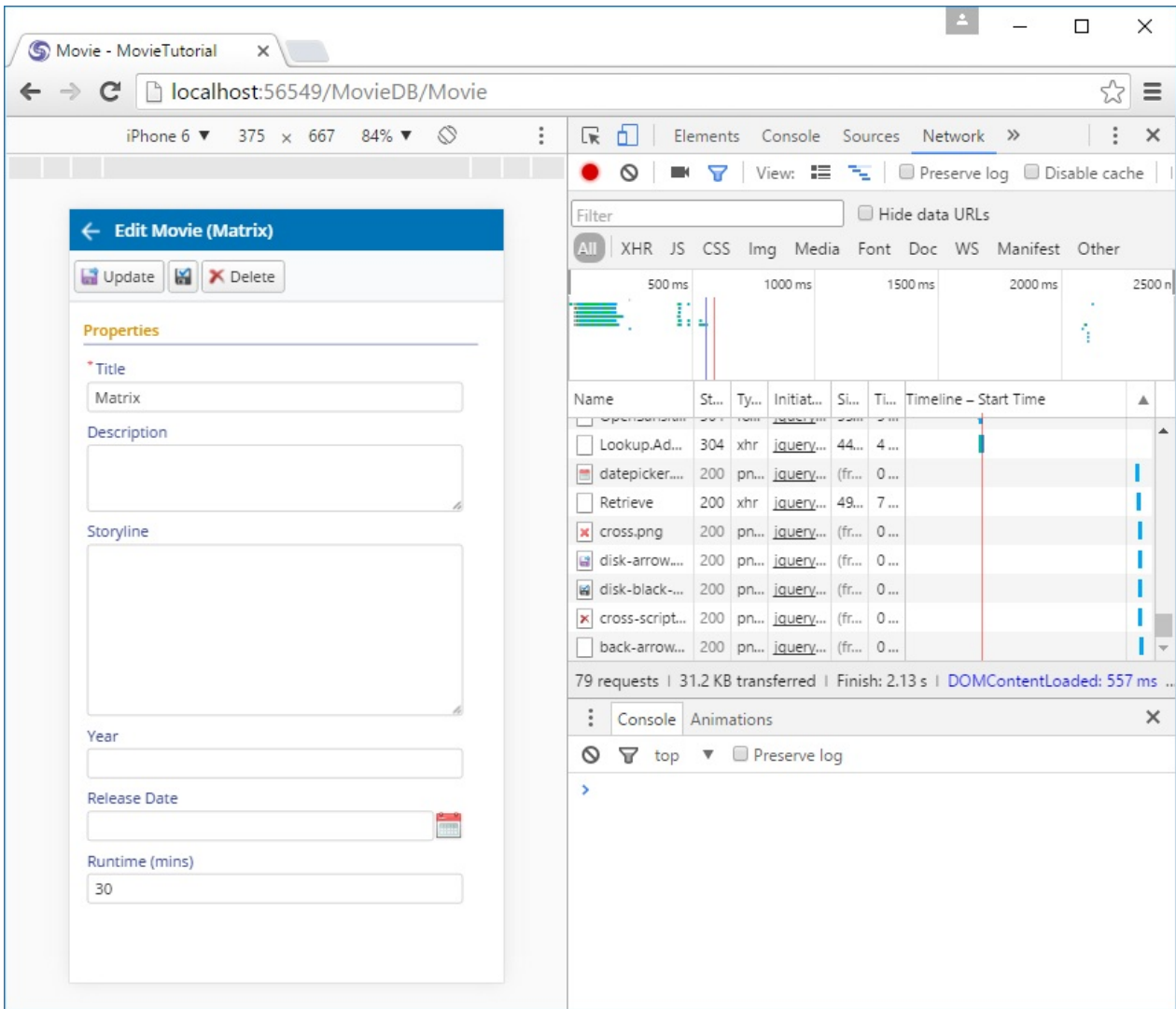
Let's change our initial dialog height to 500px (in desktop mode), so it won't require a vertical scroll bar:

```
.s-MovieDialog {
  > .size { width: 650px; height: 500px; }
  .caption { width: 150px; }
}
```

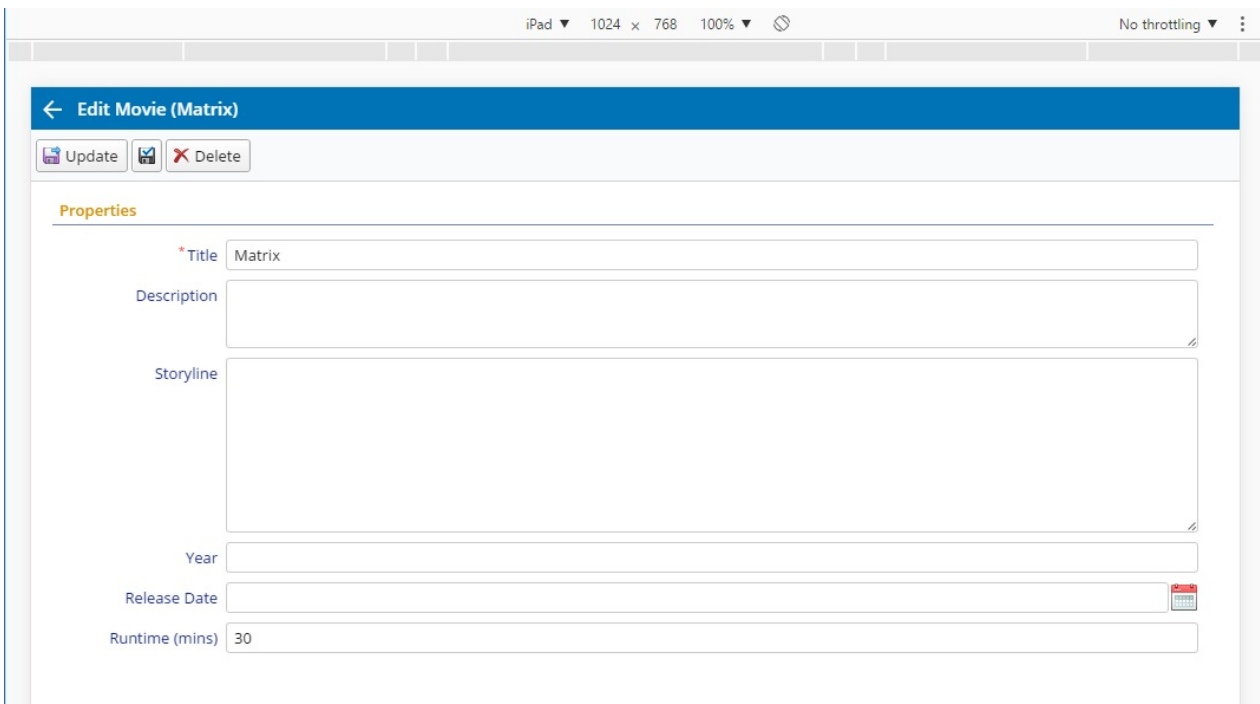
For this change to be applied to your dialog, you need to build solution. As this "site.less" file is compiled to a "site.css" file on build. Now build and refresh the page.



What i mean by *desktop mode* above will become clearer soon. Serenity dialogs are responsive by default. Let's resize our browser window to a width about 350px. I'll use mobile mode of my Chrome browser to switch to iPhone 6:



And now an iPad in landscape mode:



So, the height we set here is only meaningful for desktop mode. Dialog will turn into a responsive, device size specific mode in mobile, without having to mess with CSS @media queries.

## Changing Page Title

Our page has title of *Movie*. Let's change it to *Movies*.

Open *MovieRow.cs* again.

```
namespace MovieTutorial.MovieDB.Entities
{
    // ...
    [ConnectionString("Default"), DisplayName("Movie"), InstanceName("Movie"),
    TwoLevelCached]
    public sealed class MovieRow : Row, IIdRow, INameRow
    {
        [DisplayName("Movie Id"), Identity]
        public Int32? MovieId
    }
}
```

Change `DisplayName` attribute value to *Movies*. This is the name that is used when this table is referenced, and it is usually a plural name. This attribute is used for determining default page title.

It is also possible to override the page title in *MoviePage.Index.cshtml* file but as before, we prefer to do it from a central location so that this information can be reused in other places.

`InstanceName` corresponds to singular name and is used in New Record (New Movie) button of the grid and also determines the dialog title (e.g. Edit Movie).

```
namespace MovieTutorial.MovieDB.Entities
{
    // ...
    [ConnectionString("Default"), DisplayName("Movies"), InstanceName("Movie"),
    TwoLevelCached]
    public sealed class MovieRow : Row, IIdRow, INameRow
    {
        [DisplayName("Movie Id"), Identity]
        public Int32? MovieId
    }
}
```

# Handling Movie Navigation

## Setting Navigation Item Title and Icon

When Sergen generated code for Movie table, it also created a navigation item entry. In Serene, navigation items are created with special assembly attributes.

Open *MoviePage.cs* in the same folder, on top of it you'll find this line:

```
[assembly:Serenity.Navigation.NavigationLink(int.MaxValue, "MovieDB/Movie",
    typeof(MovieTutorial.MovieDB.Pages.MovieController))]

namespace MovieTutorial.MovieDB.Pages
{
    //...
```

First argument to this attribute is display order for this navigation item. As we only have one navigation item in Movie category yet, we don't have to mess with ordering yet.

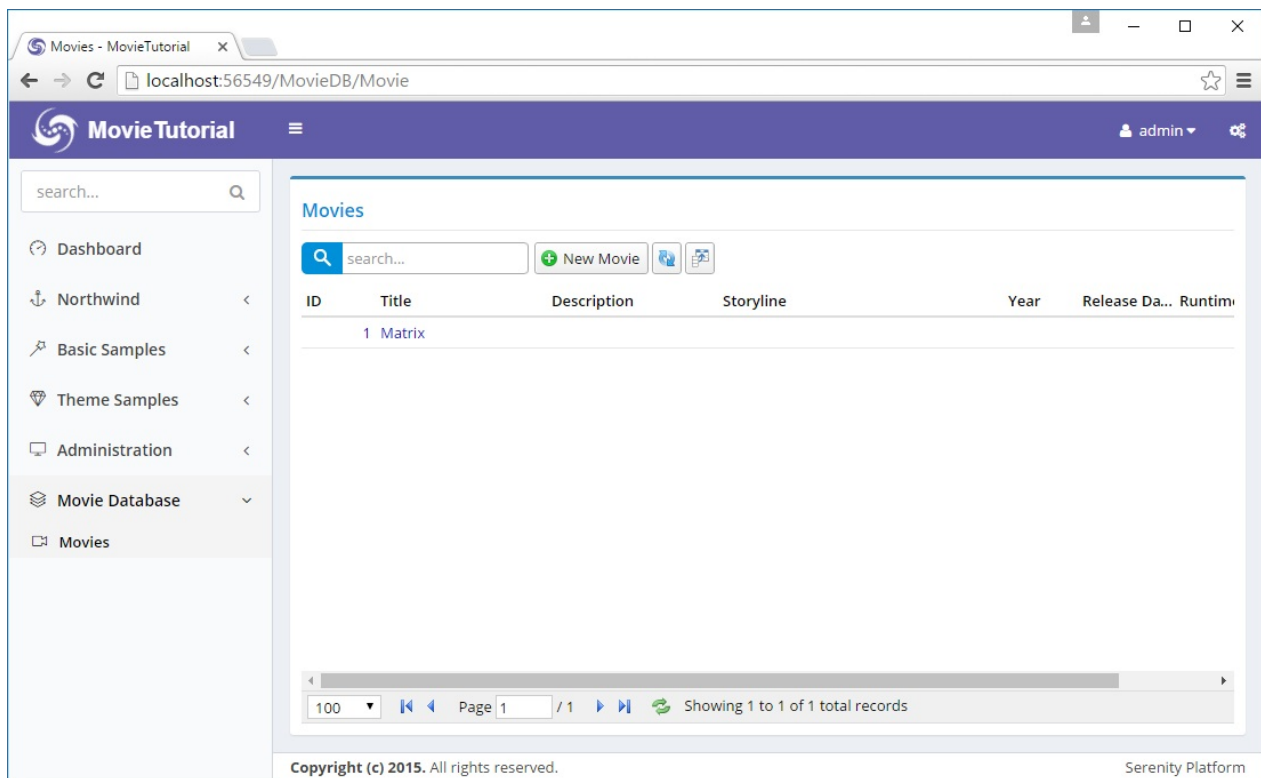
Second parameter is navigation title in "Section Title/Link Title" format. Section and navigation items are separated with a slash (/).

Lets change it to *Movie Database/Movies*.

```
[assembly:Serenity.Navigation.NavigationLink(int.MaxValue, "Movie Database/Movies",
    typeof(MovieTutorial.MovieDB.Pages.MovieController), icon: "icon-camrecorder")]

namespace MovieTutorial.MovieDB.Pages
{

    //...
```



We also changed navigation item icon to *icon-camcorder*. Serene template has two sets of font icons, Simple Line Icons and Font Awesome. Here we used a glyph from simple line icons set.

To see list of simple line icons and their css classes, visit link below:

<http://thesabbir.github.io/simple-line-icons/>

FontAwesome is available here:

<https://fontawesome.github.io/Font-Awesome/icons/>

There is also a page in Serene under *Theme Samples / UI Elements / Icons* containing a list of these icon sets.

## Ordering Navigation Sections

As our *Movie Database* section is auto generated last, it is displayed at the bottom of navigation menu.

We'll move it before Northwind menu.

As we saw recently, Sergen created a navigation item in *MoviePage.cs*. If navigation items are scattered through pages like this, it would be hard to see the big picture (list of all navigation items) and order them easily.

So we move it to our central location which is at *MovieTutorial.Web/Modules/Common/Navigation/NavigationItems.cs*.

Just cut the below lines from *MoviePage.cs*:

```
[assembly:Serenity.Navigation.NavigationLink(int.MaxValue, "Movie Database/Movies",  
    typeof(MovieTutorial.MovieDB.Pages.MovieController), icon: "icon-camrecorder")]
```

Move it into *NavigationItems.cs* and modify it like this:

```
using Serenity.Navigation;  
using Northwind = MovieTutorial.Northwind.Pages;  
using Administration = MovieTutorial.Administration.Pages;  
using MovieDB = MovieTutorial.MovieDB.Pages;  
  
[assembly: NavigationLink(1000, "Dashboard", url: "~/", permission: "",  
    icon: "icon-speedometer")]  
  
[assembly: NavigationMenu(2000, "Movie Database", icon: "icon-film")]  
[assembly: NavigationLink(2100, "Movie Database/Movies",  
    typeof(MovieDB.MovieController), icon: "icon-camrecorder")]  
  
[assembly: NavigationMenu(8000, "Northwind", icon: "icon-anchor")]  
[assembly: NavigationLink(8200, "Northwind/Customers",  
    typeof(Northwind.CustomerController), icon: "icon-wallet")]  
[assembly: NavigationLink(8300, "Northwind/Products",  
    typeof(Northwind.ProductController), icon: "icon-present")]  
// ...
```

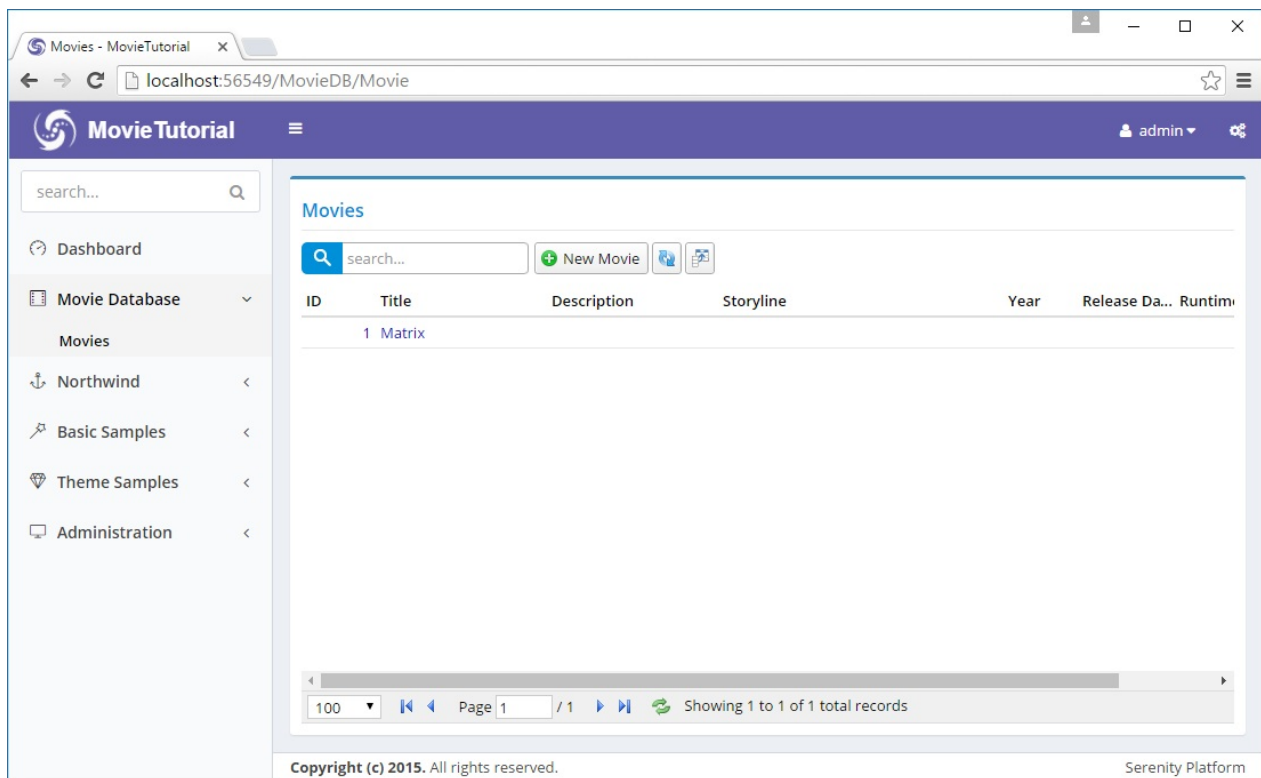
Here we also declared a navigation menu (Movie Database) with *film* icon. When you don't have an explicitly defined navigation menu, Serenity implicitly creates one, but in this case you can't order menu yourself, or set menu icon.

We assigned it a display order of *2000* so this menu will display just after Dashboard link (1000) but before Northwind menu (8000).

We assigned our *Movies* link a display order value of *2100* but it doesn't matter right now, as we have only one navigation item under *Movie Database* menu yet.

First level links and navigation menus are sorted according to their display order first, then second level links among their siblings.

Here is how it looks like after these changes:



## Troubleshooting Some Issues with Visual Studio

In case you didn't notice already, Visual Studio doesn't let you modify code while your site is running. Also your site stops when you stop debugging, so you can't keep browser window open and refresh after rebuilding.

To solve this issue, we need to disable *Edit And Continue* (have no idea why).

Right Click *MovieTutorial.Web* project, click *Properties*, in the Web tab, uncheck *Enable Edit And Continue* under *Debuggers*.

Unfortunately, the solution above stops works in Visual Studio 2015 Update 2. Only workaround so far seems like starting without debugging, e.g. Ctrl+F5 instead of F5.

Solution above only applies to ASP.NET MVC version, not ASP.NET CORE version.

Also, on your site, top blue progress bar (which is a Pace.js animation), keeps running all the time like it is still loading something. It is thanks to the *Browser Link* feature of Visual Studio. To disable it, locate its button in Visual Studio toolbar that looks like a refresh button (next to play icon with browser name like Chrome), click dropdown and uncheck *Enable Browser Link*.

It's also possible to disable it with a web.config setting

```
<appsettings>  
  <add key="vs:EnableBrowserLink" value="false" />  
</appsettings>
```

Serene 1.5.4 and later has this in web.config by default, so you might not experience this issue

I'm not sure if there is a corresponding setting in appsettings.json file of ASP.NET Core version



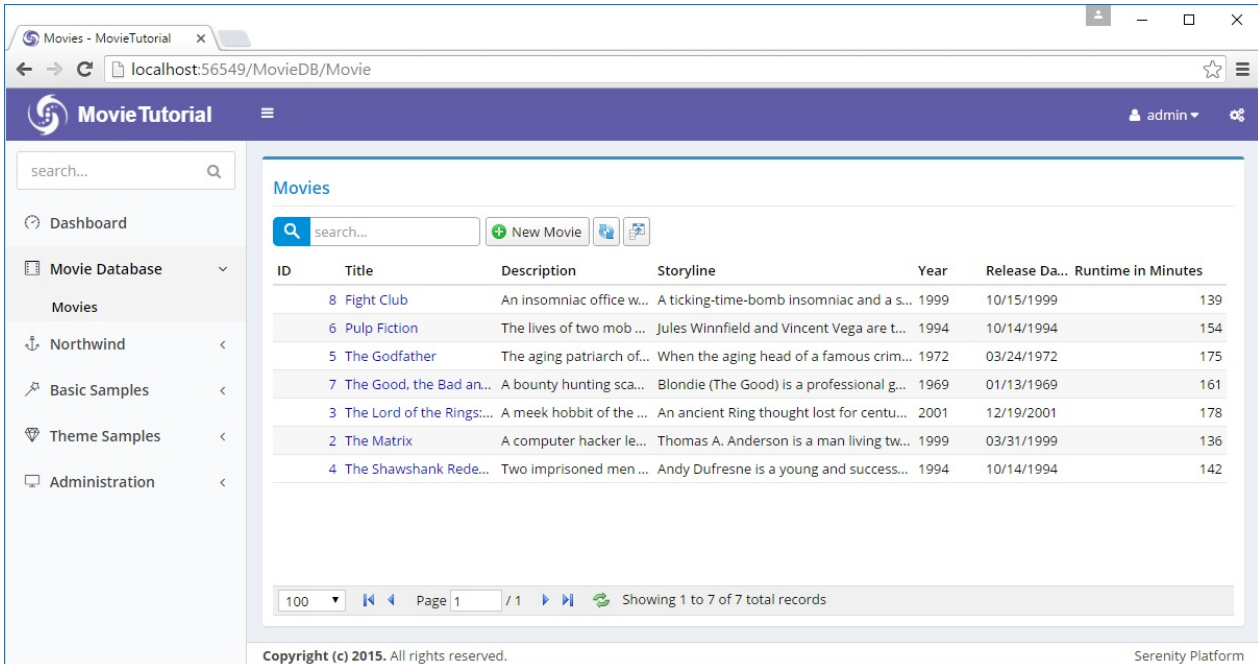
# Customizing Quick Search

## Adding Several Movie Entries

For the following sections, we need some sample data. We can copy and paste some from IMDB.

If you don't want to waste your time entering this sample data, it is available as a migration at the link below:

[https://github.com/volkanceylan/MovieTutorial/blob/master/MovieTutorial/MovieTutorial.Web/Modules/Common/Migrations/DefaultDB/DefaultDB\\_20160519\\_135200\\_SampleMovies.cs](https://github.com/volkanceylan/MovieTutorial/blob/master/MovieTutorial/MovieTutorial.Web/Modules/Common/Migrations/DefaultDB/DefaultDB_20160519_135200_SampleMovies.cs)



The screenshot shows a web application interface for 'MovieTutorial'. The main content area displays a table of movies. The table has the following columns: ID, Title, Description, Storyline, Year, Release Date, and Runtime in Minutes. The data rows are as follows:

ID	Title	Description	Storyline	Year	Release Date	Runtime in Minutes
8	Fight Club	An insomniac office w...	A ticking-time-bomb insomniac and a s...	1999	10/15/1999	139
6	Pulp Fiction	The lives of two mob ...	Jules Winnfield and Vincent Vega are t...	1994	10/14/1994	154
5	The Godfather	The aging patriarch of...	When the aging head of a famous crim...	1972	03/24/1972	175
7	The Good, the Bad and the Ugly	A bounty hunting sca...	Blondie (The Good) is a professional g...	1969	01/13/1969	161
3	The Lord of the Rings: The Fellowship Ring	A meek hobbit of the ...	An ancient Ring thought lost for centu...	2001	12/19/2001	178
2	The Matrix	A computer hacker le...	Thomas A. Anderson is a man living tw...	1999	03/31/1999	136
4	The Shawshank Redemption	Two imprisoned men ...	Andy Dufresne is a young and success...	1994	10/14/1994	142

At the bottom of the table, there is a pagination control showing 'Page 1 / 1' and 'Showing 1 to 7 of 7 total records'. The footer of the application includes 'Copyright (c) 2015. All rights reserved.' and 'Serenity Platform'.

If we typed *go* into search box, we would see two movies are filtered: *The Good, the Bad and the Ugly* and *The Godfather*.

If we typed *Gandalf* we wouldn't be able to find anything.

By default, Sergen determines first textual field of a table as *the name field*. In movies table it is *Title*. This field has a *QuickSearch* attribute on it that specifies that text searches should be performed on it.

The name field also determines initial sorting order and shown in edit dialog titles.

Sometimes, first textual column might not be the name field. If you wanted to change it to another field, you would do it in *MovieRow.cs*:

```
namespace MovieTutorial.MovieDB.Entities
{
    //...
    public sealed class MovieRow : Row, IIdRow, INameRow
    {
        //...
        StringField INameRow.NameField
        {
            get { return Fields.Title; }
        }
    }
}
```

Code generator determined that first textual (string) field in our table is Title. So it added a `INameRow` interface to our `Movies` row and implemented it by returning Title field. If wanted to use `Description` as name field, we would just replace it.

Here, *Title* is actually the name field, so we leave it as is. But we want Serenity to search also in *Description* and *Storyline* fields. To do this, you need to add *QuickSearch* attribute to these fields too, as shown below:

```

namespace MovieTutorial.MovieDB.Entities
{
    //...
    public sealed class MovieRow : Row, IIdRow, INameRow
    {
        //...
        [DisplayName("Title"), Size(200), NotNull, QuickSearch]
        public String Title
        {
            get { return Fields.Title[this]; }
            set { Fields.Title[this] = value; }
        }

        [DisplayName("Description"), Size(1000), QuickSearch]
        public String Description
        {
            get { return Fields.Description[this]; }
            set { Fields.Description[this] = value; }
        }

        [DisplayName("Storyline"), QuickSearch]
        public String Storyline
        {
            get { return Fields.Storyline[this]; }
            set { Fields.Storyline[this] = value; }
        }
        //...
    }
}

```

Now, if we search for *Gandalf*, we'll get a *The Lord of the Rings* entry:

The screenshot shows a web browser window with the URL `localhost:56549/MovieDB/Movie`. The application interface includes a search bar with the text "gandalf" entered. Below the search bar, a table displays the search results. The table has columns for ID, Title, Description, Storyline, Year, Release Date, and Runtime in Minutes. One result is shown: "3 The Lord of the Rings... A meek hobbit of the ... An ancient Ring thought lost for centu... 2001 12/19/2001 178". The footer of the application shows "Copyright (c) 2015. All rights reserved." and "Serenity Platform".

ID	Title	Description	Storyline	Year	Release Da...	Runtime in Minutes
3	The Lord of the Rings...	A meek hobbit of the ...	An ancient Ring thought lost for centu...	2001	12/19/2001	178

QuickSearch attribute, by default, searches with *contains* filter. It has some options to make it match by *starts with* filter or match only exact values.

If we wanted it to show only rows that *starts with* typed text, we would change attribute to:

```
[DisplayName("Title"), Size(200), NotNull, QuickSearch(SearchType.StartsWith)]
public String Title
{
    get { return Fields.Title[this]; }
    set { Fields.Title[this] = value; }
}
```

Here this quick search feature is not very useful, but for values like SSN, serial number, identification number, phone number etc, it might be.

If we wanted to search also in year column, but only exact integer values (1999 matches but not 19):

```
[DisplayName("Year"), QuickSearch(SearchType.Equals, numericOnly: 1)]
public Int32? Year
{
    get { return Fields.Year[this]; }
    set { Fields.Year[this] = value; }
}
```

You might have noticed that we are not writing any C# or SQL code for these basic features to work. We just specify what we want, rather than how to do it. This is what declarative programming is.

It is also possible to provide user with ability to determine which field she wants to search on.

Open *MovieTutorial.Web/Modules/MovieDB/Movie/MovieGrid.ts* and modify it like:

```

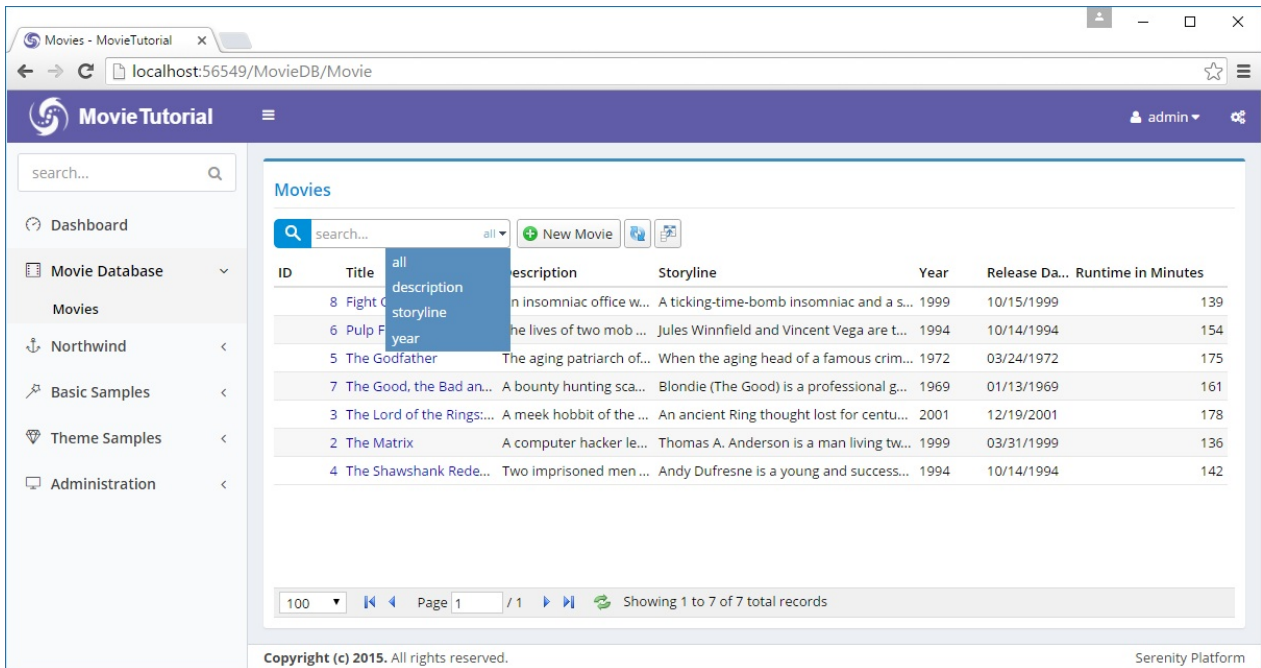
namespace MovieTutorial.MovieDB {

    @Serenity.Decorators.registerClass()
    export class MovieGrid extends
        Serenity.EntityGrid<MovieRow, any> {
        //...
        constructor(container: JQuery) {
            super(container);
        }

        protected getQuickSearchFields():
            Serenity.QuickSearchField[] {
            return [
                { name: "", title: "all" },
                { name: "Description", title: "description" },
                { name: "Storyline", title: "storyline" },
                { name: "Year", title: "year" }
            ];
        }
    }
}

```

Once you save that file, we'll have a dropdown in quick search input:



Unlike prior samples where we modified Server side code, this time we did changes in client side, and modified Javascript (TypeScript) code.

## Running T4 Templates (.tt files, ASP.NET MVC Version)

In prior sample we hardcoded field names like *Description*, *Storyline* etc. This may lead to typing errors if we forgot actual property names or their casing at server side (javascript is case sensitive).

Serene contains some T4 (.tt) files to transfer such information from server side (rows etc in C#) to client side (TypeScript) for intellisense purposes.

Before running these templates, please make sure that your solution builds successfully as templates uses your output DLL file (*MovieTutorial.Web.dll*) to generate code.

After building your solution, click on *Build* menu, than *Transform All Templates*.

## Running T4 Templates (ASP.NET Core Version)

You don't have to transform templates in ASP.NET Core version. Serene does it automatically on build.

Actually, there isn't any T4 file in ASP.NET Core version.

So from now on, when we say transform templates, just build your project (if you use ASP.NET Core version).

It is also possible to open a command prompt in project directory and type `dotnet sergen t` to transform templates manually.

## Intellisense on TypeScript Code (After transforming templates)

We can now use intellisense to replace hardcoded field names with compile time checked versions:

```

namespace MovieTutorial.MovieDB
{
    //...
    public class MovieGrid extends EntityGrid<MovieRow, any>
    {
        constructor(container: JQuery) {
            super(container);
        }

        protected getQuickSearchFields(): Serenity.QuickSearchField[]
        {
            let fld = MovieRow.Fields;
            return [
                { name: "", title: "all" },
                { name: fld.Description, title: "description" },
                { name: fld.Storyline, title: "storyline" },
                { name: fld.Year, title: "year" }
            ];
        }
    }
    ///...
}

```

What about field titles? It is not so critical as field names, but can be useful for localization purposes (if we later decide to translate it):

```

namespace MovieTutorial.MovieDB
{
    //...
    public class MovieGrid extends EntityGrid<MovieRow, any>
    {
        constructor(container: JQuery) {
            super(container);
        }

        protected getQuickSearchFields(): Serenity.QuickSearchField[] {
            let fld = MovieRow.Fields;
            let txt = (s) => Q.text("Db." +
                MovieRow.localTextPrefix + "." + s).toLowerCase();
            return [
                { name: "", title: "all" },
                { name: fld.Description, title: txt(fld.Description) },
                { name: fld.Storyline, title: txt(fld.Storyline) },
                { name: fld.Year, title: txt(fld.Year) }
            ];
        }
    }
    ///...
}

```

We made use of the local text dictionary (translations) available at client side. It's something like this:

```
{
  // ...
  "Db.MovieDB.Movie.Description": "Description",
  "Db.MovieDB.Movie.Storyline": "Storyline",
  "Db.MovieDB.Movie.Year": "Year"
  // ...
}
```

Local text keys for row fields are generated from *"Db." + (LocalTextPrefix for Row) + "." + FieldName*.

Their values are generated from [DisplayName] attributes on your fields by but might be something else in another culture if they are translated.

LocalTextPrefix corresponds to *ModuleName + "." + RowClassName* by default, but can be changed in Row fields constructor.



## Adding a Movie Kind Field

If we wanted to also keep TV series and mini series in our movie table, we would need another field to store it: *MovieKind*.

As we didn't add it while creating the Movie table, now we'll write another migration to add it to our database.

Don't modify existing migrations, they won't run again.

Create another migration file under *Modules/Common/Migrations/DefaultDB/DefaultDB\_20160519\_145500\_MovieKind.cs*:

```
using FluentMigrator;

namespace MovieTutorial.Migrations.DefaultDB
{
    [Migration(20160519145500)]
    public class DefaultDB_20160519_145500_MovieKind : Migration
    {
        public override void Up()
        {
            Alter.Table("Movie").InSchema("mov")
                .AddColumn("Kind").AsInt32().NotNullable()
                .WithDefaultValue(1);
        }

        public override void Down()
        {
        }
    }
}
```

## Declaring a MovieKind Enumeration

Now as we added *Kind* column to *Movie* table, we need a set of movie kind values. Let's define it as an enumeration at *MovieTutorial.Web/Modules/MovieDB/Movie/MovieKind.cs*:

```
using Serenity.ComponentModel;
using System.ComponentModel;

namespace MovieTutorial.MovieDB
{
    [EnumKey("MovieDB.MovieKind")]
    public enum MovieKind
    {
        [Description("Film")]
        Film = 1,
        [Description("TV Series")]
        TvSeries = 2,
        [Description("Mini Series")]
        MiniSeries = 3
    }
}
```

## Adding Kind Field to MovieRow Entity

As we are not using Sergen anymore, we need to add a mapping in our MovieRow.cs for *Kind* column manually. Add following property declaration in MovieRow.cs after *Runtime* property:

```
[DisplayName("Runtime (mins)")]
public Int32? Runtime
{
    get { return Fields.Runtime[this]; }
    set { Fields.Runtime[this] = value; }
}

[DisplayName("Kind"), NotNull]
public MovieKind? Kind
{
    get { return (MovieKind?)Fields.Kind[this]; }
    set { Fields.Kind[this] = (Int32?)value; }
}
```

We also need to declare a Int32Field object which is required for Serenity entity system. On the bottom of MovieRow.cs locate *RowFields* class and modify it to add *Kind* field after the *Runtime* field:

```
public class RowFields : RowFieldsBase
{
    // ...
    public readonly Int32Field Runtime;
    public readonly Int32Field Kind;

    public RowFields()
        : base("[mov].Movie")
    {
        LocalTextPrefix = "MovieDB.Movie";
    }
}
```

## Adding Kind Selection To Our Movie Form

If we build and run our project now, we'll see that there is no change in the Movie form, even if we added *Kind* field mapping to the *MovieRow*. This is because, fields shown/edited in the form are controlled by declarations in *MovieForm.cs*.

Modify *MovieForm.cs* as below:

```
namespace MovieTutorial.MovieDB.Forms
{
    // ...
    [FormScript("MovieDB.Movie")]
    [BasedOnRow(typeof(Entities.MovieRow))]
    public class MovieForm
    {
        // ...
        public MovieKind Kind { get; set; }
        public Int32 Runtime { get; set; }
    }
}
```

Now, build your solution and run it. When you try to edit a movie or add a new one, nothing will happen. This is an expected situation. If you check developer tools console of your browser (F12, inspect element etc.) you'll see such an error:

You might not have this error with ASP.NET Core version as it auto transforms T4

```
Uncaught Can't find MovieTutorial.MovieDB.MovieKind enum type!
```

## Please Note!

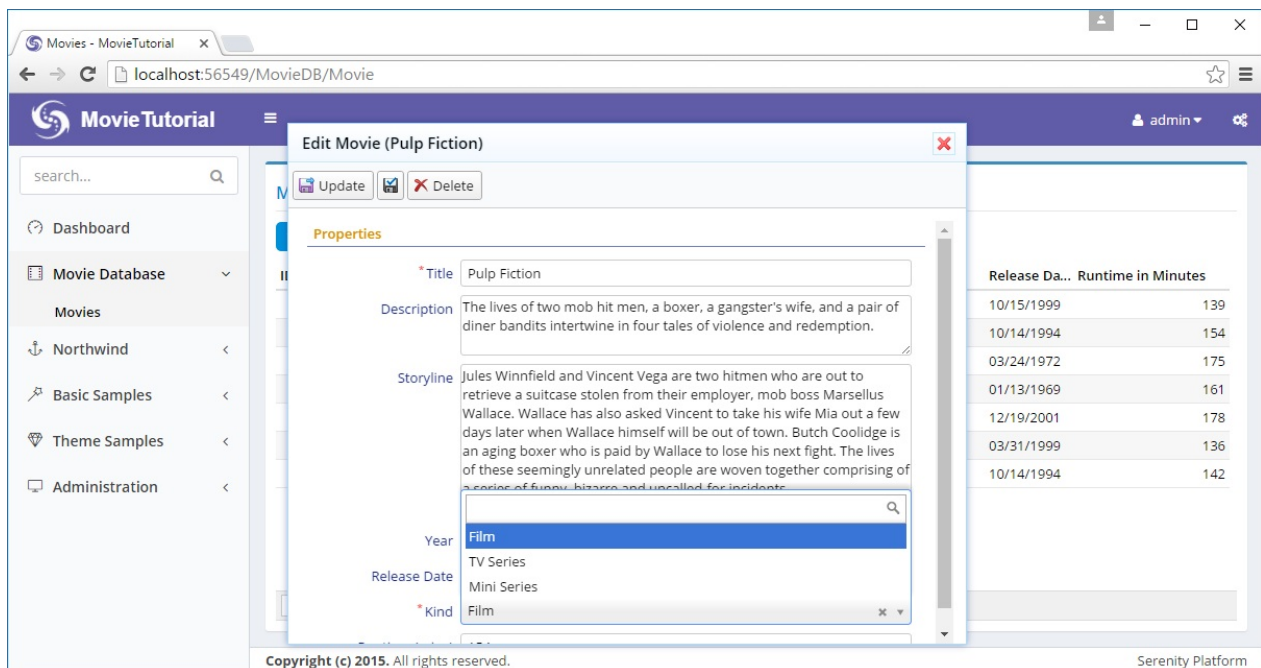
Whenever such a thing happens, e.g. some button not working, you got an empty page, grid etc, please first check browser console for errors, before reporting it.

## Why We Had This Error?

This error is caused by MoveKind enumeration not available client side. We should run our T4 templates before executing our program.

Now in Visual Studio, click *Build -> Transform All Templates* again.

Rebuild your solution and execute it. Now we have a nice dropdown in our form to select movie kind.



Just build project for ASP.NET Core version, as there is no T4 template

## Declaring a Default Value for Movie Kind

As *Kind* is a required field, we need to fill it in *Add Movie* dialog, otherwise we'll get a validation error.

But most movies we'll store are feature films, so its default should be this value.

To add a default value for *Kind* property, add a *DefaultValue* attribute like this:

```
[DisplayName("Kind"), NotNull, DefaultValue(MovieKind.Film)]
public MovieKind? Kind
{
    get { return (MovieKind?)Fields.Kind[this]; }
    set { Fields.Kind[this] = (Int32?)value; }
}
```

Now, in *Add Movie* dialog, *Kind* field will come prefilled as *Film*.

# Adding Movie Genres

## Adding Genre Field

To hold Movie genres we need a lookup table. For *Kind* field we used an enumeration but this time genres might not be that *static* to declare them as an enumeration.

As usual, we start with a migration.

*Modules/Common/Migrations/DefaultDB/DefaultDB\_20160519\_154700\_GenreTable.cs:*

```
using FluentMigrator;
using System;

namespace MovieTutorial.Migrations.DefaultDB
{
    [Migration(20160519154700)]
    public class DefaultDB_20160519_154700_GenreTable : Migration
    {
        public override void Up()
        {
            Create.Table("Genre").InSchema("mov")
                .WithColumn("GenreId").AsInt32().NotNullable()
                .PrimaryKey().Identity()
                .WithColumn("Name").AsString(100).NotNullable();

            Alter.Table("Movie").InSchema("mov")
                .AddColumn("GenreId").AsInt32().Nullable()
                .ForeignKey("FK_Movie_GenreId", "mov", "Genre", "GenreId");
        }

        public override void Down()
        {
        }
    }
}
```

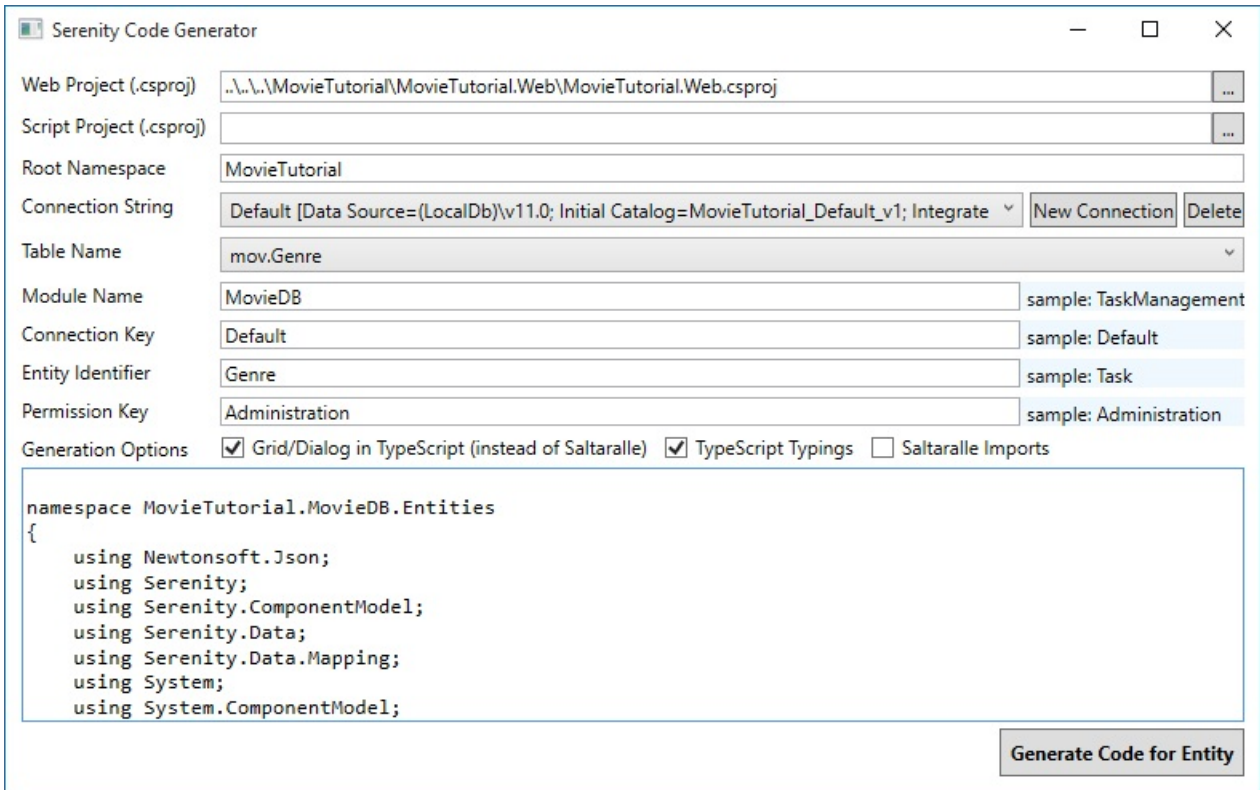
We also added a *GenreId* field to *Movie* table.

Actually a movie can have multiple genres so we should keep it in a separate *MovieGenres* table. But for now, we think it as single. We'll see how to change it to multiple later.

## Generating Code For Genre Table

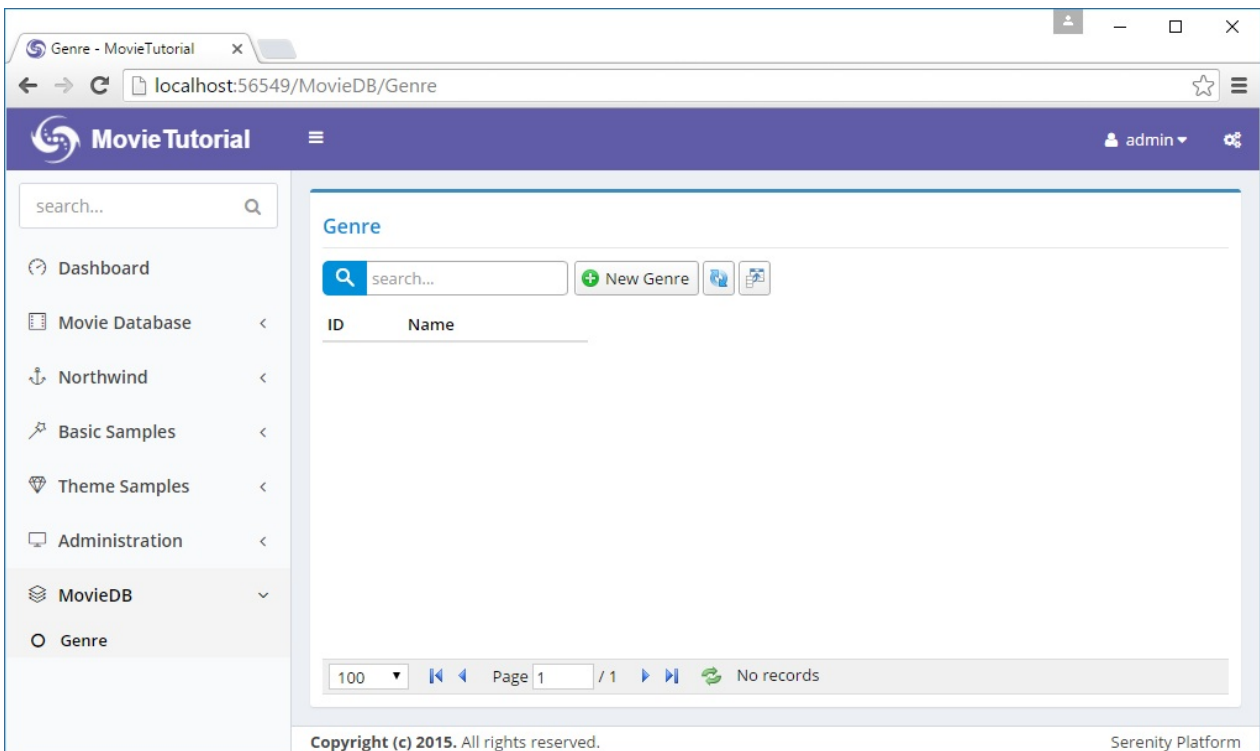
Fire sergen.exe using Package Manager Console again and generate code for *Genre* table with the parameters shown below:

Use parameters shown with `dotnet sergen g` if you are using ASP.NET Core version.



This screenshot belongs to an older version of Sergen, just use parameters shown in new version

Rebuild solution and run it. We'll get a new page like this:



As you see in screenshot, it is generated under a new section *MovieDB* instead of the one we renamed recently: *Movie Database*.

This is because *Sergen* has no idea of what customizations we performed on our *Movie* page. So we need to move it under *Movie Database* manually.

Open *Modules/Movie/GenrePage.cs*, cut the navigation link shown below:

```
[assembly:Serenity.Navigation.NavigationLink(int.MaxValue, "MovieDB/Genre",
    typeof(MovieTutorial.MovieDB.Pages.GenreController))]
```

And move it to *Modules/Common/Navigation/NavigationItems.cs*:

```
//...
[assembly: NavigationMenu(2000, "Movie Database", icon: "icon-film")]
[assembly: NavigationLink(2100, "Movie Database/Movies",
    typeof(MovieDB.MovieController), icon: "icon-camcorder")]
[assembly: NavigationLink(2200, "Movie Database/Genres",
    typeof(MovieDB.GenreController), icon: "icon-pin")]
//...
```

## Adding Several Genre Definitions

Now let's add some sample genres. I'll do it through migration, to not to repeat it in another PC, but you might want to add them manually through Genre page.



```
using FluentMigrator;
using System;

namespace MovieTutorial.Migrations.DefaultDB
{
    [Migration(20160519181800)]
    public class DefaultDB_20160519_181800_SampleGenres : Migration
    {
        public override void Up()
        {
            Insert.IntoTable("Genre").InSchema("mov")
                .Row(new
                {
                    Name = "Action"
                })
                .Row(new
                {
                    Name = "Drama"
                })
                .Row(new
                {
                    Name = "Comedy"
                })
                .Row(new
                {
                    Name = "Sci-fi"
                })
                .Row(new
                {
                    Name = "Fantasy"
                })
                .Row(new
                {
                    Name = "Documentary"
                });
        }

        public override void Down()
        {
        }
    }
}
```

## Mapping GenreId Field in MovieRow

As we did with *Kind* field before, *GenreId* field needs to be mapped in *MovieRow.cs*.

```

namespace MovieTutorial.MovieDB.Entities
{
    // ...
    public sealed class MovieRow : Row, IIdRow, INameRow
    {
        [DisplayName("Kind"), NotNull, DefaultValue(1)]
        public MovieKind? Kind
        {
            get { return (MovieKind?)Fields.Kind[this]; }
            set { Fields.Kind[this] = (Int32?)value; }
        }

        [DisplayName("Genre"), ForeignKey("[mov].Genre", "GenreId"), LeftJoin("g")]
        public Int32? GenreId
        {
            get { return Fields.GenreId[this]; }
            set { Fields.GenreId[this] = value; }
        }

        [DisplayName("Genre"), Expression("g.Name")]
        public String GenreName
        {
            get { return Fields.GenreName[this]; }
            set { Fields.GenreName[this] = value; }
        }

        // ...

        public class RowFields : RowFieldsBase
        {
            // ...
            public readonly Int32Field Kind;
            public readonly Int32Field GenreId;
            public readonly StringField GenreName;

            public RowFields()
                : base("[mov].Movie")
            {
                LocalTextPrefix = "MovieDB.Movie";
            }
        }
    }
}

```

Here we mapped *GenreId* field and also declared that it has a foreign key relation to *GenreId* field in *[mov].Genre* table using *ForeignKey* attribute.

If we did generate code for *Movie* table after we added this *Genre* table, SerGen would understand this relation by checking foreign key definition at database level, and generate similar code for us.

We also added another field, *GenreName* that is not actually a field in *Movie* table, but in *Genre* table.

Serenity entities are more like SQL views. You can bring in fields from other tables with joins.

By adding *LeftJoin("g")* attribute to *MovieId* property, we declared that whenever *Genre* table needs to be joined to, its alias will be *g*.

So when Serenity needs to select from *Movies* table, it will produce an SQL query like this:

```
SELECT t0.MovieId, t0.Kind, t0.GenreId, g.Name as GenreName
FROM Movies t0
LEFT JOIN Genre g on t0.GenreId = g.GenreId
```

This join will only be performed if a field from *Genre* table requested to be selected, e.g. its column is visible in a data grid.

By adding *Expression("g.Name")* on top of *GenreName* property, we specified that this field has an SQL expression of *g.Name*, thus it is a view field originating from our *g* join.

## Adding Genre Selection To Movie Form

Let's add *GenreId* field to our form in *MovieForm.cs*:

```
namespace MovieTutorial.MovieDB.Forms
{
    //...
    [FormScript("MovieDB.Movie")]
    [BasedOnRow(typeof(Entities.MovieRow))]
    public class MovieForm
    {
        //...
        public Int32 GenreId { get; set; }
        public MovieKind Kind { get; set; }
    }
}
```

Now if we build and run application, we'll see that a *Genre* field is added to our form. The problem is, it accepts data entry as an integer. We want it to use a dropdown.

It's clear that we need to change editor type for *GenreId* field.

## Declaring a Lookup Script for Genres

To show an editor for *Genre* field, list of genres in our database should be available at client side.

For enumeration values, it was simple, we just run T4 templates, and they copied enum declaration to script side.

Here we can't do the same. Genre list is a database based dynamic list.

Serenity has notion of *dynamic scripts* to make dynamic data available to script side in the form of runtime generated scripts.

Dynamic scripts are similar to web services, but their outputs are dynamic javascript files that can be cached on client side.

The *dynamic* here corresponds to the data they contain, not their behavior. Unlike web services, dynamic scripts can't accept any parameters. And their data is shared among all users of your site. They are like singletons or static variables.

You shouldn't try to write a dynamic script (e.g. lookup) that acts like a web service.

To declare a dynamic lookup script for Genre table, open *GenreRow.cs* and modify it like below:

```
namespace MovieTutorial.MovieDB.Entities
{
    // ...

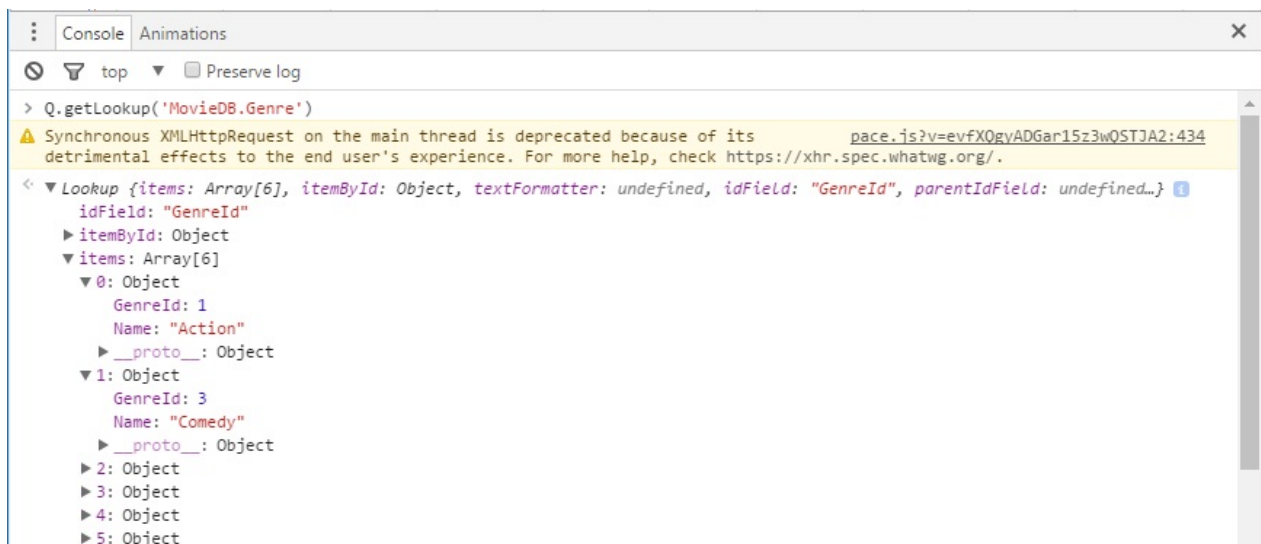
    [ConnectionString("Default"), DisplayName("Genre"), InstanceName("Genre"),
      TwoLevelCached]
    [ReadPermission("Administration")]
    [ModifyPermission("Administration")]
    [JsonConverter(typeof(JsonRowConverter))]
    [LookupScript("MovieDB.Genre")]
    public sealed class GenreRow : Row, IIdRow, INameRow
    {
        // ...
    }
}
```

We just added line with `[LookupScript("MovieDB.Genre")]`.

Rebuild your project, launch it, after logging in, open developer console by *F12*.

Type `Q.getLookup('MovieDB.Genre')`

and you will get something like this:



Here *MovieDB.Genre* is the lookup key we assigned to this lookup script when declaring it with:

```
[LookupScript("MovieDB.Genre")]
```

This step was just to show how to check if a lookup script is available client side.

Lookup key, *"MovieDB.Genre"* is case sensitive. Make sure you type exact same case everywhere.

## Using LookupEditor for Genre Field

There are two places to set editor type for *GenreId* field. One is *MovieForm.cs*, other is *MovieRow.cs*.

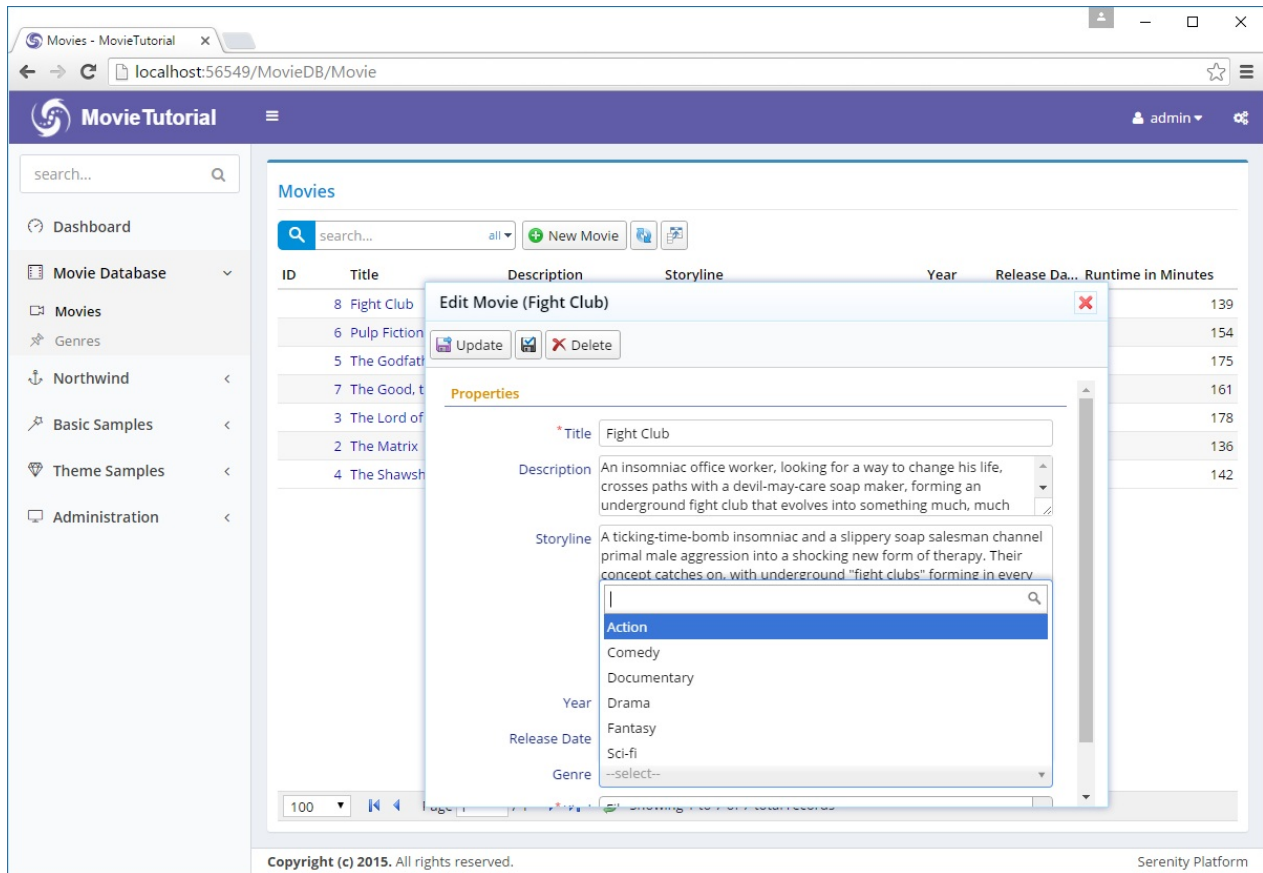
I usually prefer the latter, as it is the central place, but you may choose to set it on a form, if that editor type is specific to that form only.

Information defined on a form can't be reused. For example, grids use information in *XYZColumn.cs* / *XYZRow.cs* while dialogs use information in *XYZForm.cs* / *XYZRow.cs*. So it's usually better to define things in *XYZRow.cs*.

Open *MovieRow.cs* and add *LookupEditor* attribute to *GenreId* property as shown below:

```
[DisplayName("Genre"), ForeignKey("[mov].Genre", "GenreId"), LeftJoin("g")]
[LookupEditor("MovieDB.Genre")]
public Int32? GenreId
{
    get { return Fields.GenreId[this]; }
    set { Fields.GenreId[this] = value; }
}
```

After we build and launch our project, we'll now have a searchable dropdown (Select2.js) on our Genre field.



While defining [LookupEditor] we hardcoded the lookup key. It's also possible to reuse information on GenreRow:

```
[DisplayName("Genre"), ForeignKey("[mov].Genre", "GenreId"), LeftJoin("g")]
[LookupEditor(typeof(GenreRow))]
public Int32? GenreId
{
    get { return Fields.GenreId[this]; }
    set { Fields.GenreId[this] = value; }
}
```

This is functionally equivalent. I'd prefer latter. Here, Serenity will locate the [LookupScript] attribute on GenreRow, and get lookup key information from there. If we had no [LookupScript] attribute on GenreRow, you'd get an error on application startup:

```
Server Error in '/' Application.

'MovieTutorial.MovieDB.Entities.GenreRow' type doesn't have a
[LookupScript] attribute, so it can't be used with a LookupEditor!

Parameter name: lookupType
```

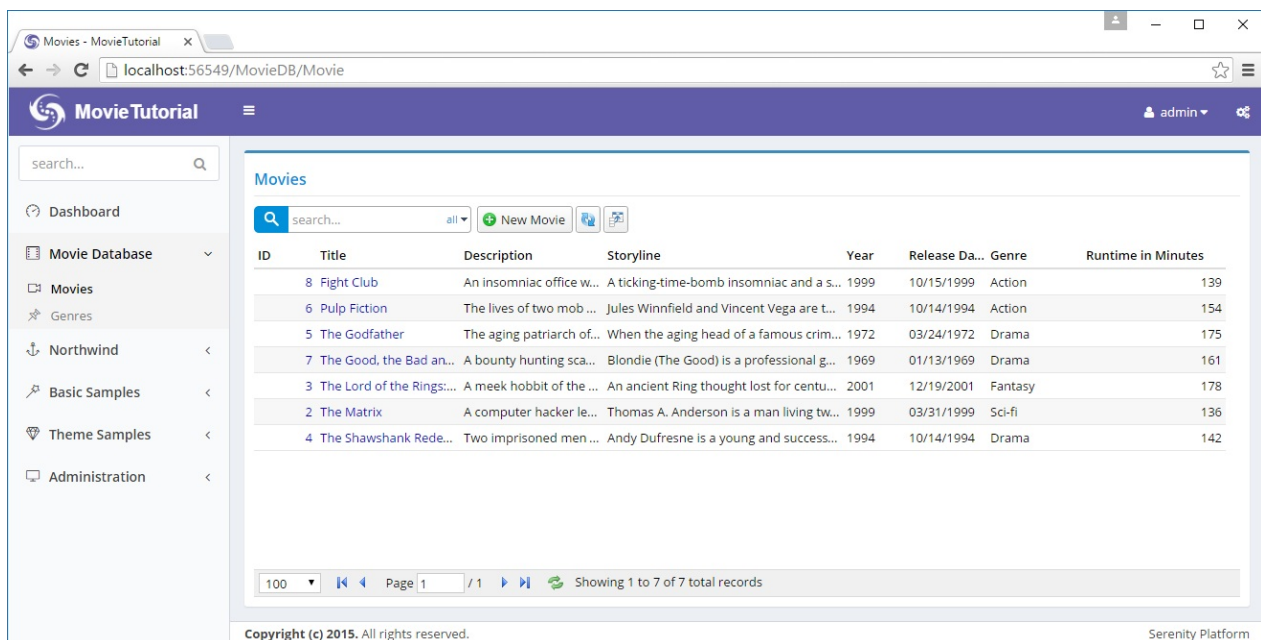
Forms are scanned at application startup, so there is no way to handle this error without fixing the issue.

## Display Genre in Movie Grid

Currently, movie genre can be edited in the form but is not displayed in Movie grid. Edit `MovieColumns.cs` to show `GenreName` (not `GenreId`).

```
namespace MovieTutorial.MovieDB.Columns
{
    // ...
    public class MovieColumns
    {
        //...
        [Width(100)]
        public String GenreName { get; set; }
        [DisplayName("Runtime in Minutes"), Width(150), AlignRight]
        public Int32 Runtime { get; set; }
    }
}
```

Now `GenreName` is shown in the grid.



The screenshot shows a web application interface for "Movie Tutorial". The main content area displays a table of movies. The table has the following columns: ID, Title, Description, Storyline, Year, Release Date, Genre, and Runtime in Minutes. The data rows are as follows:

ID	Title	Description	Storyline	Year	Release Date	Genre	Runtime in Minutes
8	Fight Club	An insomniac office w...	A ticking-time-bomb insomniac and a s...	1999	10/15/1999	Action	139
6	Pulp Fiction	The lives of two mob ...	Jules Winnfield and Vincent Vega are t...	1994	10/14/1994	Action	154
5	The Godfather	The aging patriarch of...	When the aging head of a famous crim...	1972	03/24/1972	Drama	175
7	The Good, the Bad an...	A bounty hunting sca...	Blondie (The Good) is a professional g...	1969	01/13/1969	Drama	161
3	The Lord of the Rings...	A meek hobbit of the ...	An ancient Ring thought lost for centu...	2001	12/19/2001	Fantasy	178
2	The Matrix	A computer hacker le...	Thomas A. Anderson is a man living tw...	1999	03/31/1999	Sci-fi	136
4	The Shawshank Rede...	Two imprisoned men ...	Andy Dufresne is a young and success...	1994	10/14/1994	Drama	142

At the bottom of the grid, there is a pagination control showing "Page 1 / 1" and "Showing 1 to 7 of 7 total records".

## Making It Possible To Define A New Genre Inplace

While setting genre for our sample movies, we notice that *The Good, the Bad and the Ugly* is *Western* but there is no such genre in *Genre* dropdown yet (so I had to choose Drama).

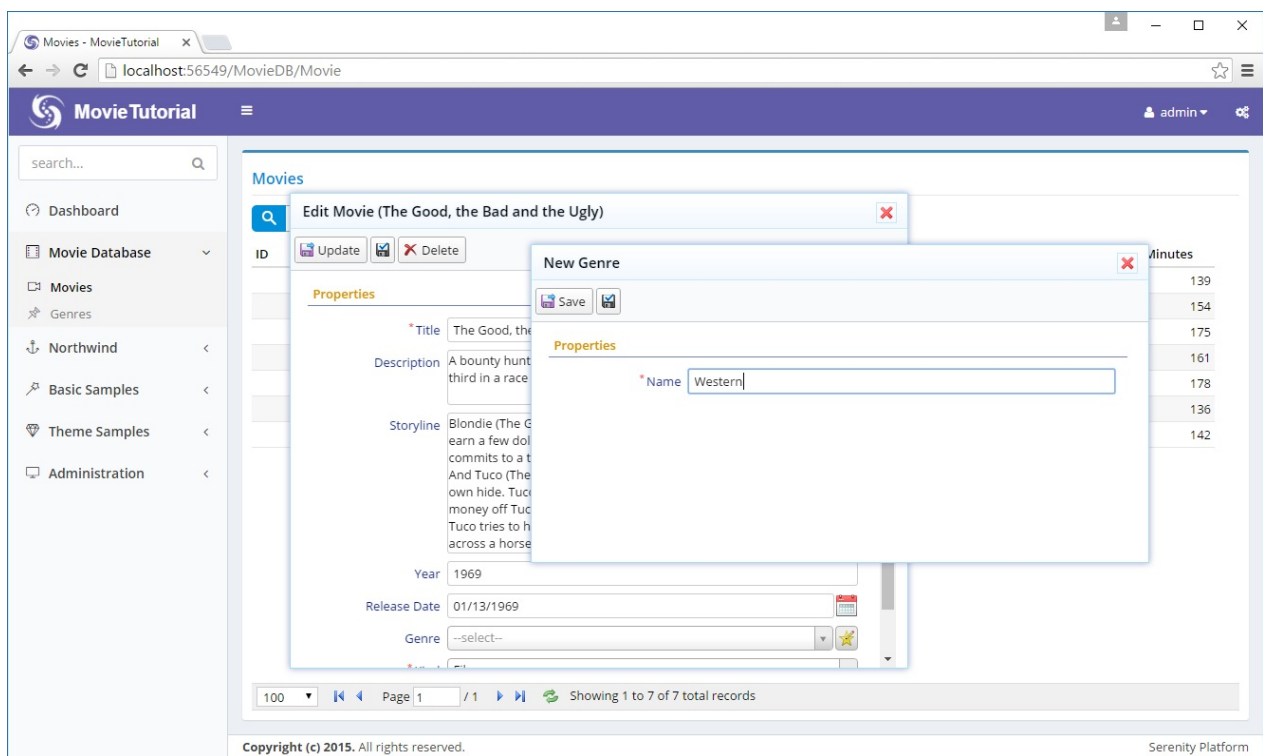
One option is to open Genres page, add it, and come back to movie form again. Not so pretty...

Fortunately, Serenity has integrated inplace item definition ability for lookup editors.

Open `MovieRow.cs` and modify `LookupEditor` attribute like this:

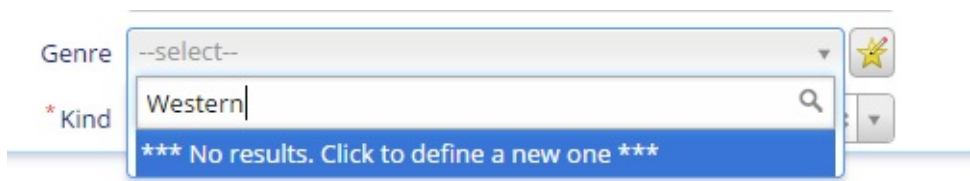
```
[DisplayName("Genre"), ForeignKey("[mov].Genre", "GenreId"), LeftJoin("g")]
[LookupEditor(typeof(GenreRow), InplaceAdd = true)]
public Int32? GenreId
{
    get { return Fields.GenreId[this]; }
    set { Fields.GenreId[this] = value; }
}
```

Now we can define a new Genre by clicking star/pen icon next to genre field.



Here we also see that we can use a dialog from another page (`GenreDialog`) in the movies page. In Serenity applications, all client side objects (dialogs, grids, editors, formatters etc.) are self-contained reusable components (widgets) that are not bound to any page.

It is also possible to start typing in genre editor, and it will provide you with an option to add a new genre.





## How Did It Determine Which Dialog Type To Use

You probably didn't notice this detail. Our lookup editor for genre selection, automatically opened a new *GenreDialog* when you wanted to add a new genre inplace.

Here, our lookup editor made use of a convention. Because its lookup key is *MovieDB.Genre*, it searched for a dialog class with full names below:

```
MovieDB.GenreDialog
MovieTutorial.MovieDB.GenreDialog
...
...
```

Luckily, we have a *GenreDialog*, which is defined in *Modules/Genre/GenreDialog.ts* and its full name is *MovieTutorial.MovieDB.GenreDialog*.

```
namespace MovieTutorial.MovieDB {

  @Serenity.Decorators.registerClass()
  @Serenity.Decorators.responsive()
  export class GenreDialog extends Serenity.EntityDialog<GenreRow, any> {
    protected getFormKey() { return GenreForm.formKey; }
    protected getIdProperty() { return GenreRow.idProperty; }
    protected getLocalTextPrefix() { return GenreRow.localTextPrefix; }
    protected getNameProperty() { return GenreRow.nameProperty; }
    protected getService() { return GenreService.baseUrl; }

    protected form = new GenreForm(this.idPrefix);
  }
}
```

If, lookup key for *GenreRow* and its dialog class didn't match, we would get an error in browser console, as soon as we click the inplace add button:

```
Uncaught MovieDB.GenreDialog dialog class is not found!
```

But this is not the case as they match. In such a case, either you'd have to use a compatible lookup key like *"ModuleName.RowType"*, or you'd need to specify dialog type explicitly:

```
[DisplayName("Genre"), ForeignKey("[mov].Genre", "GenreId"), LeftJoin("g")]
[LookupEditor(typeof(GenreRow), InplaceAdd = true, DialogType = "MovieDB.Genre")]
public Int32? GenreId
{
    get { return Fields.GenreId[this]; }
    set { Fields.GenreId[this] = value; }
}
```

You shouldn't specify *Dialog* suffix, nor the full namespace, e.g.

*MovieTutorial.MovieDB.Genre*, as Serenity automatically searches for them.

## Adding Quick Filter for Genre To Grid

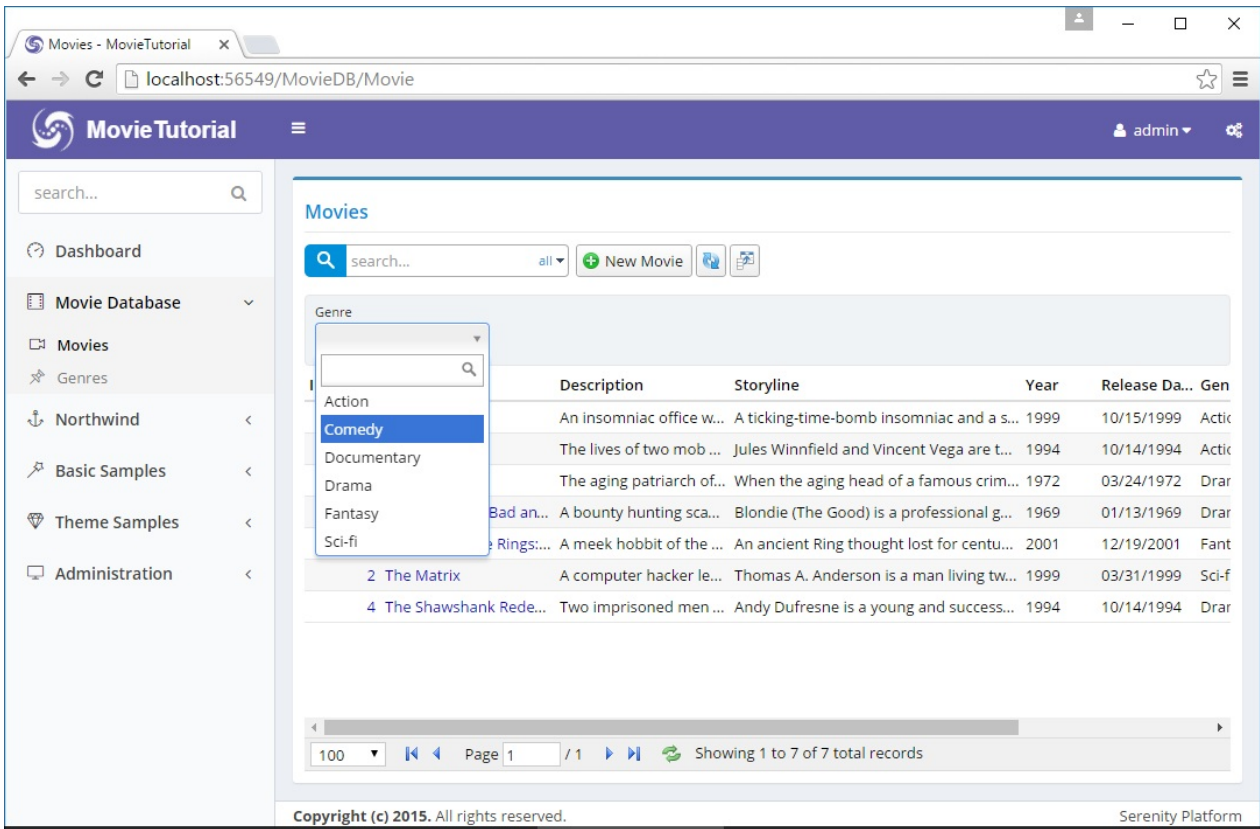
As our list of movies becomes larger, we might need to filter movies based on values of some fields, besides the quick search functionality.

Serenity has several filtering methods. One of them is Quick Filter, which we'll use on Genre field.

Edit *Modules/MovieDB/Movie/MovieColumns.cs* to add a [QuickFilter] attribute on top of GenreName field:

```
public class MovieColumns
{
    //...
    public DateTime ReleaseDate { get; set; }
    [width(100), QuickFilter]
    public String GenreName { get; set; }
    [DisplayName("Runtime in Minutes"), Width(150), AlignRight]
    public Int32 Runtime { get; set; }
}
```

Build and navigate to Movies page. You'll a quick filtering dropdown for genre field is available:



The field that is filtered is actually *GenreId* not *GenreName* that we attached this attribute to. Serenity is clever enough to understand this relation, and determined editor type to use by looking at attributes of *GenreId* property in *GenreRow.cs*.

## Re-running T4 Templates

As we added a new entity to our application, we should run T4 templates after building solution.

## Updating Serenity Packages (ASP.NET MVC Version)

When i started writing this tutorial, Serenity (NuGet packages containing Serenity assemblies and standard scripts libraries) and Serene (the application template) was at version 2.1.8.

When you read this you are probably using a later version, so you might not have to update serenity yet.

But, i want to show how you can update Serenity NuGet packages, in case another version comes out in the future.

I prefer to work with NuGet from *Package Manager Console* instead of using NuGet GUI interface as it performs much faster.

So, click *View -> Other Windows -> Package Manager Console*.

Type:

```
Update-Package Serenity.Web
```

This will also update following NuGet packages in *MovieTutorial.Web* because of dependencies:

```
Serenity.Core  
Serenity.Data  
Serenity.Data.Entity  
Serenity.Services
```

To update Serenity.CodeGenerator (containg *sergen.exe*), type:

```
Update-Package Serenity.CodeGenerator
```

Serenity.CodeGenerator is also installed in *MovieTutorial.Web* project.

```
During updates, if NuGet asks to override changes in some script files, you can safely say yes unless you did manual modifications to Serenity script files (which i suggest you avoid).
```

## Updating Serenity Packages (ASP.NET

## Core Version)

Theoretically, you should be able to update Serenity just like ASP.NET MVC version using NuGet package manager console, but it might not work, probably due to some conditionals in CSPROJ file confusing NuGet.

These conditionals are there to support switching easily to full .NET Framework if you have to.

Right click your project file, click *Edit MySerene.csproj*:

```
<PackageReference Include="Serenity.Web" Version="3.0.5" />
<PackageReference Include="Serenity.Web.AspNetCore" Version="3.0.5" />
<DotNetCliToolReference Include="Serenity.CodeGenerator" Version="3.0.5" >
```

Find three lines that include *Serenity.Web*, *Serenity.Web.AspNetCore* and *Serenity.CodeGenerator* like shown above and change their versions to latest *Serenity* version.

Open a command prompt in your project directory and type these two lines:

```
dotnet restore
dotnet sergen restore
```

## Building Project

Now rebuild your solution and it should build successfully.

From time to time, breaking changes might happen in Serenity, but they are kept to minimum, and you might have to do a few manual changes in your application code.

Such changes are documented with a [BREAKING CHANGE] tag in change log at: <https://github.com/volkanceylan/Serenity/blob/master/CHANGELOG.md>

If you still have a problem after upgrade, feel free to open an issue at: <https://github.com/volkanceylan/Serenity/issues>

## What Is Updated

Updating Serenity NuGet packages, takes Serenity assemblies up to the latest version.

It might also update some other third-party packages like ASP.NET MVC, FluentMigrator, Select2.js, SlickGrid etc.

Please don't update Select2.js to a version after 3.5.1 yet as it has some compability problems with jQuery validation.

Serenity.Web package also comes with some static script and css resources like the following:

```
Content/serenity/serenity.css
Scripts/saltarelle/mscorlib.js
Scripts/saltarelle/linq.js
Scripts/serenity/Serenity.CoreLib.js
Scripts/serenity/Serenity.Script.UI.js
```

So, these and a few more are also updated in MovieApplication.Web.

## What Is Not Updated (OR Can't Be Updated Automatically)

Updating Serenity packages, updates Serenity assemblies and most static scripts, but not all *Serene* template content is updated.

We are trying to keep updating your application as simple as possible, but Serene is just a project template, not a static package. Your application is a customizable copy of Serene.

You might have done modifications to application source, so updating a Serene application created with an older version of Serene template, might not be as easy as it sounds.

So sometimes you might have to create a new Serene application with up-to-date Serene template version, and compare it to your application, and merge features you need. This is a manual process.

Usually, updating Serenity packages is enough. Updating Serene itself is not required unless you need some recent features from latest Serene version.

We have some plans to make parts of Serene template also a NuGet package, but it is still not trivial how to update your application without overriding your changes, e.g. to shared code like Navigation items. And what if you removed Northwind code, but our update reinstalls it? I'm open to suggestions...

# Allowing Multiple Genre Selection

It happens. Requirements change. Now we want to allow selecting multiple genres for a movie.

For this, we need a M-N mapping table that will let us link any movie to multiple genres.

## Creating MovieGenres Table

As usual, we start with a migration:

```
Modules/Common/Migrations/DefaultDB/  
DefaultDB_20160528_115400_MovieGenres.cs:
```

```

using FluentMigrator;

namespace MovieTutorial.Migrations.DefaultDB
{
    [Migration(20160528115400)]
    public class DefaultDB_20160528_115400_MovieGenres : Migration
    {
        public override void Up()
        {
            Create.Table("MovieGenres").InSchema("mov")
                .WithColumn("MovieGenreId").AsInt32()
                    .Identity().PrimaryKey().NotNullable()
                .WithColumn("MovieId").AsInt32().NotNullable()
                    .ForeignKey("FK_MovieGenres_MovieId",
                        "mov", "Movie", "MovieId")
                .WithColumn("GenreId").AsInt32().NotNullable()
                    .ForeignKey("FK_MovieGenres_GenreId",
                        "mov", "Genre", "GenreId");

            Execute.Sql(
                @"INSERT INTO mov.MovieGenres (MovieId, GenreId)
                SELECT m.MovieId, m.GenreId
                FROM mov.Movie m
                WHERE m.GenreId IS NOT NULL");

            Delete.ForeignKey("FK_Movie_GenreId")
                .OnTable("Movie").InSchema("mov");
            Delete.Column("GenreId")
                .FromTable("Movie").InSchema("mov");
        }

        public override void Down()
        {
        }
    }
}

```

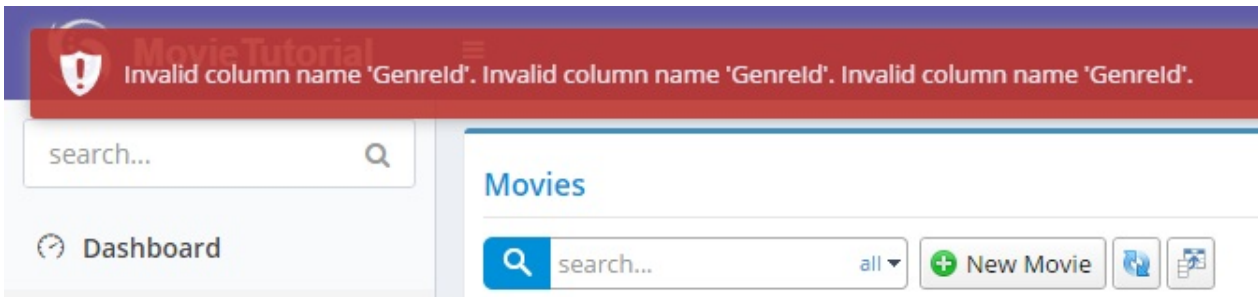
I tried to save existing Genre declarations on Movie table, by copying them to our new MovieGenres table. The line above with *Execute.Sql* does this.

Then we should remove GenreId column, by first deleting the foreign key declaration *FK\_Movie\_GenreId* that we defined on it previously.

## Deleting Mapping for GenreId Column

As soon as you build and open the Movies page, you'll get this error:





This is because we still have mapping for GenreId column in our row. Error above is received from AJAX call to *List* service handler for *Movie* table.

Repeating of error message originates from SQL server. *MovieId* column name passes several times within the generated dynamic SQL.

Remove GenreId and GenreName properties and their related field objects from *MovieRow.cs*:

```
// remove this
public Int32? GenreId
{
    get { return Fields.GenreId[this]; }
    set { Fields.GenreId[this] = value; }
}

// remove this
public String GenreName
{
    get { return Fields.GenreName[this]; }
    set { Fields.GenreName[this] = value; }
}

public class RowFields : RowFieldsBase
{
    // and remove these
    public Int32Field GenreId;
    public StringField GenreName;
}
```

Remove GenreName property from *MovieColumns.cs*:

```
// remove this
[Width(100), QuickFilter]
public String GenreName { get; set; }
```

Remove GenreId property from *MovieForm.cs*:

```
// remove this
public Int32 GenreId { get; set; }
```

After building, we at least have a working *Movies* page again.

## Generating Code For MovieGenres Table

Fire up sergen and generate code for *MovieGenres* table as usual:

Serenity Code Generator

Web Project (.csproj) ..\..\..\MovieTutorial\MovieTutorial.Web\MovieTutorial.Web.csproj

Script Project (.csproj)

Root Namespace MovieTutorial

Connection String Default [Data Source=(LocalDb)\v11.0; Initial Catalog=MovieTutorial\_Default\_v1; Integrate] New Connection Delete

Table Name mov.MovieGenres

Module Name MovieDB sample: TaskManagement

Connection Key Default sample: Default

Entity Identifier MovieGenres sample: Task

Permission Key Administration sample: Administration

Generation Options  Grid/Dialog in TypeScript (instead of Saltaralle)  TypeScript Typings  Saltaralle Imports

```
namespace MovieTutorial.MovieDB.Entities
{
    using Newtonsoft.Json;
    using Serenity;
    using Serenity.ComponentModel;
    using Serenity.Data;
    using Serenity.Data.Mapping;
    using System;
    using System.ComponentModel;
}
```

Generate Code for Entity

As we're not going to edit movie genres from a separate page, you can safely delete the generated files below:

```
MovieGenresColumns.cs
MovieGenresDialog.ts
MovieGenresEndpoint.cs
MovieGenresForm.cs
MovieGenresGrid.cs
MovieGenresIndex.cshtml
MovieGenresPage.cs
```

You can also remove CSS entries for s-MovieDB-MovieGenresDialog from *site.less*.

Only leave last two files, *MovieGenresRow.cs* and *MovieGenresRepository.cs*.

After building, run T4 templates to be sure, no T4 generated files related to *MovieGenresForm* etc. is left behind.

## Adding GenreList Field

As one movie might have multiple genres now, instead of a `Int32` property, we need a list of `Int32` values, e.g. `List<Int32>`. Add the `GenreList` property to `MovieRow.cs`:

You might have to add `System.Collections.Generic` to usings.

```
//...
[DisplayName("Kind"), NotNull, DefaultValue(MovieKind.Film)]
public MovieKind? Kind
{
    get { return (MovieKind?)Fields.Kind[this]; }
    set { Fields.Kind[this] = (Int32?)value; }
}

[DisplayName("Genres")]
[LookupEditor(typeof(GenreRow), Multiple = true), NotMapped]
[LinkingSetRelation(typeof(MovieGenresRow), "MovieId", "GenreId")]
public List<Int32> GenreList
{
    get { return Fields.GenreList[this]; }
    set { Fields.GenreList[this] = value; }
}

public class RowFields : RowFieldsBase
{
    //...
    public Int32Field Kind;
    public ListField<Int32> GenreList;
}
```

Our property has `[LookupEditor]` attribute just like `GenreId` property had, but with one difference. This one accepts multiple genre selection. We set it with `Multiple = true` argument.

This property also has `NotMapped` flag, which is something similar to `Unmapped` fields in Serenity. It specifies that this property has no matching database column in database.

We don't have a `GenreList` column in `Movie` table, so we should set it as an unmapped field. Otherwise, Serenity will try to `SELECT` it, and we'll get SQL errors.

In the next line, we use another new attribute, `LinkingSetRelation`:

```
[LinkingSetRelation(typeof(MovieGenresRow), "MovieId", "GenreId")]
```

This is an attribute which is specific to M-N relations that links a row in this table to multiple rows from another table.

First argument of it is the type of M-N mapping row, which is *MovieGenresRow* here.

Second argument is the property name of field in that row (*MovieGenresRow*) that matches this row's ID property, e.g. *MovieId*.

Third argument is the property name of field in that row (*MovieGenresRow*) that links multiple Genres by their IDs, e.g. *GenreId*.

*LinkingSetRelation* has a related Serenity service behavior, named *LinkingSetRelationBehavior* that is automatically activated for all fields with a *LinkingSetRelation* attribute.

This behavior, will intercept service handlers for *Create*, *Update*, *Delete*, *Retrieve* and *List* and inject code to populate or update our *GenreList* column and its related *MovieGenres* table.

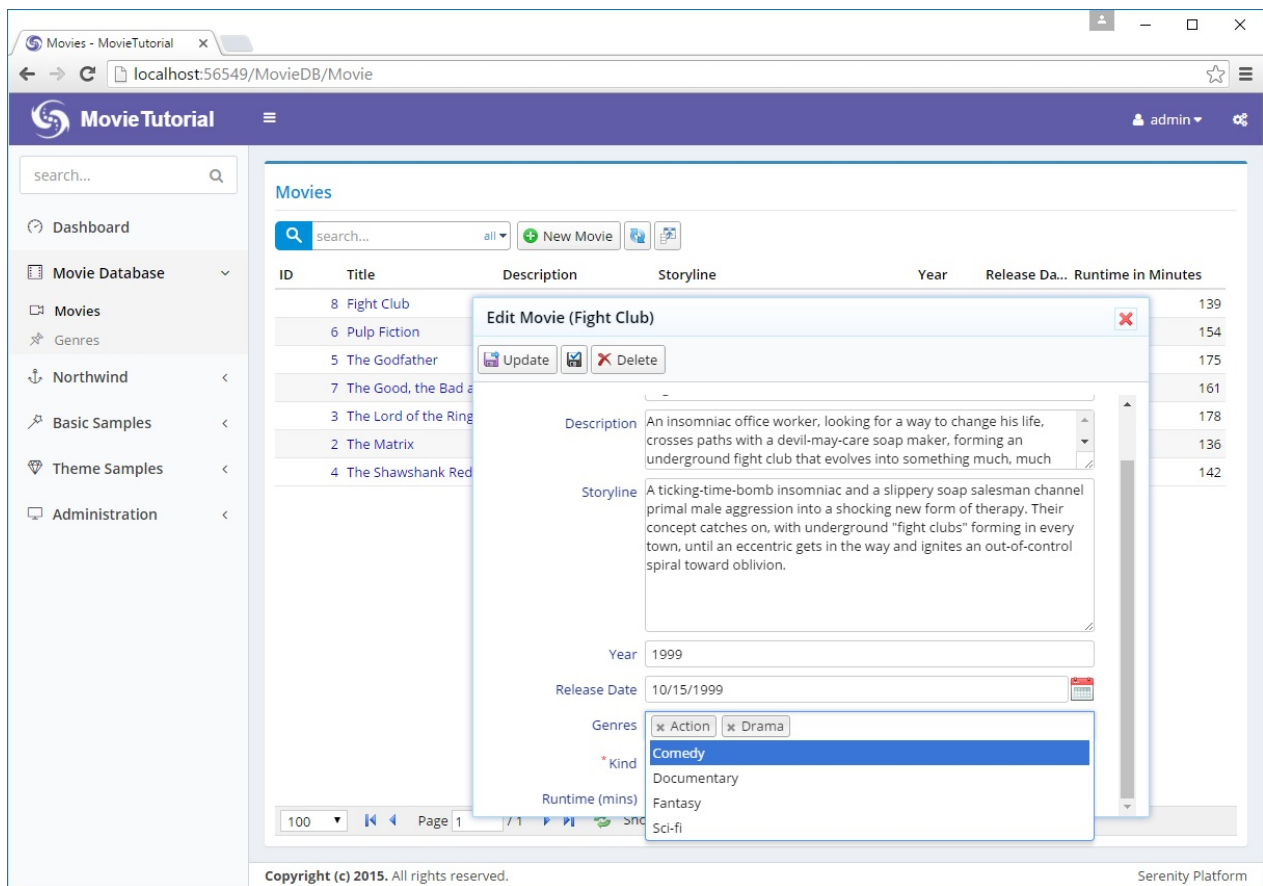
We'll talk about Serenity service behaviors in following chapters.

## Adding Genre List To Form

Edit *MovieForm.cs* and add *GenreList* property:

```
public class MovieForm
{
    //...
    public List<Int32> GenreList { get; set; }
    public MovieKind Kind { get; set; }
    public Int32 Runtime { get; set; }
}
```

Now we can add multiple genres to a Movie:



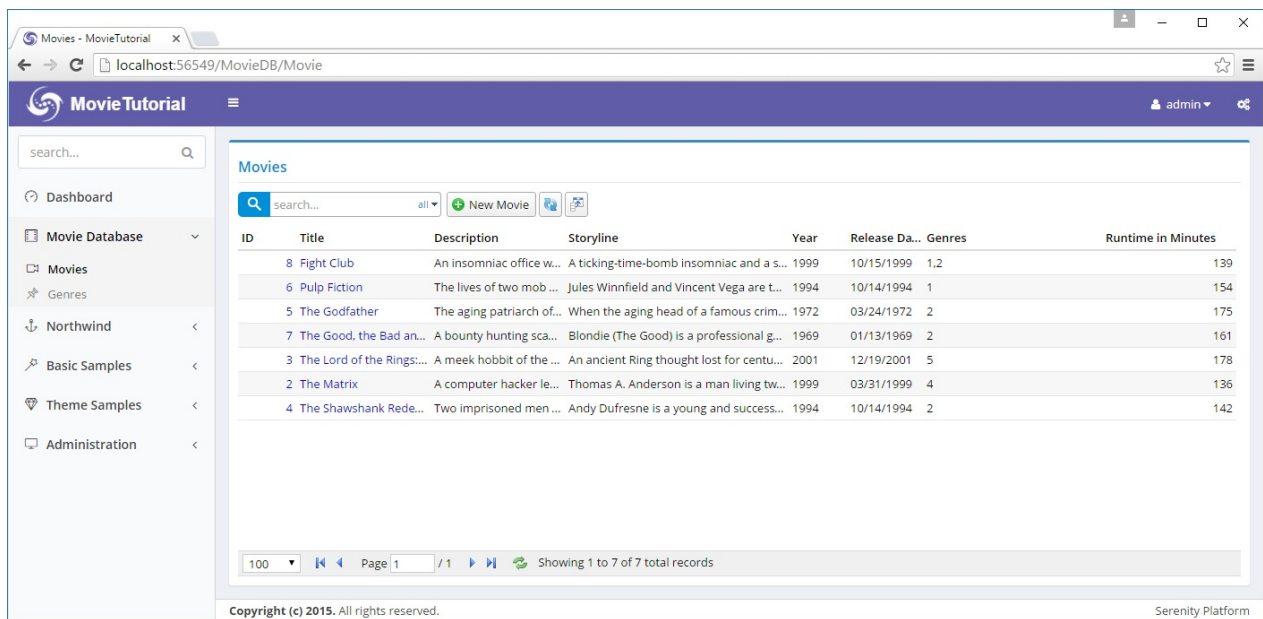
## Showing Selected Genres in a Column

Previously, when we had only one Genre per Movie. We could show the selected genre in a column, by adding a view field to *MovieRow.cs*. It is not going to be so simple this time.

Let's start by adding *GenreList* property to *MovieColumns.cs*:

```
public class MovieColumns
{
    //...
    [Width(200)]
    public List<Int32> GenreList { get; set; }
    [DisplayName("Runtime in Minutes"), Width(150), AlignRight]
    public Int32 Runtime { get; set; }
}
```

This is what we got:



GenreList column contains a list of Int32 values, which corresponds to an array in Javascript. Luckily, Javascript .toString() method for an array returns items separated by comma, so we got "1,2" for *Fight Club* movie.

We would prefer genre names instead of Genre IDs, so it's clear that we need to *format* these values, by converting GenreId to their Genre name equivalents.

## Creating GenreListFormatter Class

It's time to write a SlickGrid column formatter. Create file *GenreListFormatter.ts* next to *MovieGrid.ts*:

```

namespace MovieTutorial.MovieDB {

    @Serenity.Decorators.registerFormatter()
    export class GenreListFormatter implements Slick.Formatter {
        format(ctx: Slick.FormatterContext) {
            let idList = ctx.value as number[];
            if (!idList || !idList.length)
                return "";

            let byId = GenreRow.getLookup().itemById;

            return idList.map(x => {
                let g = byId[x];
                if (!g)
                    return x.toString();

                return Q.htmlEncode(g.Name);
            }).join(", ");
        }
    }
}

```

Here we define a new formatter, *GenreListFormatter* and register it with Serenity type system, using `@Serenity.Decorators.registerFormatter` decorator. Decorators are similar to .NET attributes.

All formatters should implement `Slick.Formatter` interface, which has a *format* method that takes a *ctx* parameter of type *Slick.FormatterContext*.

*ctx*, which is the formatting context, is an object with several members. One of them is *value* that contains the column value for current grid row/column being formatted.

As we know that we'll use this formatter on column with a `List<Int32>` value, we start by casting value to *number[]*.

There is no `Int32` type in Javascript. `Int32`, `Double`, `Single` etc. corresponds to `number` type. Also, generic `List<>` type in C# corresponds to an `Array` in Javascript.

If the array is empty or null, we can safely return an empty string:

```

let idList = ctx.value as number[];
if (!idList || !idList.length)
    return "";

```

Then we get a reference to *Genre* lookup, which has a dictionary of *Genre* rows in its *itemById* property:

```
let byId = GenreRow.getLookup().itemById;
```

Next, we start mapping these ID values in our *idList* to their Genre name equivalents, using *Array.map* function in Javascript, which is pretty similar to LINQ Select statement:

```
return idList.map(x => {
```

We lookup an ID in our Genre dictionary. It should be in dictionary, but we play safe here, and return its numeric value, if the genre is not found in dictionary.

```
let g = byId[x];
if (!g)
    return x.toString();
```

If we could find the genre row, corresponding to this ID, we return its Name value. We should HTML encode the genre name, just in case it contains invalid HTML characters, like `<`, `>` or `&`.

```
return Q.htmlEncode(g.Name);
```

We could also write a generic formatter that works with any type of lookup list, but it's beyond scope of this tutorial.

## Assigning GenreListFormatter to GenreList Column

As we defined a new formatter class, we should build and transform T4 files, so that we can reference *GenreListFormatter* in server side code.

After building and transforming, open *MovieColumns.cs* and attach this formatter to *MovieList* property:

```
public class MovieColumns
{
    //...
    [Width(200), GenreListFormatter]
    public List<Int32> GenreList { get; set; }
    [DisplayName("Runtime in Minutes"), Width(150), AlignRight]
    public Int32 Runtime { get; set; }
}
```

Now we can see Genre names in Genres column:



# Allowing Multiple Genre Selection

The screenshot displays a web application titled "Movie Tutorial" running on localhost:56549/MovieDB/Movie. The main content area shows a table of movies with the following data:

ID	Title	Description	Storyline	Year	Release Da...	Genres	Runtime in Minutes
8	Fight Club	An insomniac office w...	A ticking-time-bomb insomniac and a s...	1999	10/15/1999	Action, Drama	139
6	Pulp Fiction	The lives of two mob ...	Jules Winnfield and Vincent Vega are t...	1994	10/14/1994	Action	154
5	The Godfather	The aging patriarch of...	When the aging head of a famous crim...	1972	03/24/1972	Drama	175
7	The Good, the Bad an...	A bounty hunting sca...	Blondie (The Good) is a professional g...	1969	01/13/1969	Drama	161
3	The Lord of the Rings:...	A meek hobbit of the ...	An ancient Ring thought lost for centu...	2001	12/19/2001	Fantasy	178
2	The Matrix	A computer hacker le...	Thomas A. Anderson is a man living tw...	1999	03/31/1999	Sci-fi	136
4	The Shawshank Rede...	Two imprisoned men ...	Andy Dufresne is a young and success...	1994	10/14/1994	Drama	142

The interface also features a sidebar with navigation options: Dashboard, Movie Database, Movies, Genres, Northwind, Basic Samples, Theme Samples, and Administration. The footer contains the text "Copyright (c) 2015. All rights reserved." and "Serenity Platform".

# Filtering with Multiple Genre List

Remember that when we had only one Genre per Movie, it was easy to quick filter, by adding a [QuickFilter] attribute to GenreId field.

Let's try to do similar in MovieColumns.cs:

```
[ColumnsScript("MovieDB.Movie")]
[BasedOnRow(typeof(Entities.MovieRow))]
public class MovieColumns
{
    //...
    [Width(200), GenreListFormatter, QuickFilter]
    public List<Int32> GenreList { get; set; }
}
```

As soon as you type a Genre into Genres you'll have this error:

The screenshot shows a web application interface for a movie database. At the top, a red error banner displays the message: "Invalid column name 'GenreList'. Invalid column name 'GenreList'". Below the banner, there is a search bar and a navigation menu on the left. The main content area shows a table of movies with the following columns: ID, Title, Description, Storyline, Year, Release Da..., Genres, and Runtime in Minutes. The table contains 7 rows of movie data. The Genres column for the first row shows "Action, Drama". A search box above the table has "Comedy" entered, and a "New Movie" button is visible.

As of Serenity 2.6.3, LinkingSetRelation will automatically handle equality filter for its field, so you won't get this error and it will just work. Anyway, it's still recommended to follow steps below as it is a good sample for defining custom list requests and handling them when required.

ListHandler tried to filter by GenreList field, but as there is no such column in database, we got this error.

So, now we have to handle it somehow.

## Declaring MovieListRequest Type

As we are going to do something non-standard, e.g. filtering by values in a linking set table, we need to prevent `ListHandler` from filtering itself on `GenreList` property.

We could process the request *Criteria* object (which is similar to an expression tree) using a visitor and handle `GenreList` ourselves, but it would be a bit complex. So I'll take a simpler road for now.

Let's take a subclass of standard *ListRequest* object and add our Genres filter parameter there. Add a *MovieListRequest.cs* file next to *MovieRepository.cs*:

```
namespace MovieTutorial.MovieDB
{
    using Serenity.Services;
    using System.Collections.Generic;

    public class MovieListRequest : ListRequest
    {
        public List<int> Genres { get; set; }
    }
}
```

We added a *Genres* property to our list request object, which will hold the optional *Genres* we want movies to be filtered on.

## Modifying Repository/Endpoint for New Request Type

For our list handler and service to use our new list request type, need to do changes in a few places.

Start with *MovieRepository.cs*:

```
public class MovieRepository
{
    //...
    public ListResponse<MyRow> List(IDbConnection connection, MovieListRequest request
    )
    {
        return new MyListHandler().Process(connection, request);
    }

    //...
    private class MyListHandler : ListRequestHandler<MyRow, MovieListRequest> { }
}
```

We changed `ListRequest` to `MovieListRequest` in `List` method and added a generic parameter to `MyListHandler`, to use our new type instead of `ListRequest`.

And another little change in *MovieEndpoint.cs*, which is the actual web service:

```
public class MovieController : ServiceEndpoint
{
    //...
    public ListResponse<MyRow> List(IDbConnection connection, MovieListRequest request
)
    {
        return new MyRepository().List(connection, request);
    }
}
```

Now its time to build and transform templates, so our *MovieListRequest* object and related service methods will be available at client side.

## Moving Quick Filter to Genres Parameter

We still have the same error as quick filter is not aware of the parameter we just added to list request type and still uses the *Criteria* parameter.

Need to intercept quick filter item and move the genre list to *Genres* property of our *MovieListRequest*.

Edit *MovieGrid.ts*:

```
export class MovieGrid extends Serenity.EntityGrid<MovieRow, any> {
    //...
    protected getQuickFilters() {
        let items = super.getQuickFilters();

        var genreListFilter = Q.first(items, x =>
            x.field == MovieRow.Fields.GenreList);

        genreListFilter.handler = h => {
            var request = (h.request as MovieListRequest);
            var values = (h.widget as Serenity.LookupEditor).values;
            request.Genres = values.map(x => parseInt(x, 10));
            h.handled = true;
        };

        return items;
    }
}
```

*getQuickFilters* is a method that is called to get a list of quick filter objects for this grid type.

By default grid enumerates properties with [QuickFilter] attributes in MovieColumns.cs and creates suitable quick filter objects for them.

We start by getting list of QuickFilter objects from super class.

```
let items = super.getQuickFilters();
```

Then locate the quick filter object for *GenreList* property:

```
var genreListFilter = Q.first(items, x =>
    x.field == MovieRow.Fields.GenreList);
```

Actually there is only one quick filter now, but we want to play safe.

Next step is to set the *handler* method. This is where a quick filter object reads the editor value and applies it to request's *Criteria* (if multiple) or *EqualityFilter* (if single value) parameters, just before its submitted to list service.

```
genreListFilter.handler = h => {
```

Then we get a reference to current *ListRequest* being prepared:

```
var request = (h.request as MovieListRequest);
```

And read the current value in lookup editor:

```
var values = (h.widget as Serenity.LookupEditor).values;
```

Set it in *request.Genres* property:

```
request.Genres = values.map(x => parseInt(x, 10));
```

As values is a list of string, we needed to convert them to integer.

Last step is to set *handled* to true, to disable default behavior of quick filter object, so it won't set *Criteria* or *EqualityFilter* itself:

```
h.handled = true;
```

Now we'll no longer have *Invalid Column Name GenreList* error but Genres filter is not applied server side yet.

## Handling Genre Filtering In Repository

Modify *MyListHandler* in *MovieRepository.cs* like below:

```
private class MyListHandler : ListRequestHandler<MyRow, MovieListRequest>
{
    protected override void ApplyFilters(SqlQuery query)
    {
        base.ApplyFilters(query);

        if (!Request.Genres.IsEmptyOrNull())
        {
            var mg = Entities.MovieGenresRow.Fields.As("mg");

            query.Where(Criteria.Exists(
                query.SubQuery()
                    .From(mg)
                    .Select("1")
                    .Where(
                        mg.MovieId == fld.MovieId &&
                        mg.GenreId.In(Request.Genres))
                    .ToString()));
        }
    }
}
```

*ApplyFilters* is a method that is called to apply filters specified in list request's *Criteria* and *EqualityFilter* parameters. This is a good place to apply our custom filter.

We first check if *Request.Genres* is null or an empty list. If so no filtering needs to be done.

Next, we get a reference to *MovieGenresRow*'s fields with alias *mg*.

```
var mg = Entities.MovieGenresRow.Fields.As("mg");
```

Here it needs some explanation, as we didn't cover Serenity entity system yet.

Let's start by not aliasing *MovieGenresRow.Fields*:

```
var x = MovieGenresRow.Fields;
new SqlQuery()
    .From(x)
    .Select(x.MovieId)
    .Select(x.GenreId);
```

If we wrote a query like above, its SQL output would be something like this:

```
SELECT t0.MovieId, t0.GenreId FROM MovieGenres t0
```

Unless told otherwise, Serenity always assigns *t0* to a row's primary table. Even if we named *MovieGenresRow.Fields* as variable *x*, its alias will still be *t0*.

Because when compiled, *x* won't be there and Serenity has no way to know its variable name. Serenity entity system doesn't use an expression tree like in LINQ to SQL or Entity Framework. It makes use of very simple string / query builders.

So, if wanted it to use *x* as an alias, we'd have to write it explicitly:

```
var x = MovieGenresRow.Fields.As("x");
new SqlQuery()
    .From(x)
    .Select(x.MovieId)
    .Select(x.GenreId);
```

...results at:

```
SELECT x.MovieId, x.GenreId FROM MovieGenres x
```

In *MyListHandler*, which is for *MovieRow* entities, *t0* is already used for *MovieRow* fields. So, to prevent clashes with *MovieGenresRow* fields (which is named *fld*), i had to assign *MovieGenresRow* an alias, *mg*.

```
var mg = Entities.MovieGenresRow.Fields.As("mg");
```

What i'm trying to achieve, is a query like this (just the way we'd do this in bare SQL):

```
SELECT t0.MovieId, t0.Title, ... FROM Movies t0
WHERE EXISTS (
    SELECT 1
    FROM MovieGenres mg
    WHERE
        mg.MovieId = t0.MovieId AND
        mg.GenreId IN (1, 3, 5, 7)
)
```

So i'm adding a WHERE filter to main query with Where method, using an EXISTS criteria:

```
query.Where(Criteria.Exists(
```

Then starting to write the subquery:

```
query.SubQuery()  
    .From(mg)  
    .Select("1")
```

And adding the where statement for subquery:

```
.Where(  
    mg.MovieId == fld.MovieId &&  
    mg.GenreId.In(Request.Genres))
```

Here fld actually contains the alias t0 for MovieRow fields.

As *Criteria.Exists* method expects a simple string, i had to use `.ToString()` at the end, to convert subquery to a string:

Yes, i should add one overload that accepts a subquery... noted.

```
.ToString());
```

It might look a bit alien at start, but by time you'll understand that Serenity query system matches SQL almost 99%. It can't be the exact SQL as we have to work in a different language, C#.

Now our filtering for *GenreList* property works perfectly...



## The Cast and Characters They Played

If we wanted to keep a record of actors and the roles they played like this:

<i>Actor/Actress</i>	<i>Character</i>
Keanu Reeves	Neo
Laurence Fishburne	Morpheus
Carrie-Anne Moss	Trinity

We need a table `MovieCast` with columns like:

<i>MovieCastId</i>	<i>MovieId</i>	<i>PersonId</i>	<i>Character</i>
...	...	...	...
11	2 (Matrix)	77 (Keanu Reeves)	Neo
12	2 (Matrix)	99 (Laurence Fisburne)	Morpheus
13	2 (Matrix)	30 (Carrie-Anne Moss)	Trinity
...	...	...	...

It's clear that we also need a `Person` table as we'll keep actors/actresses by their ID.

It's better to call it *Person* as actors/actresses might become directors, scenario writers and such later.

### Creating Person and MovieCast Tables

Now its time to create a migration with two tables:

MovieTutorial.Web/Modules/Common/Migrations/DefaultDB/  
DefaultDB\_20160528\_141600\_PersonAndMovieCast.cs:

```
using FluentMigrator;
using System;

namespace MovieTutorial.Migrations.DefaultDB
{
    [Migration(20160528141600)]
    public class DefaultDB_20160528_141600_PersonAndMovieCast : Migration
    {
        public override void Up()
        {
            Create.Table("Person").InSchema("mov")
                .WithColumn("PersonId").AsInt32().Identity()
                .PrimaryKey().NotNullable()
                .WithColumn("Firstname").AsString(50).NotNullable()
                .WithColumn("Lastname").AsString(50).NotNullable()
                .WithColumn("BirthDate").AsDateTime().Nullable()
                .WithColumn("BirthPlace").AsString(100).Nullable()
                .WithColumn("Gender").AsInt32().Nullable()
                .WithColumn("Height").AsInt32().Nullable();

            Create.Table("MovieCast").InSchema("mov")
                .WithColumn("MovieCastId").AsInt32().Identity()
                .PrimaryKey().NotNullable()
                .WithColumn("MovieId").AsInt32().NotNullable()
                .ForeignKey("FK_MovieCast_MovieId", "mov", "Movie", "MovieId")
                .WithColumn("PersonId").AsInt32().NotNullable()
                .ForeignKey("FK_MovieCast_PersonId", "mov", "Person", "PersonId")
                .WithColumn("Character").AsString(50).Nullable();
        }

        public override void Down()
        {
        }
    }
}
```

## Generating Code For Person Table

First generate code for Person table:

Serenity Code Generator

Web Project (.csproj) ..\..\..\MovieTutorial\MovieTutorial.Web\MovieTutorial.Web.csproj

Script Project (.csproj)

Root Namespace MovieTutorial

Connection String Default [Data Source=(LocalDb)\v11.0; Initial Catalog=MovieTutorial\_Default\_v1; Integrate ...] New Connection Delete

Table Name mov.Person

Module Name MovieDB sample: TaskManagement

Connection Key Default sample: Default

Entity Identifier Person sample: Task

Permission Key Administration sample: Administration

Generation Options  Grid/Dialog in TypeScript (instead of Saltaralle)  TypeScript Typings  Saltaralle Imports

```

namespace MovieTutorial.MovieDB.Entities
{
    using Newtonsoft.Json;
    using Serenity;
    using Serenity.ComponentModel;
    using Serenity.Data;
    using Serenity.Data.Mapping;
    using System;
    using System.ComponentModel;
}

```

Generate Code for Entity

## Changing Gender To Enumeration

Gender column in Person table should be an enumeration. Declare a Gender enumeration in *Gender.cs* next to *PersonRow.cs*:

```

using Serenity.ComponentModel;
using System.ComponentModel;

namespace MovieTutorial.MovieDB
{
    [EnumKey("MovieDB.Gender")]
    public enum Gender
    {
        [Description("Male")]
        Male = 1,
        [Description("Female")]
        Female = 2
    }
}

```

Change Gender property declaration in *PersonRow.cs* as below:

```
//...
[DisplayName("Gender")]
public Gender? Gender
{
    get { return (Gender?)Fields.Gender[this]; }
    set { Fields.Gender[this] = (Int32?)value; }
}
//...
```

For consistency, change type of Gender property in PersonForm.cs and PersonColumns.cs from Int32 to Gender.

## Rebuilding T4 Templates

As we declared a new enumeration and used it, we should rebuild solution, convert T4 templates

Now after launching your project, you should be able to enter actors:

The screenshot shows a web browser window displaying the 'Person' management interface. The browser address bar shows 'localhost:56549/MovieDB/Person'. The application has a purple header with the 'Movie Tutorial' logo and a user profile for 'admin'. A left sidebar contains navigation links: Dashboard, Movie Database, Northwind, Basic Samples, Theme Samples, Administration, MovieDB, and Person. The main content area is titled 'Person' and features a search bar, a '+ New Person' button, and a table with columns: ID, Firstname, Lastname, Birth Date, Birth Place, Gender, and Height. A 'New Person' modal dialog is open, showing a 'Save' button and a 'Properties' section with the following fields: Firstname (Carrie-Anne), Lastname (Moss), Birth Date (08/21/1967), Birth Place (Vancouver, British Columbia, Canada), Gender (Female), and Height (174). At the bottom of the page, there is a pagination control showing 'Page 1 / 1' and 'No records', along with a copyright notice 'Copyright (c) 2015. All rights reserved.' and the text 'Serenity Platform'.

## Declaring FullName Field

On the title of edit dialog, first name of the person is shown (*Carrie-Anne*). It would be nice to show full name. And also search with full name in grid.

So let's edit our `PersonRow.cs`:

```
namespace MovieTutorial.MovieDB.Entities
{
    //...
    public sealed class PersonRow : Row, IIdRow, INameRow
    {
        //... remove QuickSearch from FirstName
        [DisplayName("First Name"), Size(50), NotNull]
        public String Firstname
        {
            get { return Fields.Firstname[this]; }
            set { Fields.Firstname[this] = value; }
        }

        [DisplayName("Last Name"), Size(50), NotNull]
        public String Lastname
        {
            get { return Fields.Lastname[this]; }
            set { Fields.Lastname[this] = value; }
        }

        [DisplayName("Full Name"),
         Expression("(t0.Firstname + ' ' + t0.Lastname)"), QuickSearch]
        public String Fullname
        {
            get { return Fields.Fullname[this]; }
            set { Fields.Fullname[this] = value; }
        }

        //... change NameField to Fullname
        StringField INameRow.NameField
        {
            get { return Fields.Fullname; }
        }

        //...

        public class RowFields : RowFieldsBase
        {
            public readonly Int32Field PersonId;
            public readonly StringField Firstname;
            public readonly StringField Lastname;
            public readonly StringField Fullname;
            //...
        }
    }
}
```

We specified SQL expression `Expression("(t0.Firstname + ' ' + t0.Lastname)")` on top of Fullname property. Thus, it is a server side calculated field.

By adding QuickSearch attribute to FullName, instead of Firstname, grid will now search by default on Fullname field.

But dialog will still show Firstname. We need to build and transform templates to make it show Fullname.

## Why Had to Transform Templates?

This will become more clear after looking at `PersonDialog.ts` file:

```
namespace MovieTutorial.MovieDB {

    @Serenity.Decorators.registerClass()
    @Serenity.Decorators.responsive()
    export class PersonDialog extends Serenity.EntityDialog<PersonRow, any> {
        protected getFormKey() { return PersonForm.formKey; }
        protected getIdProperty() { return PersonRow.idProperty; }
        protected getLocalTextPrefix() { return PersonRow.localTextPrefix; }
        protected getNameProperty() { return PersonRow.nameProperty; }
        protected getService() { return PersonService.baseUrl; }

        protected form = new PersonForm(this.idPrefix);
    }
}
```

Here we see that `getNameProperty()` method returns `PersonRow.nameProperty`. `PersonRow` typing in TypeScript is in a file (`MovieDB.PersonRow.ts`) generated by our T4 templates.

Thus, unless we transform T4 templates, the name property change we did in `PersonRow.cs` won't be reflected in corresponding `MovieDB.PersonRow.ts` file under

`*Modules/Common/Imports/ServerTypings/ ServerTypings.tt`:

```

namespace MovieTutorial.MovieDB {
  export interface PersonRow {
    PersonId?: number;
    Firstname?: string;
    Lastname?: string;
    Fullname?: string;
    //...
  }

  export namespace PersonRow {
    export const idProperty = 'PersonId';
    export const nameProperty = 'Fullname';
    export const localTextPrefix = 'MovieDB.Person';

    export namespace Fields {
      export declare const PersonId: string;
      //...
    }
    //...
  }
}

```

This metadata (name property of PersonRow) is transferred to TypeScript with a code file (MovieDB.PersonRow.ts) that is generated by ServerTypings.tt file.

Similarly, *idProperty*, *localTextPrefix*, *Enum Types* etc. are also generated under *ServerTypings.tt*. Thus, when you make a change that affects a metadata in these generated files, you should transform T4 templates to transfer that information to TypeScript.

You should always build before transforming, as T4 files uses output DLL of MovieTutorial.Web project. Otherwise you'll be generating code for an older version of your Web project.

## Declaring PersonRow Lookup Script

While we are still here, let's declare a LookupScript for Person table in PersonRow.cs:

```

namespace MovieTutorial.MovieDB.Entities
{
  //...
  [LookupScript("MovieDB.Person")]
  public sealed class PersonRow : Row, IIdRow, INameRow
  //...
}

```

We'll use it for editing Movie cast later.

Build and transform templates again, you'll see that *MovieDB.PersonRow.ts* now has a *getLookup()* method alongside with a new *lookupKey* property:

```
namespace MovieTutorial.MovieDB {
    export interface PersonRow {
        //...
    }

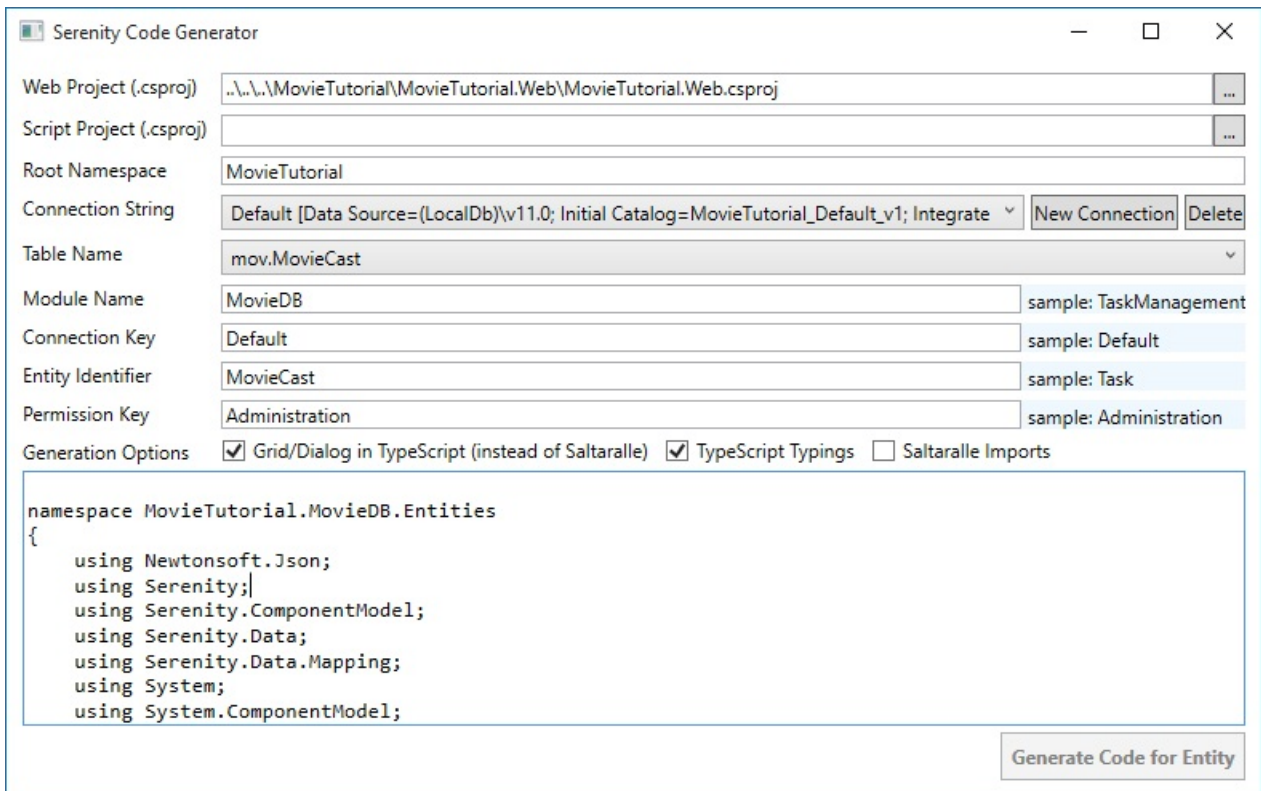
    export namespace PersonRow {
        export const idProperty = 'PersonId';
        export const nameProperty = 'Fullname';
        export const localTextPrefix = 'MovieDB.Person';
        export const lookupKey = 'MovieDB.Person';

        export function getLookup(): Q.Lookup<PersonRow> {
            return Q.getLookup<PersonRow>('MovieDB.Person');
        }
    }

    //...
}
```

## Generating Code For MovieCast Table

Generate code for MovieCast table using *sergen*:



After generating code, as we don't need a separate page to edit movie cast table, you may delete files listed below:



```
MovieCastIndex.cshtml
MovieCastPage.cs
MovieDialog.ts
MovieGrid.ts
```

Again, build and transform templates.

## Master/Detail Editing Logic For MovieCast Table

Up until now, we created a page for each table, and list and edit its records in that page. This time we are going to use a different strategy.

We'll list the cast for a movie in the Movie dialog and allow them to be edited along with the movie. Also, cast will be saved together with movie entity in one transaction.

Thus, cast editing will be in memory, and when user presses save button in Movie dialog, the movie and its cast will be saved to database in one shot (one transaction).

It would be possible to edit the cast independently, here we just want to show how it can be done.

For some types of master/detail records like order/detail, details shouldn't be allowed to be edited independently for consistency reasons. Serene already has a sample for this kind of editing in Northwind/Order dialog.

## Creating an Editor For Movie Cast List

Next to `MovieCastRow.cs` (at `MovieTutorial.Web/Modules/MovieDB/MovieCast/`), create a file named `MovieCastEditor.ts` with contents below:

```
/// <reference path="../../../Common/Helpers/GridEditorBase.ts" />

namespace MovieTutorial.MovieDB {
    @Serenity.Decorators.registerEditor()
    export class MovieCastEditor
        extends Common.GridEditorBase<MovieCastRow> {
        protected getColumnsKey() { return "MovieDB.MovieCast"; }
        protected getLocalTextPrefix() { return MovieCastRow.localTextPrefix; }

        constructor(container: JQuery) {
            super(container);
        }
    }
}
```

This editor derives from *Common.GridEditorBase* class in Serene, which is a special grid type that is designed for in-memory editing. It is also the base class for Order Details editor used in Order dialog.

The `<reference />` line at top of the file is important. TypeScript has ordering problems with input files. If we didn't put it there, TypeScript would sometimes output *GridEditorBase* after our *MovieCastEditor*, and we'd get runtime errors.

As a rule of thumb, if you are deriving some class from another in your project (not Serenity classes), you should put a reference to file containing that base class.

This helps TypeScript to convert *GridEditorBase* to javascript before other classes that might need it.

To reference this new editor type from server side, build and transform all templates.

This base class might be integrated to Serenity in later versions. In that case, its namespace will become Serenity, instead of Serene or MovieTutorial.

## Using MovieCastEditor in Movie Form

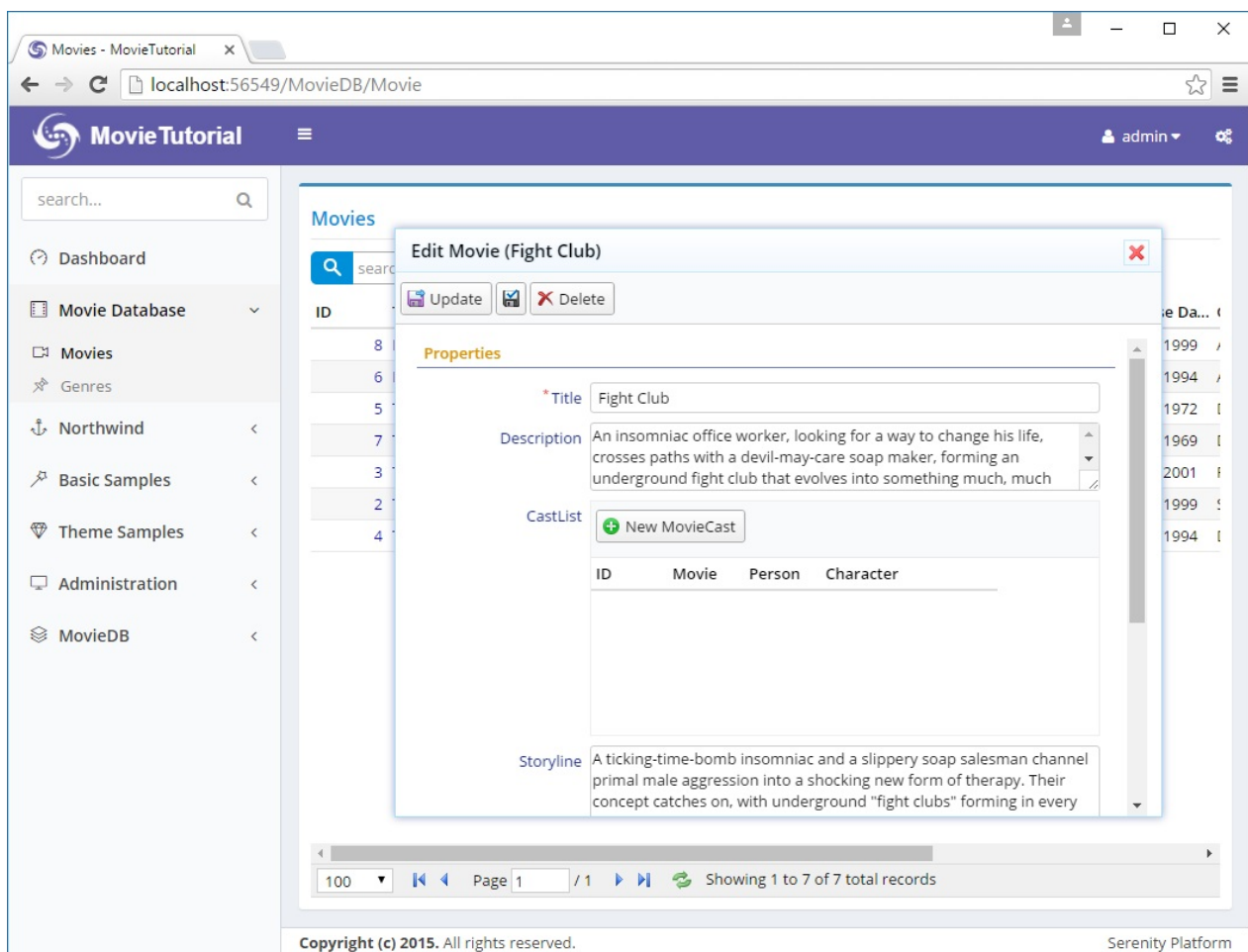
Open *MovieForm.cs*, between *Description* and *Storyline* fields, add a *CastList* property like:

```
namespace MovieTutorial.MovieDB.Forms
{
    //...
    public class MovieForm
    {
        public String Title { get; set; }
        [TextAreaEditor(Rows = 3)]
        public String Description { get; set; }
        [MovieCastEditor]
        public List<Entities.MovieCastRow> CastList { get; set; }
        [TextAreaEditor(Rows = 8)]
        public String Storyline { get; set; }
        //...
    }
}
```

By putting *[MovieCastEditor]* attribute on top of *CastList* property, we specified that this property will be edited by our new *MovieCastEditor* type which is defined in TypeScript code.

We could also write *[EditorType("MovieDB.MovieCast")]* but who really likes hard-coded strings? Not me...

Now build and launch your application. Open a movie dialog and you'll be greeted by our new editor:



OK, it looked easy, but I'll be honest, we are not even half the way.

That *New MovieCast* button doesn't work, need to define a dialog for it. The grid columns are not what I'd like them to be and the field and button titles are not so user friendly...

Also we'll have to handle a bit more plumbing like loading and saving cast list on server side (we'll show the harder - manual way first, then we'll see how easy it can be using a service behavior).

## Configuring MovieCastEditor to Use MovieCastEditDialog

Create a *MovieCastEditDialog.ts* file next to *MovieCastEditor.ts* and modify it like below:

```
/// <reference path="../../Common/Helpers/GridEditorDialog.ts" />

namespace MovieTutorial.MovieDB {

    @Serenity.Decorators.registerClass()
    export class MovieCastEditDialog extends
        Common.GridEditorDialog<MovieCastRow> {
        protected getFormKey() { return MovieCastForm.formKey; }
        protected getNameProperty() { return MovieCastRow.nameProperty; }
        protected getLocalTextPrefix() { return MovieCastRow.localTextPrefix; }

        protected form: MovieCastForm;

        constructor() {
            super();
            this.form = new MovieCastForm(this.idPrefix);
        }
    }
}
```

We are using another base class from Serene, *Common.GridEditorDialog* which is also used by *OrderDetailEditDialog*.

Open *MovieCastEditor.ts* again, add a *getDialogType* method and override *getAddButtonCaption*:

```
/// <reference path="../../Common/Helpers/GridEditorBase.ts" />

namespace MovieTutorial.MovieDB {

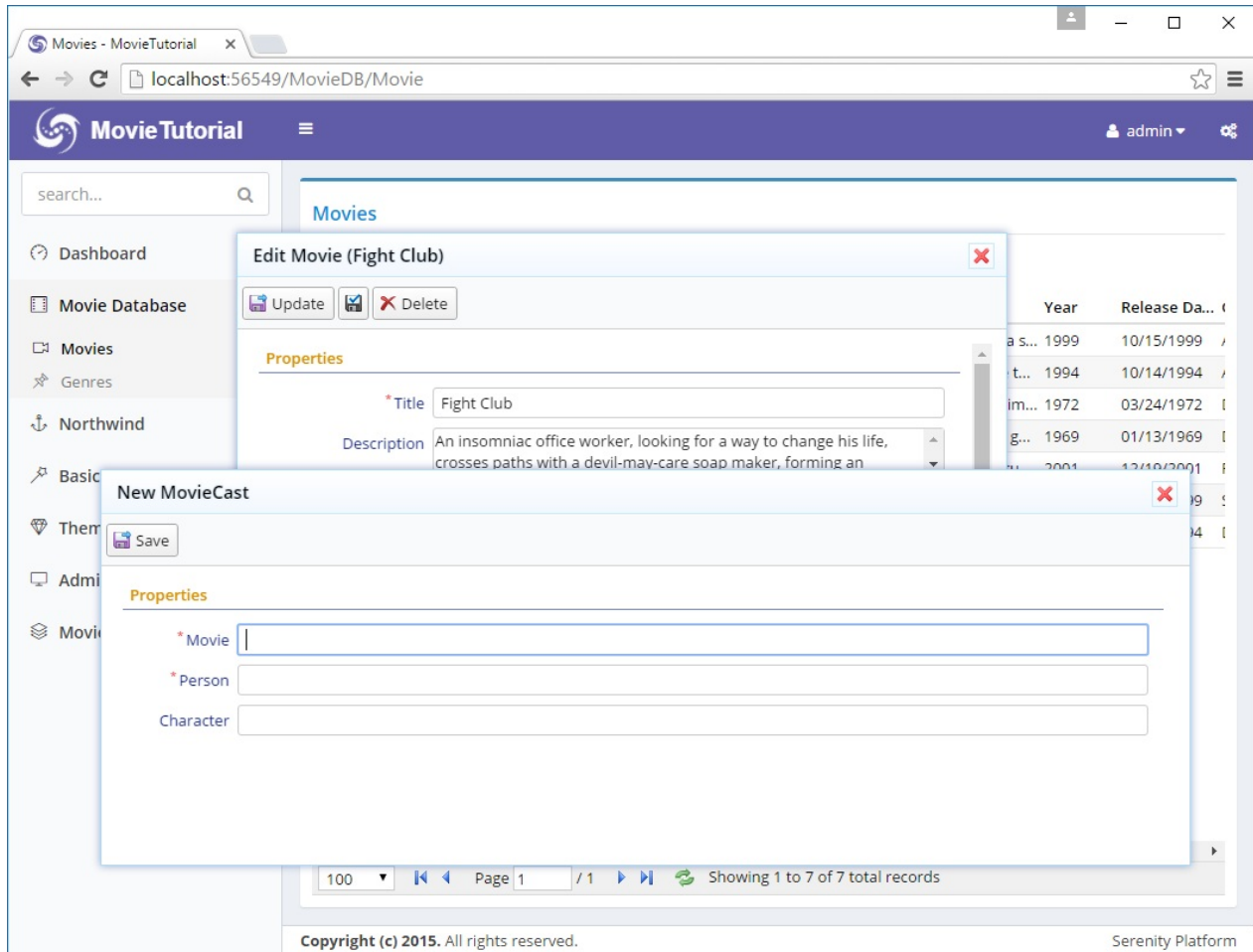
    @Serenity.Decorators.registerEditor()
    export class MovieCastEditor
        extends Common.GridEditorBase<MovieCastRow> {
        protected getColumnsKey() { return "MovieDB.MovieCast"; }
        protected getDialogType() { return MovieCastEditDialog; }
        protected getLocalTextPrefix() { return MovieCastRow.localTextPrefix; }

        constructor(container: JQuery) {
            super(container);
        }

        protected getAddButtonCaption() {
            return "Add";
        }
    }
}
```

We specified that *MovieCastEditor* uses a *MovieCastEditDialog* by default which is also used by *Add* button.

Now, instead of doing nothing, *Add* button shows a dialog.



This dialog needs some CSS formatting. Movie title and person name fields accepts integer inputs (as they are actually `MovieId` and `PersonId` fields).

## Editing `MovieCastForm.cs`

`getFormKey()` method of `MovieCastEditDialog` returns `MovieCastForm.formKey`, so it currently uses `MovieCastForm.cs` generated by Sergen.

It is possible to have multiple forms for one entity in Serenity. If i wanted to save `MovieCastForm` for some other standalone dialog, e.g. `MovieCastDialog` (which we actually deleted), i would prefer to define a new form like `MovieCastEditForm`, but this is not the case.

Open `MovieCastForm.cs` and modify it:

```

namespace MovieTutorial.MovieDB.Forms
{
    using Serenity.ComponentModel;
    using System;
    using System.ComponentModel;

    [FormScript("MovieDB.MovieCast")]
    [BasedOnRow(typeof(Entities.MovieCastRow))]
    public class MovieCastForm
    {
        public Int32 PersonId { get; set; }
        public String Character { get; set; }
    }
}

```

I have removed *MovieId* as this form is going to be used in *MovieCastEditDialog*, so *MovieCast* entities will have the *MovieId* of the movie currently being edited in the *MovieDialog* automatically. Opening *Lord of the Rings* movie and adding a cast entry for *the Matrix* would be non-sense.

Next, edit *MovieCastRow.cs*:

```

[ConnectionKey("Default"), TwoLevelCached]
[DisplayName("Movie Casts"), InstanceName("Cast")]
[ReadPermission("Administration")]
[ModifyPermission("Administration")]
public sealed class MovieCastRow : Row, IIdRow, INameRow
{
    //...
    [DisplayName("Actor/Actress"), NotNull, ForeignKey("[mov].[Person]", "PersonId
    ")]
    [LeftJoin("jPerson"), TextualField("PersonFirstname")]
    [LookupEditor(typeof(PersonRow))]
    public Int32? PersonId
    {
        get { return Fields.PersonId[this]; }
        set { Fields.PersonId[this] = value; }
    }
}

```

I have set editor type for *PersonId* field to a lookup editor and as i have already added a *LookupScript* attribute to *PersonRow*, i can reuse that information for setting the lookup key.

We could have also written `[LookupEditor("MovieDB.Person")]`

Changed *PersonId* display name to *Actor/Actress*.

Also changed *DisplayName* and *InstanceName* attributes for row to set dialog title.

Build solution, launch and now `MovieCastEditDialog` has a better editing experience. But still too big in width and height.

## Fixing the Look Of `MovieCastEditDialog`

Let's check `site.less` to understand why our `MovieCastEditDialog` is not styled.

```
.s-MovieDB-MovieCastDialog {
  > .size { width: 650px; }
  .caption { width: 150px; }
}
```

The CSS at the bottom of `site.less` is for the `MovieCastDialog`, not `MovieCastEditDialog`, because we defined this class ourselves, not with code generator.

We created a new dialog type `MovieCastEditDialog`, so now our new dialog has a CSS class of `s-MovieDB-MovieCastEditDialog`, but code generator only generated CSS rules for `s-MovieDB-MovieCastDialog`.

Serenity dialogs automatically assigns CSS classes to dialog elements, by prefixing type name with "s-". You can see this by inspecting the dialog in developer tools. `MovieCastEditDialog` has CSS classes of `s-MovieCastEditDialog` and `s-MovieDB-MovieCastEditDialog`, along with some other like `ui-dialog`.

`s-ModuleName-TypeName` CSS class helps with individual styling when two modules has a type with the same name.

As we are not gonna actually use `MovieCastDialog` (we deleted it), let's rename the one in `site.less`:

```
.s-MovieDB-MovieCastEditDialog {
  > .size { width: 450px; }
  .caption { width: 120px; }
  .s-PropertyGrid .categories { height: 120px; }
}
```

Now `MovieCastEditDialog` has a better look:

The screenshot shows a web application window titled "Edit Movie (Fight Club)". At the top, there are buttons for "Update", "Save", and "Delete". Below this is a "Properties" section. A "New Cast" dialog box is open, which also has a "Save" button and a "Properties" section. In the dialog's properties, there is a dropdown menu for "Actor/Actress" with "--select--" selected, and a text input field for "Character". The background form shows a "Storyline" field with the text: "A ticking-time-bomb insomniac and a slippery soap salesman channel primal male aggression into a shocking new form of therapy. Their concept catches on, with underground 'fight clubs' forming in every

## Fixing MovieCastEditor Columns

MovieCastEditor is currently using columns defined in *MovieCastColumns.cs* (because it returns "MovieDB.MovieCast" in *getColumnKey()* method).

We have *MovieCastId*, *MovieId*, *PersonId* (shown as Actor/Actress) and *Character* columns there. It is better to show only Actor/Actress and Character columns.

We want to show actors fullname instead of *PersonId* (integer value), so we'll declare this field in *MovieCastRow.cs* first:



```
namespace MovieTutorial.MovieDB.Entities
{
    //...
    public sealed class MovieCastRow : Row, IIdRow, INameRow
    {
        // ...

        [DisplayName("Person Firstname"), Expression("jPerson.Firstname")]
        public String PersonFirstname
        {
            get { return Fields.PersonFirstname[this]; }
            set { Fields.PersonFirstname[this] = value; }
        }

        [DisplayName("Person Lastname"), Expression("jPerson.Lastname")]
        public String PersonLastname
        {
            get { return Fields.PersonLastname[this]; }
            set { Fields.PersonLastname[this] = value; }
        }

        [DisplayName("Actor/Actress"),
         Expression("(jPerson.Firstname + ' ' + jPerson.Lastname)")]
        public String PersonFullname
        {
            get { return Fields.PersonFullname[this]; }
            set { Fields.PersonFullname[this] = value; }
        }

        // ...

        public class RowFields : RowFieldsBase
        {
            // ...
            public readonly StringField PersonFirstname;
            public readonly StringField PersonLastname;
            public readonly StringField PersonFullname;
            // ...
        }
    }
}
```

and modify `MovieCastColumns.cs`:

```

namespace MovieTutorial.MovieDB.Columns
{
    using Serenity.ComponentModel;
    using System;

    [ColumnsScript("MovieDB.MovieCast")]
    [BasedOnRow(typeof(Entities.MovieCastRow))]
    public class MovieCastColumns
    {
        [EditLink, Width(220)]
        public String PersonFullname { get; set; }
        [EditLink, Width(150)]
        public String Character { get; set; }
    }
}

```

Rebuild and cast grid has better columns:

✖
Edit Movie (The Matrix)

Update
 Delete

**Properties**

---

**Title** The Matrix

**Description** A computer hacker learns from mysterious rebels about the true nature of his reality and his role in the war against its controllers.

**CastList** + Add

Actor/Actress	Character

**Storyline** Thomas A. Anderson is a man living two lives. By day he is an average computer programmer and by night a hacker known as Neo. Neo has always questioned his reality, but the truth is far

Now try adding an actor/actress, for example, Keanu Reeves / Neo:

**Edit Movie (The Matrix)**
✖

Update
 Save
 Delete

**Properties**

---

**\*Title**

**Description**

**CastList**

+ Add

Actor/Actress	Character
	Neo

**Storyline**

Why Actor/Actress column is empty??

## Resolving Empty Actor/Actress Column Problem

Remember that we are editing in-memory. There is no service call involved here. So, grid is displaying whatever entity is sent back to it from the dialog.

When you click the save button, dialog builds an entity to save like this:

```
{
  PersonId: 7,
  Character: 'Neo'
}
```

These fields corresponds to the form fields you previously set in MovieCastForm.cs:

```
public class MovieCastForm
{
  public Int32 PersonId { get; set; }
  public String Character { get; set; }
}
```

But in grid, we are showing these columns:

```
public class MovieCastColumns
{
    public String PersonFullname { get; set; }
    public String Character { get; set; }
}
```

There is no PersonFullname field in this entity, so grid can't display its value.

We need to set PersonFullname ourself. Let's first transform T4 templates to have access to PersonFullname field that we recently added, then edit MovieCastEditor.ts:

```
/// <reference path="../../../Common/Helpers/GridEditorBase.ts" />

namespace MovieTutorial.MovieDB {
    @Serenity.Decorators.registerEditor()
    export class MovieCastEditor extends Common.GridEditorBase<MovieCastRow> {
        //...

        protected validateEntity(row: MovieCastRow, id: number) {
            if (!super.validateEntity(row, id))
                return false;

            row.PersonFullname = PersonRow.getLookup()
                .itemById[row.PersonId].Fullname;

            return true;
        }
    }
}
```

ValidateEntity is a method from our GridEditorBase class in Serene. This method is called when Save button is clicked to validate the entity, just before it is going to be added to the grid. But we are overriding it here for another purpose (to set PersonFullname field value) rather than validation.

As we saw before, our entity has PersonId and Character fields filled in. We can use the value of PersonId field to determine the person fullname.

For this, we need a dictionary that maps PersonId to their Fullname values. Fortunately, person lookup has such a dictionary. We can access the lookup for PersonRow through its *getLookup* method.

Another way to access person lookup is by *Q.getLookup('MovieDB.Person')*. The one in PersonRow is just a shortcut defined by T4 templates.

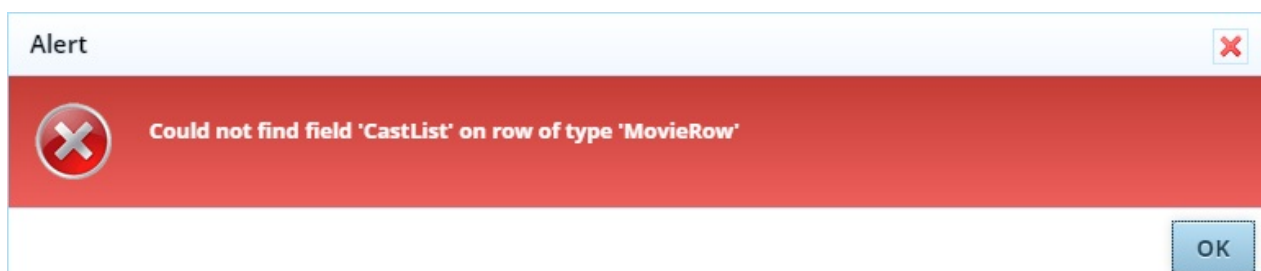
All lookups has a *itemById* dictionary that allows you to access an entity of that type by its ID.

Lookups are a simple way to share server side data with client side. But they are only suitable for small sets of data.

If a table has hundreds of thousands of records, it wouldn't be reasonable to define a lookup for it. In that case, we would use a service request to query a record by its ID.

## Defining CastList in MovieRow

While having a Movie dialog open, and at least one cast in CastList, click save button, and you'll get such an error:



This error is raised from -> Row deserializer (JsonRowConverter for JSON.NET) at server side.

We defined CastList property in MovieForm, but have no corresponding field declaration in MovieRow. So deserializer can't find where to write *CastList* value that is received from client side.

If you open developer tools with F12, click Network tab, and watch AJAX request after clicking Save button, you'll see that it has such a request payload:

```

{
  "Entity": {
    "Title": "The Matrix",
    "Description": "A computer hacker...",
    "CastList": [
      {
        "PersonId": "1",
        "Character": "Neo",
        "PersonFullname": "Keanu Reeves"
      }
    ],
    "Storyline": "Thomas A. Anderson is a man living two lives...",
    "Year": 1999,
    "ReleaseDate": "1999-03-31",
    "Runtime": 136,
    "GenreId": "",
    "Kind": "1",
    "MovieId": 1
  }
}

```

Here, CastList property can't be deserialized at server side. So we need to declare it in MovieRow.cs:

```

namespace MovieTutorial.MovieDB.Entities
{
  // ...
  public sealed class MovieRow : Row, IIdRow, INameRow
  {
    [DisplayName("Cast List"), NotMapped]
    public List<MovieCastRow> CastList
    {
      get { return Fields.CastList[this]; }
      set { Fields.CastList[this] = value; }
    }

    public class RowFields : RowFieldsBase
    {
      // ...
      public readonly RowListField<MovieCastRow> CastList;
      // ...
    }
  }
}

```

We defined a CastList property that will accept a *List* of MovieCastRow objects. The type of *Field* class that is used for such row list properties is *RowListField*.

By adding `[NotMapped]` attribute, we specified that this field is not available directly in database table, thus can't be selected through simple SQL queries. It is analogous to an unmapped field in other ORM systems.

Now, when you click the Save button, you will not get an error.

But reopen the Matrix entity you just saved. There is no cast entry there. What happened to Neo?

As this is an unmapped field, so movie Save service just ignored the CastList property.

If you remember that in prior section, our GenreList also was an unmapped field, but somehow it worked there. That's because we made use of a behavior, `LinkedListRelationBehavior` with that property.

Here we are sampling what would happen if we had no such service behavior.

## Handling Save for CastList

Open `MovieRepository.cs`, find the empty `MySaveHandler` class, and modify it like below:

```
private class MySaveHandler : SaveRequestHandler<MyRow>
{
    protected override void AfterSave()
    {
        base.AfterSave();

        if (Row.CastList != null)
        {
            var mc = Entities.MovieCastRow.Fields;
            var oldList = IsCreate ? null :
                Connection.List<Entities.MovieCastRow>(
                    mc.MovieId == this.Row.MovieId.Value);

            new Common.DetailListSaveHandler<Entities.MovieCastRow>(
                oldList, Row.CastList,
                x => x.MovieId = Row.MovieId.Value).Process(this.UnitOfWork);
        }
    }
}
```

`MySaveHandler`, processes both CREATE (insert), and UPDATE service requests for Movie rows. As most of its logic is handled by base `SaveRequestHandler` class, its class definition was empty before.

We should first wait for Movie entity to be inserted / updated successfully, before inserting / updating the cast list. Thus, we are including our customized code by overriding the base *AfterSave* method.

If this is CREATE (insert) operation, we need the *MovieId* field value to reuse in *MovieCast* records. As *MovieId* is an IDENTITY field, it is only available after inserting the movie record.

As we are editing cast list in memory (client-side), this will be a batch update.

We need to compare old list of the cast records for this movie to the new list of cast records, and INSERT/UPDATE/DELETE them.

Let's say we had cast records A, B, C, D in database for movie X.

User did some modifications in edit dialogs to cast list, and now we have A, B, D, E, F.

So we need to update A, B, D (in case character / actor changed), delete C, and insert new records E and F.

Fortunately, *DetailListSaveHandler* class that is defined in *Serene*, handles all these comparisons and performs insert/update/delete operations automatically (by ID values). Otherwise we would have to write much more code here.

To get a list of old records, we need to query database if this is an UPDATE movie operation. If this is a CREATE movie operation there shouldn't be any old cast record.

We are using *Connection.List< Entities.MovieCastRow >* extension method. *Connection* here is a property of *SaveRequestHandler* that returns the current connection used. *List* selects records that matches the specified criteria (*mc.MovieId == this.Row.MovieId.Value*). *this.Row* refers to currently inserted / updated record (movie) with its new field values, so it contains the *MovieId* value (new or existing).

To update cast records, we are creating a *DetailListHandler* object, with old cast list, new cast list, and a delegate to set the *MovieId* field value in a cast record. This is to link new cast records with the current movie.

Then we call *DetailListHandler.Process* with current unit of work. *UnitOfWork* is a special object that wraps the current connection/transaction.

All *Serene* CREATE/UPDATE/DELETE handlers works with implicit transactions (*IUnitOfWork*).

## Handling Retrieve for CastList



We are not done yet. When a *Movie* entity is clicked in movie grid, movie dialog loads the movie record by calling the movie *Retrieve* service. As *CastList* is an unmapped field, even if we saved them properly, they won't be loaded into the dialog.

We need to also edit *MyRetrieveHandler* class in *MovieRepository.cs*:

```
private class MyRetrieveHandler : RetrieveRequestHandler<MyRow>
{
    protected override void OnReturn()
    {
        base.OnReturn();

        var mc = Entities.MovieCastRow.Fields;
        Row.CastList = Connection.List<Entities.MovieCastRow>(q => q
            .SelectTableFields()
            .Select(mc.PersonFullname)
            .Where(mc.MovieId == Row.MovieId.Value));
    }
}
```

Here, we are overriding *OnReturn* method, to inject *CastList* into movie row just before returning the it from retrieve service.

I used a different overload of *Connection.List* extension, which allows me to modify the select query.

By default, *List* selects all table fields (not foreign view fields coming from other tables), but to show actor name, i needed to also select *PersonFullName* field.

Now build the solution, and we can finally list / edit the cast.

## Handling Delete for CastList

When you try to delete a *Movie* entity, you'll get foreign key errors. You could use a "CASCADE DELETE" foreign key while creating *MovieCast* table. But we'll handle this at repository level again:

```

private class MyDeleteHandler : DeleteRequestHandler<MyRow>
{
    protected override void OnBeforeDelete()
    {
        base.OnBeforeDelete();

        var mc = Entities.MovieCastRow.Fields;
        foreach (var detailID in Connection.Query<Int32>(
            new SqlQuery()
                .From(mc)
                .Select(mc.MovieCastId)
                .Where(mc.MovieId == Row.MovieId.Value)))
        {
            new DeleteRequestHandler<Entities.MovieCastRow>().Process(this.UnitOfWork,
                new DeleteRequest
                {
                    EntityId = detailID
                });
        }
    }
}

```

The way we implemented this master/detail handling is not very intuitive and included several manual steps at repository layer. Keep on reading to see how easily it could be done by using an integrated feature (*MasterDetailRelationAttribute*).

## Handling Save / Retrieve / Delete With a Behavior

Master/detail relations are an integrated feature (at least on server side), so instead of manually overriding Save / Retrieve and Delete handlers, i'll use an attribute, *MasterDetailRelation*.

Open *MovieRow.cs* and modify *CastList* property:

```

[MasterDetailRelation(foreignKey: "MovieId", IncludeColumns = "PersonFullname")]
[DisplayName("Cast List"), NotMapped]
public List<MovieCastRow> CastList
{
    get { return Fields.CastList[this]; }
    set { Fields.CastList[this] = value; }
}

```

We specified that this field is a detail list of a master/detail relation and master ID field (foreignKey) of the detail table is *MovieId*.

Now undo all changes we made in *MovieRepository.cs*:

```
private class MySaveHandler : SaveRequestHandler<MyRow> { }
private class MyDeleteHandler : DeleteRequestHandler<MyRow> { }
private class MyRetrieveHandler : RetrieveRequestHandler<MyRow> { }
```

In our *MasterDetailRelation* attribute, we specified an extra property, *IncludeColumns*:

```
[MasterDetailRelation(foreignKey: "MovieId", IncludeColumns = "PersonFullname")]
```

This ensures that *PersonFullname* field on cast list is selected on retrieve. Otherwise, it wouldn't be loaded as only table fields are selected by default. When you open a movie dialog with existing cast list, full name would be empty.

Make sure you add any view field you use in grid columns to *IncludeColumns*. Put comma between names of multiple fields, e.g. *IncludeColumns* = "FieldA, FieldB, FieldC".

Now build your project and you'll see that same functionality works with much less code.

*MasterDetailRelationAttribute* triggers an intrinsic (automatic) behavior, *MasterDetailRelationBehavior* which intercepts Retrieve/Save/Delete handlers and methods we had overridden before and performs similar operations.

So we did the same thing, but this time *declaratively*, not *imperatively* (what should be done, instead of how to do it)

[https://en.wikipedia.org/wiki/Declarative\\_programming](https://en.wikipedia.org/wiki/Declarative_programming)

We'll see how to write your own request handler behaviors in following chapters.

## Listing Movies in Person Dialog

To show list of movies a person acted in, we'll add a tab to PersonDialog.

By default all entity dialogs (ones we used so far, which derive from EntityDialog) uses *EntityDialog* template at *MovieTutorial.Web/Views/Templates/EntityDialog.Template.html*:

```
<div class="s-DialogContent">
  <div id="~_Toolbar" class="s-DialogToolbar">
  </div>
  <div class="s-Form">
    <form id="~_Form" action="">
      <div class="fieldset ui-widget ui-widget-content ui-corner-all">
        <div id="~_PropertyGrid"></div>
        <div class="clear"></div>
      </div>
    </form>
  </div>
</div>
```

This template contains a toolbar placeholder (*~\_Toolbar*), form (*~\_Form*) and PropertyGrid (*\*~\_PropertyGrid*).

*~\_* is a special prefix that is replaced with a unique dialog ID at runtime. This ensures that objects in two instances of a dialog won't have the same ID values.

EntityDialog template is shared by all dialogs, so we are not gonna modify it to add a tab to PersonDialog.

### Defining a Tabbed Template for PersonDialog

Create a new file, *MovieDB.PersonDialog.Template.html* under *Modules/MovieDB/Person/* folder with contents:

```

<div id="~_Tabs" class="s-DialogContent">
  <ul>
    <li><a href="#~_TabInfo"><span>Person</span></a></li>
    <li><a href="#~_TabMovies"><span>Movies</span></a></li>
  </ul>
  <div id="~_TabInfo" class="tab-pane s-TabInfo">
    <div id="~_Toolbar" class="s-DialogToolbar">
    </div>
    <div class="s-Form">
      <form id="~_Form" action="">
        <div class="fieldset ui-widget ui-widget-content ui-corner-all">
          <div id="~_PropertyGrid"></div>
          <div class="clear"></div>
        </div>
      </form>
    </div>
  </div>
  <div id="~_TabMovies" class="tab-pane s-TabMovies">
    <div id="~_MoviesGrid">

    </div>
  </div>
</div>

```

The syntax we used here is specific to jQuery UI tabs widget. It needs an UL element with list of tab links pointing to tab pane divs (*.tab-pane*).

When *EntityDialog* finds a div with ID `~_Tabs` in its template, it automatically initializes a tabs widget on it.

Naming of the template file is important. It must end with *.Template.html* extension. All files with this extension are made available at client side through a dynamic script.

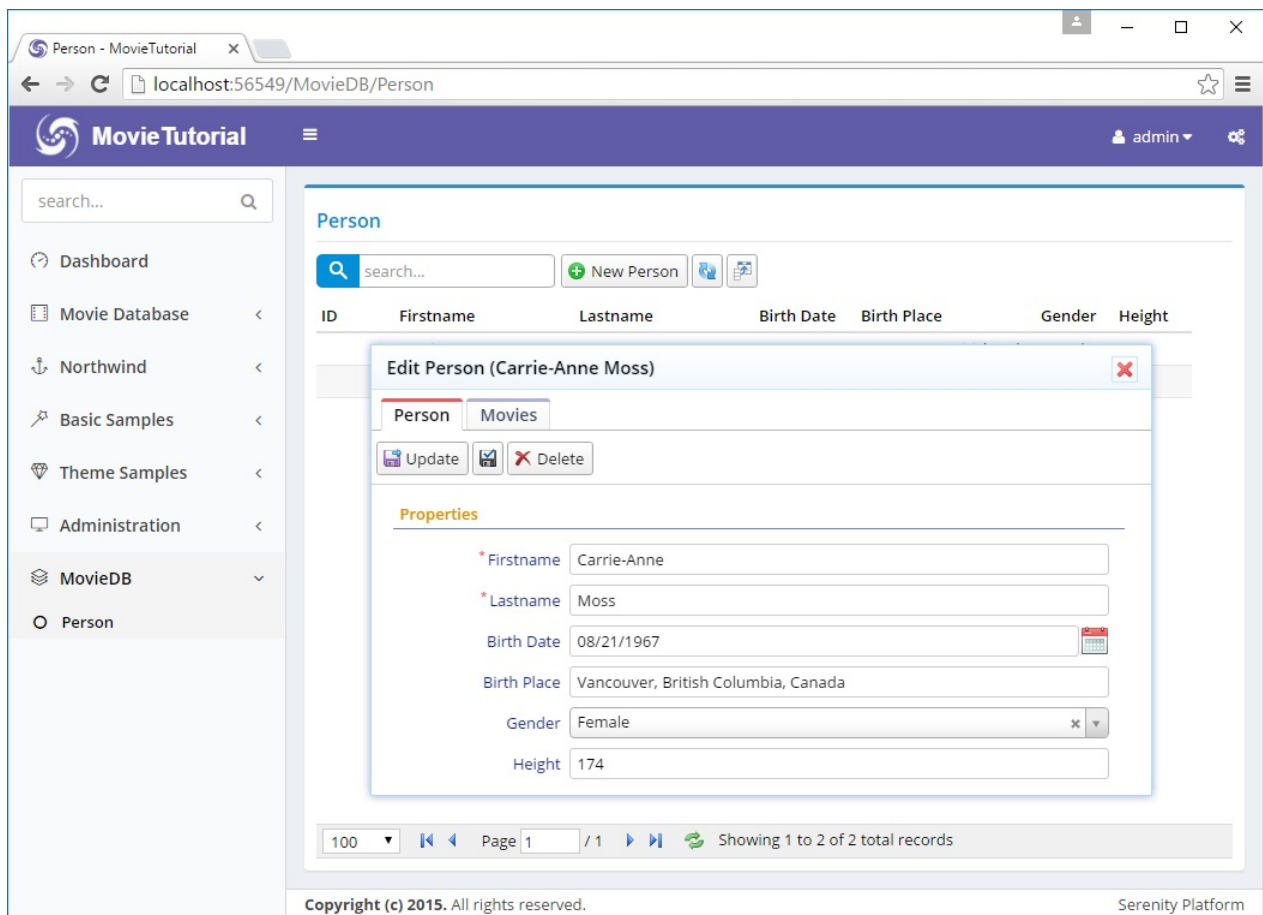
Folder of the template file is ignored, but templates must be under *Modules* or *Views/Template* directories.

By default, all templated widgets (*EntityDialog* also derives from *TemplatedWidget* class), looks for a template with their own classname. Thus, *PersonDialog* looks for a template with the name *MovieDB.PersonDialog.Template.html*, followed by *PersonDialog.Template.html*.

MovieDB comes from *PersonDialog* namespace with the root namespace (*MovieTutorial*) removed. You can also think of it as module name dot class name.

If a template with class name is not found, search continues to base classes and eventually a fallback template, *EntityDialog.Template.html* is used.

Now, we have a tab in *PersonDialog*:



Meanwhile, i noticed Person link is still under MovieDB and we forgot to remove MovieCast link. I'm fixing them now...

## Creating PersonMovieGrid

Movie tab is empty for now. We need to define a grid with suitable columns and place it in that tab.

First, declare the columns we'll use with the grid, in file *PersonMovieColumns.cs* next to *PersonColumns.cs*:

```
namespace MovieTutorial.MovieDB.Columns
{
    using Serenity.ComponentModel;
    using System;

    [ColumnsScript("MovieDB.PersonMovie")]
    [BasedOnRow(typeof(Entities.MovieCastRow))]
    public class PersonMovieColumns
    {
        [Width(220)]
        public String MovieTitle { get; set; }
        [Width(100)]
        public Int32 MovieYear { get; set; }
        [Width(200)]
        public String Character { get; set; }
    }
}
```

Next define a *PersonMovieGrid* class, in file *PersonMovieGrid.ts* next to *PersonGrid.ts*:

```
namespace MovieTutorial.MovieDB {

    @Serenity.Decorators.registerClass()
    export class PersonMovieGrid extends Serenity.EntityGrid<MovieCastRow, any>
    {
        protected getColumnsKey() { return "MovieDB.PersonMovie"; }
        protected getIdProperty() { return MovieCastRow.idProperty; }
        protected getLocalTextPrefix() { return MovieCastRow.localTextPrefix; }
        protected getService() { return MovieCastService.baseUrl; }

        constructor(container: JQuery) {
            super(container);
        }
    }
}
```

We'll actually use *MovieCast* service, to list movies a person acted in.

Last step is to instantiate this grid in *PersonDialog.ts*:

```
@Serenity.Decorators.registerClass()
@Serenity.Decorators.responsive()
export class PersonDialog extends Serenity.EntityDialog<PersonRow, any> {
    protected getFormKey() { return PersonForm.formKey; }
    protected getIdProperty() { return PersonRow.idProperty; }
    protected getLocalTextPrefix() { return PersonRow.localTextPrefix; }
    protected getNameProperty() { return PersonRow.nameProperty; }
    protected getService() { return PersonService.baseUrl; }

    protected form = new PersonForm(this.idPrefix);

    private moviesGrid: PersonMovieGrid;

    constructor() {
        super();

        this.moviesGrid = new PersonMovieGrid(this.byId("MoviesGrid"));
        this.tabs.on('tabsactivate', (e, i) => {
            this.arrange();
        });
    }
}
```

Remember that in our template we had a div with id `~_MoviesGrid` under movies tab pane. We created `PersonMovie` grid on that div.

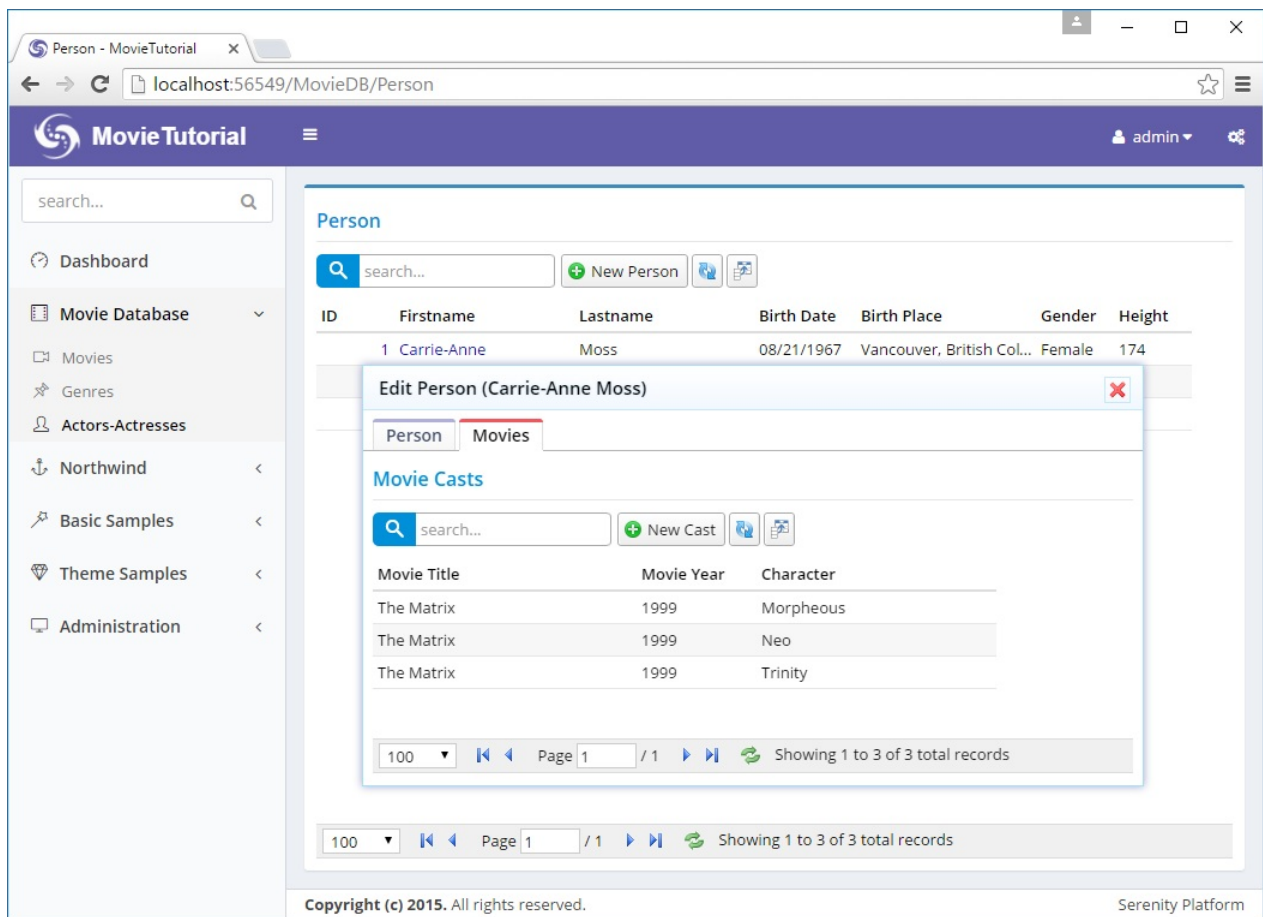
`this.ById("MoviesGrid")` is a special method for templated widgets. `$('#MoviesGrid')` wouldn't work here, as that div actually has some ID like `PersonDialog17_MoviesGrid`.

`~_` in templates are replaced with a unique container widget ID.

We also attached to `OnActivate` event of jQuery UI tabs, and called `Arrange` method of the dialog. This is to solve a problem with `SlickGrid`, when it is initially created in invisible tab. `Arrange` triggers relayout for `SlickGrid` to solve this problem.

OK, now we can see list of movies in Movies tab, but something is strange:





## Filtering Movies for the Person

No, Carrie-Anne Moss didn't act in three roles. This grid is showing all movie cast records for now, as we didn't tell what filter it should apply yet.

PersonMovieGrid should know the person it shows the movie cast records for. So, we add a *PersonID* property to this grid. This *PersonID* should be passed somehow to list service for filtering.

```
namespace MovieTutorial.MovieDB
{
    @Serenity.Decorators.registerClass()
    export class PersonMovieGrid extends Serenity.EntityGrid<MovieCastRow, any>
    {
        protected getColumnsKey() { return "MovieDB.PersonMovie"; }
        protected getIdProperty() { return MovieCastRow.idProperty; }
        protected getLocalTextPrefix() { return MovieCastRow.localTextPrefix; }
        protected getService() { return MovieCastService.baseUrl; }

        constructor(container: JQuery) {
            super(container);
        }

        protected getButtons() {
            return null;
        }

        protected getInitialTitle() {
            return null;
        }

        protected usePager() {
            return false;
        }

        protected getGridCanLoad() {
            return this.personID != null;
        }

        private _personID: number;

        get personID() {
            return this._personID;
        }

        set personID(value: number) {
            if (this._personID != value) {
                this._personID = value;
                this.setEquality(MovieCastRow.Fields.PersonId, value);
                this.refresh();
            }
        }
    }
}
```

We are using ES5 (EcmaScript 5) property (get/set) features. It's pretty similar to C# properties.

We store the person ID in a private variable. When it changes, we also set an equality filter for `PersonId` field using `SetEquality` method (which will be sent to list service), and refresh to see changes.

Equality filter is the list request parameter that is also used by quick filter items.

Overriding `GetGridCanLoad` method allows us to control when grid can call list service. If we didn't override it, while creating a new Person, grid would load all movie cast records, as there is not a `PersonID` yet (it is null).

List handler ignores an equality filter parameter if its value is null. Just like when a quick filter dropdown is empty, all records are shown.

We also did three cosmetic changes, by overriding three methods, first to remove all buttons from toolbar (`getButtons`), second to remove title from the grid (`getInitialTitle`) as tab title is enough), and third to remove paging functionality (`usePager`), a person can't have a million movies right?).

## Setting PersonID of PersonMovieGrid in PersonDialog

If nobody sets grid's `PersonID` property, it will always be null, and no records will be loaded. We should set it in `afterLoadEntity` method of Person dialog:

```
namespace MovieTutorial.MovieDB
{
    // ...
    export class PersonDialog extends Serenity.EntityDialog<PersonRow>
    {
        // ...
        protected afterLoadEntity()
        {
            super.afterLoadEntity();

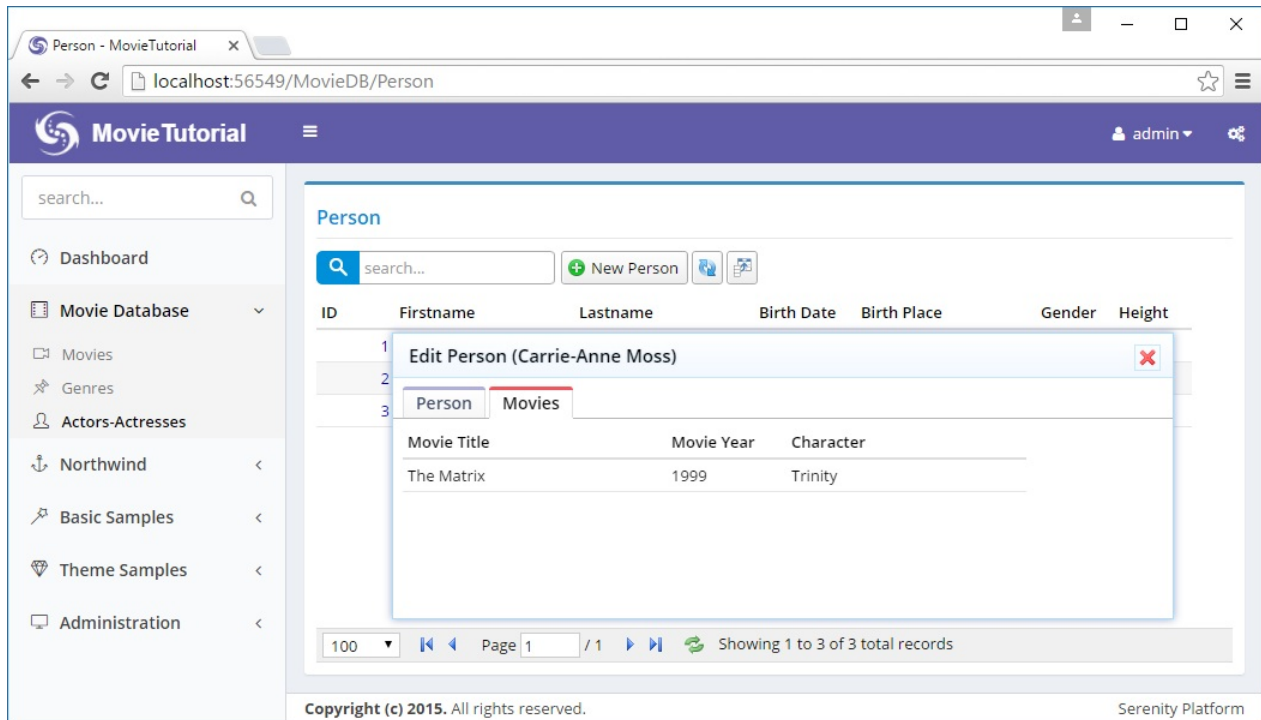
            this.moviesGrid.personID = this.entityId;
        }
    }
}
```

`afterLoadEntity` is called after an entity or a new entity is loaded into dialog.

Please note that entity is loaded in a later phase, so it won't be available in dialog constructor.

`this.EntityId` refers to the identity value of the currently loaded entity. In new record mode, it is null.

AfterLoadEntity and LoadEntity might be called several times during dialog lifetime, so avoid creating some child objects in these events, otherwise you will have multiple instances of created objects. That's why we created the grid in dialog constructor.



## Fixing Movies Tab Size

You might have noticed that when you switch to Movies tab, dialog gets a bit less in height. This is because dialog is set to auto height and grids are 200px by default. When you switch to movies tab, form gets hidden, so dialog adjusts to movies grid height.

Edit *s-MovieDB-PersonDialog* css in *site.less*:

```
.s-MovieDB-PersonDialog {
  > .size { width: 650px; }
  .caption { width: 150px; }
  .s-PersonMovieGrid > .grid-container { height: 287px; }
}
```

# Adding Primary and Gallery Images

To add a primary image and multiple gallery images to both Movie and Person records, need to start with a migration:

```
using FluentMigrator;

namespace MovieTutorial.Migrations.DefaultDB
{
    [Migration(20160603205900)]
    public class DefaultDB_20160603_205900_PersonMovieImages : Migration
    {
        public override void Up()
        {
            Alter.Table("Person").InSchema("mov")
                .AddColumn("PrimaryImage").AsString(100).Nullable()
                .AddColumn("GalleryImages").AsString(int.MaxValue).Nullable();

            Alter.Table("Movie").InSchema("mov")
                .AddColumn("PrimaryImage").AsString(100).Nullable()
                .AddColumn("GalleryImages").AsString(int.MaxValue).Nullable();
        }

        public override void Down()
        {
        }
    }
}
```

Then modify MovieRow.cs and PersonRow.cs:

```
namespace MovieTutorial.MovieDB.Entities
{
    // ...
    public sealed class PersonRow : Row, IIdRow, INameRow
    {
        [DisplayName("Primary Image"), Size(100),
         ImageUploadEditor(FileNameFormat = "Person/PrimaryImage/~")]
        public string PrimaryImage
        {
            get { return Fields.PrimaryImage[this]; }
            set { Fields.PrimaryImage[this] = value; }
        }

        [DisplayName("Gallery Images"),
         MultipleImageUploadEditor(FileNameFormat = "Person/GalleryImages/~")]
        public string GalleryImages
        {
            get { return Fields.GalleryImages[this]; }
            set { Fields.GalleryImages[this] = value; }
        }

        // ...

        public class RowFields : RowFieldsBase
        {
            // ...
            public readonly StringField PrimaryImage;
            public readonly StringField GalleryImages;
            // ...
        }
    }
}
```

```

namespace MovieTutorial.MovieDB.Entities
{
    // ...
    public sealed class MovieRow : Row, IIdRow, INameRow
    {
        [DisplayName("Primary Image"), Size(100),
         ImageUploadEditor(FileNameFormat = "Movie/PrimaryImage/~")]
        public string PrimaryImage
        {
            get { return Fields.PrimaryImage[this]; }
            set { Fields.PrimaryImage[this] = value; }
        }

        [DisplayName("Gallery Images"),
         MultipleImageUploadEditor(FileNameFormat = "Movie/GalleryImages/~")]
        public string GalleryImages
        {
            get { return Fields.GalleryImages[this]; }
            set { Fields.GalleryImages[this] = value; }
        }

        // ...
        public class RowFields : RowFieldsBase
        {
            // ...
            public readonly StringField PrimaryImage;
            public readonly StringField GalleryImages;
            // ...
        }
    }
}

```

Here we specify that these fields will be handled by *ImageUploadEditor* and *MultipleImageUploadEditor* types.

FileNameFormat specifies the naming of uploaded files. For example, Person primary image will be uploaded to a folder under *App\_Data/upload/Person/PrimaryImage/*.

You may change upload root (*App\_Data/upload*) to anything you like by modifying *UploadSettings* appSettings key in *web.config*.

~ at the end of FileNameFormat is a shortcut for the automatic naming scheme

```
{1:00000}/{0:00000000}_{2} .
```

Here, parameter {0} is replaced with identity of the record, e.g. PersonID.

Parameter {1} is identity / 1000. This is useful to limit number of files that is stored in one directory.

Parameter {2} is a unique string like *6l55nk6v2tiyi*, which is used to generate a new file name on every upload. This helps to avoid problems caused by caching on client side.

It also provides some security so file names can't be known without having a link.

Thus, a file we upload for person primary image will be located at a path like this:

```
> App_Data\upload\Person\PrimaryImage\00000\000000001_6l55nk6v2tiyi.jpg
```

You don't have to follow this naming scheme. You can specify your own format like

```
PersonPrimaryImage_{0}_{2} .
```

Next step is to add these fields to forms (MovieForm.cs and PersonForm.cs):

```
namespace MovieTutorial.MovieDB.Forms
{
    //...
    public class PersonForm
    {
        public String Firstname { get; set; }
        public String Lastname { get; set; }
        public String PrimaryImage { get; set; }
        public String GalleryImages { get; set; }
        public DateTime BirthDate { get; set; }
        public String BirthPlace { get; set; }
        public Gender Gender { get; set; }
        public Int32 Height { get; set; }
    }
}
```

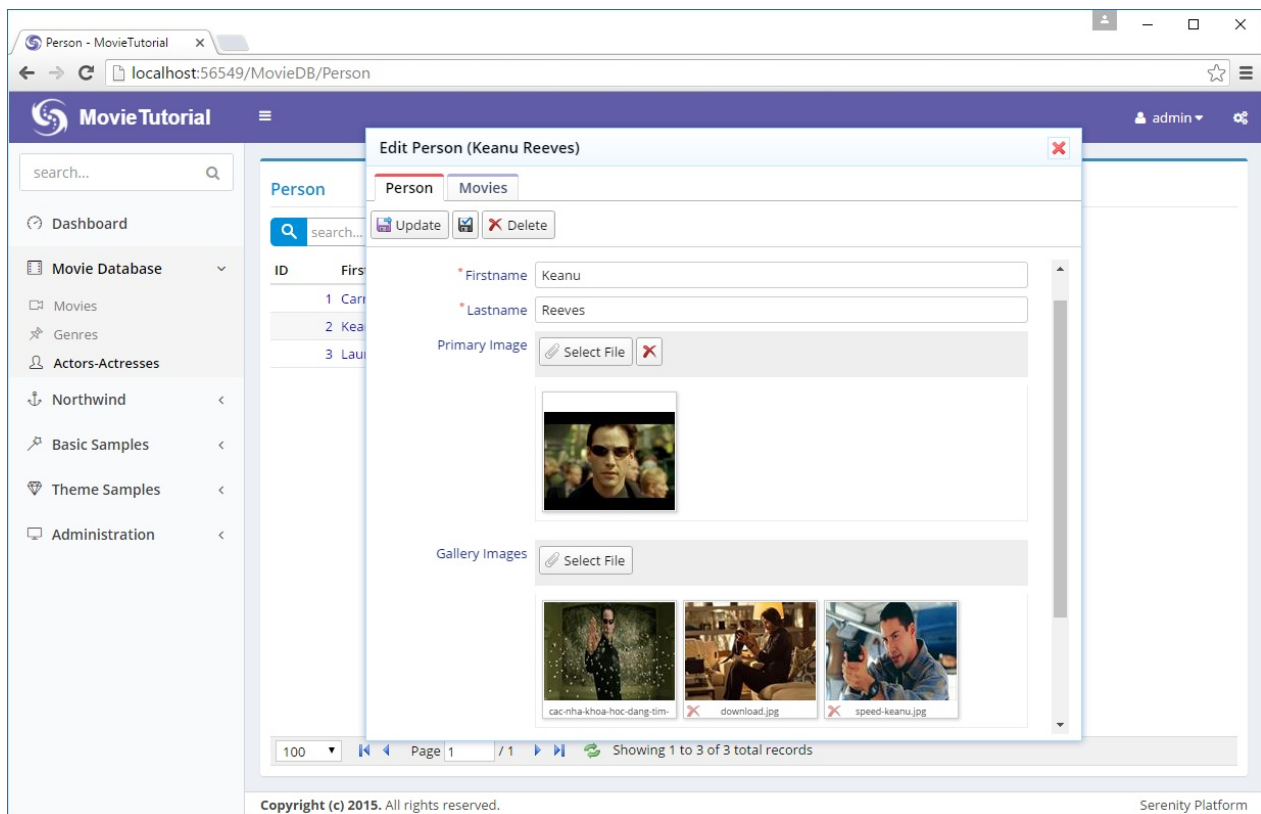


```
namespace MovieTutorial.MovieDB.Forms
{
    //...
    public class MovieForm
    {
        public String Title { get; set; }
        [TextAreaEditor(Rows = 3)]
        public String Description { get; set; }
        [MovieCastEditor]
        public List<Entities.MovieCastRow> CastList { get; set; }
        public String PrimaryImage { get; set; }
        public String GalleryImages { get; set; }
        [TextAreaEditor(Rows = 8)]
        public String Storyline { get; set; }
        public Int32 Year { get; set; }
        public DateTime ReleaseDate { get; set; }
        public Int32 Runtime { get; set; }
        public Int32 GenreId { get; set; }
        public MovieKind Kind { get; set; }
    }
}
```

I also modified Person dialog css a bit to have more space:

```
.s-MovieDB-PersonDialog {
    > .size { width: 700px; height: 600px; }
    .caption { width: 150px; }
    .s-PersonMovieGrid > .grid-container { height: 500px; }
}
```

This is what we get now:



ImageUploadEditor stores file name directly in a string field, while MultipleImageUpload editor stores file names in a string field with JSON array format.

## Removing Northwind and Other Samples

As i think our project has reached a good state, i'm now going to remove Northwind and other samples from MovieTutorial project.

See following how-to topic:

[How To: Removing Northwind and Other Samples](#)

# Multi Tenancy

In this tutorial we are going to turn Norhwind into a multi-tenant application.

Here is a definition of multi-tenant software from Wikipedia:

Software Multitenancy refers to a software architecture in which a single instance of a software runs on a server and serves multiple tenants. A tenant is a group of users who share a common access with specific privileges to the software instance. With a multitenant architecture, a software application is designed to provide every tenant a dedicated share of the instance including its data, configuration, user management, tenant individual functionality and non-functional properties. Multitenancy contrasts with multi-instance architectures, where separate software instances operate on behalf of different tenants. ---Wikipedia

We'll add a TenantId field to every table, including Users, and let user see and modify only records belonging to her tenant. So, tenants will work in isolation, as if they are working with their own database.

Multi tenant applications has some advantages like reduced cost of management. But they also have some disadvantages. For example, as all tenant data is in a single database, a tenant can't simply take or backup her data alone. Performance is usually reduced as there are more records to handle.

With increasing trend of cloud applications, decreased cost of virtualization, and with features like migration, its now easier to setup multi-instance apps.

I'd personally avoid multi-tenant applications. It's better to have one database per customer in my opinion.

But some users asked about how to implement this feature. This tutorial will help us explain some advanced Serenity topics as a bonus, along with multi tenancy.

You can find source code for this tutorial at:

<https://github.com/volkanceylan/MultiTenancy>

## Create a new project named *MultiTenancy*

In Visual Studio click File -> New Project. Make sure you choose *Serene* template. Type *MultiTenancy* as name and click *OK*.

In Solution explorer, you should see a project with name *MultiTenancy.Web*.



## Adding Tenants Table and TenantId Field

We need to add a TenantId field to all tables, to isolate tenants from each other.

So, we first need a Tenants table.

As Northwind tables already have records, we'll define a primary tenant with ID 1, and set all existing records TenantId to it.

It's time to write a migration, actually two migrations, one for Northwind and one for Default database.

**DefaultDB\_20170430\_134800\_MultiTenant.cs:**

```
using FluentMigrator;

namespace MultiTenancy.Migrations.DefaultDB
{
    [Migration(20170430134800)]
    public class DefaultDB_20170430_134800_MultiTenant
        : AutoReversingMigration
    {
        public override void Up()
        {
            this.CreateTableWithId32("Tenants", "TenantId", s => s
                .WithColumn("TenantName").AsString(100)
                .NotNullable());

            Insert.IntoTable("Tenants")
                .Row(new
                {
                    TenantName = "Primary Tenant"
                });

            Insert.IntoTable("Tenants")
                .Row(new
                {
                    TenantName = "Second Tenant"
                });

            Insert.IntoTable("Tenants")
                .Row(new
                {
                    TenantName = "Third Tenant"
                });

            Alter.Table("Users")
                .AddColumn("TenantId").AsInt32()
                .NotNullable().WithDefaultValue(1);

            Alter.Table("Roles")
                .AddColumn("TenantId").AsInt32()
                .NotNullable().WithDefaultValue(1);

            Alter.Table("Languages")
                .AddColumn("TenantId").AsInt32()
                .NotNullable().WithDefaultValue(1);
        }
    }
}
```

I have created Tenants table in Default database where user tables are. Here we add 3 predefined tenants. We actually only need first one with ID 1.

We didn't add TenantId column to tables like UserPermissions, UserRoles, RolePermissions etc, as they intrinsically have TenantId information through their UserId or RoleId (as these tables already have TenantId value)

Let's write another migration for Northwind database to add TenantId column to required tables:

**NorthwindDB\_20160110\_093500\_MultiTenant.cs:**

```
using FluentMigrator;

namespace MultiTenancy.Migrations.NorthwindDB
{
    [Migration(20170430194100)]
    public class NorthwindDB_20170430_194100_MultiTenant
        : AutoReversingMigration
    {
        public override void Up()
        {
            Alter.Table("Employees")
                .AddColumn("TenantId").AsInt32()
                .NotNullable().WithDefaultValue(1);

            Alter.Table("Categories")
                .AddColumn("TenantId").AsInt32()
                .NotNullable().WithDefaultValue(1);

            Alter.Table("Customers")
                .AddColumn("TenantId").AsInt32()
                .NotNullable().WithDefaultValue(1);

            Alter.Table("Shippers")
                .AddColumn("TenantId").AsInt32()
                .NotNullable().WithDefaultValue(1);

            Alter.Table("Suppliers")
                .AddColumn("TenantId").AsInt32()
                .NotNullable().WithDefaultValue(1);

            Alter.Table("Orders")
                .AddColumn("TenantId").AsInt32()
                .NotNullable().WithDefaultValue(1);

            Alter.Table("Products")
                .AddColumn("TenantId").AsInt32()
                .NotNullable().WithDefaultValue(1);

            Alter.Table("Region")
                .AddColumn("TenantId").AsInt32()
                .NotNullable().WithDefaultValue(1);

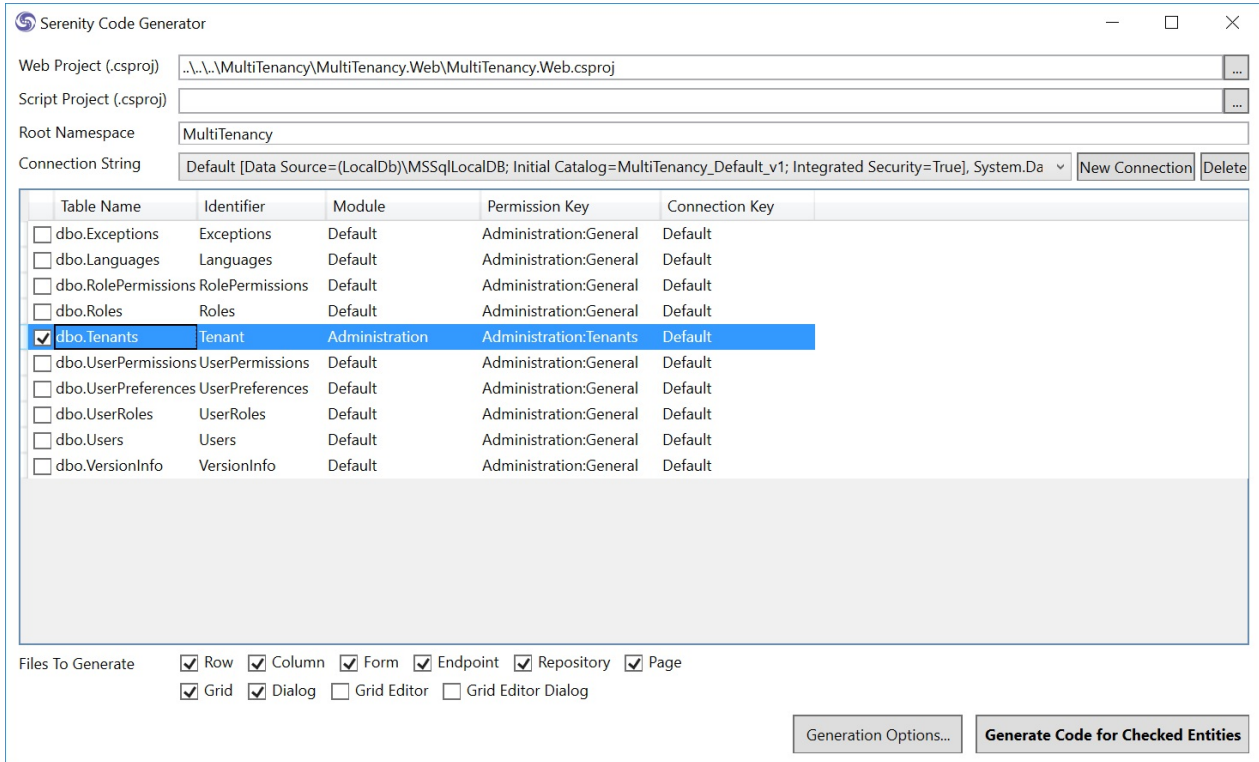
            Alter.Table("Territories")
                .AddColumn("TenantId").AsInt32()
                .NotNullable().WithDefaultValue(1);
        }
    }
}
```





# Generating Code for Tenants Table

Launch *Sergen* and generate code for *Tenants* table in *Default* connection:



Next we'll define a lookup script in *TenantRow* and set *DisplayName* property to *Tenants*:

```
namespace MultiTenancy.Administration.Entities
{
    //...
    [ConnectionString("Default"), DisplayName("Tenants"),
        InstanceName("Tenant"), TwoLevelCached]
    [LookupScript("Administration.Tenant")]
    public sealed class TenantRow : Row, IIdRow, INameRow
    {
        [DisplayName("Tenant Id"), Identity]
        public Int32? TenantId
        {
            get { return Fields.TenantId[this]; }
            set { Fields.TenantId[this] = value; }
        }
    }

    //...
}
```

Let's define a *Administration:Tenants* permission that only *admin* user will have (in *AdministrationPermissionKeys.cs*):

```
namespace MultiTenancy.Administration
{
    public class PermissionKeys
    {
        public const string Security = "Administration:Security";
        public const string Translation = "Administration:Translation";
        public const string Tenants = "Administration:Tenants";
    }
}
```

And use it on TenantRow:

```
[ConnectionKey("Default"), DisplayNahme("Tenants"),
 InstanceName("Tenant"), TwoLevelCached]
[ReadPermission(PermissionKeys.Tenants)]
[ModifyPermission(PermissionKeys.Tenants)]
[LookupScript("Administration.Tenant")]
public sealed class TenantRow : Row, IIdRow, INameRow
{
```

## Tenant Selection in User Dialog

We added a *TenantId* field to *Users* table, but it's not defined in *UserRow*, and not visible in user dialog.

This field, should only be seen and edited by *admin* user. Other users, even if we give them access to users page to manage their tenant users, shouldn't be able to see or change this information.

Let's first add it to *UserRow.cs*:

```
namespace MultiTenancy.Administration.Entities
{
    //...
    public sealed class UserRow : LoggingRow, IIdRow, INameRow
    {
        //...
        [DisplayName("Last Directory Update"), Insertable(false), Updatable(false)]
        public DateTime? LastDirectoryUpdate
        {
            get { return Fields.LastDirectoryUpdate[this]; }
            set { Fields.LastDirectoryUpdate[this] = value; }
        }

        [DisplayName("Tenant"), ForeignKey("Tenants", "TenantId"), LeftJoin("tnt")]
        [LookupEditor(typeof(TenantRow))]
        public Int32? TenantId
        {
            get { return Fields.TenantId[this]; }
            set { Fields.TenantId[this] = value; }
        }

        [DisplayName("Tenant"), Expression("tnt.TenantName")]
        public String TenantName
        {
            get { return Fields.TenantName[this]; }
            set { Fields.TenantName[this] = value; }
        }

        //...
        public class RowFields : LoggingRowFields
        {
            //...
            public readonly DateTimeField LastDirectoryUpdate;
            public readonly Int32Field TenantId;
            public readonly StringField TenantName;
            //...
        }
    }
}
```

To edit it, we need to add it to *UserForm.cs*:

```

namespace MultiTenancy.Administration.Forms
{
    using Serenity;
    using Serenity.ComponentModel;
    using System;
    using System.ComponentModel;

    [FormScript("Administration.User")]
    [BasedOnRow(typeof(Entities.UserRow))]
    public class UserForm
    {
        public String Username { get; set; }
        public String DisplayName { get; set; }
        [EmailEditor]
        public String Email { get; set; }
        [PasswordEditor]
        public String Password { get; set; }
        [PasswordEditor, OneWay]
        public String PasswordConfirm { get; set; }
        [OneWay]
        public string Source { get; set; }
        public Int32? TenantId { get; set; }
    }
}

```

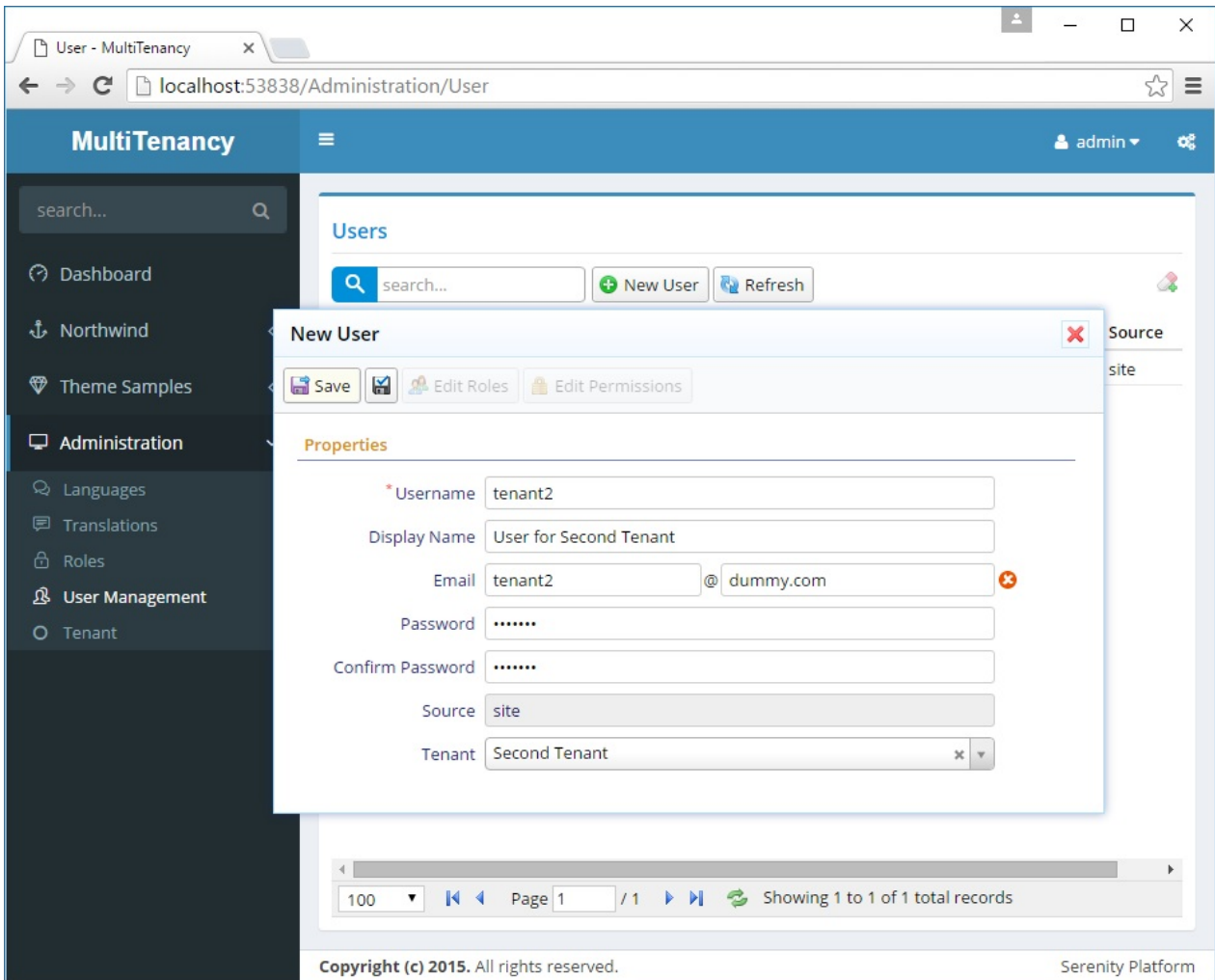
Need to also increase size of user dialog a bit, in *site.administration.less* to make space for tenant selection:

```

.s-Administration-UserDialog {
    > .size { width: 650px; }
    .caption { width: 150px; }
    .s-PropertyGrid .categories { height: 470px; }
}

```

Now open *User Management* page and create a user *tenant2* that belongs to *Second Tenant*.



After creating this user, edit its permissions and grant him *User, Role Management and Permissions* permission as this will be our administrative user for *Second Tenant*.

## Logging In With Tenant2

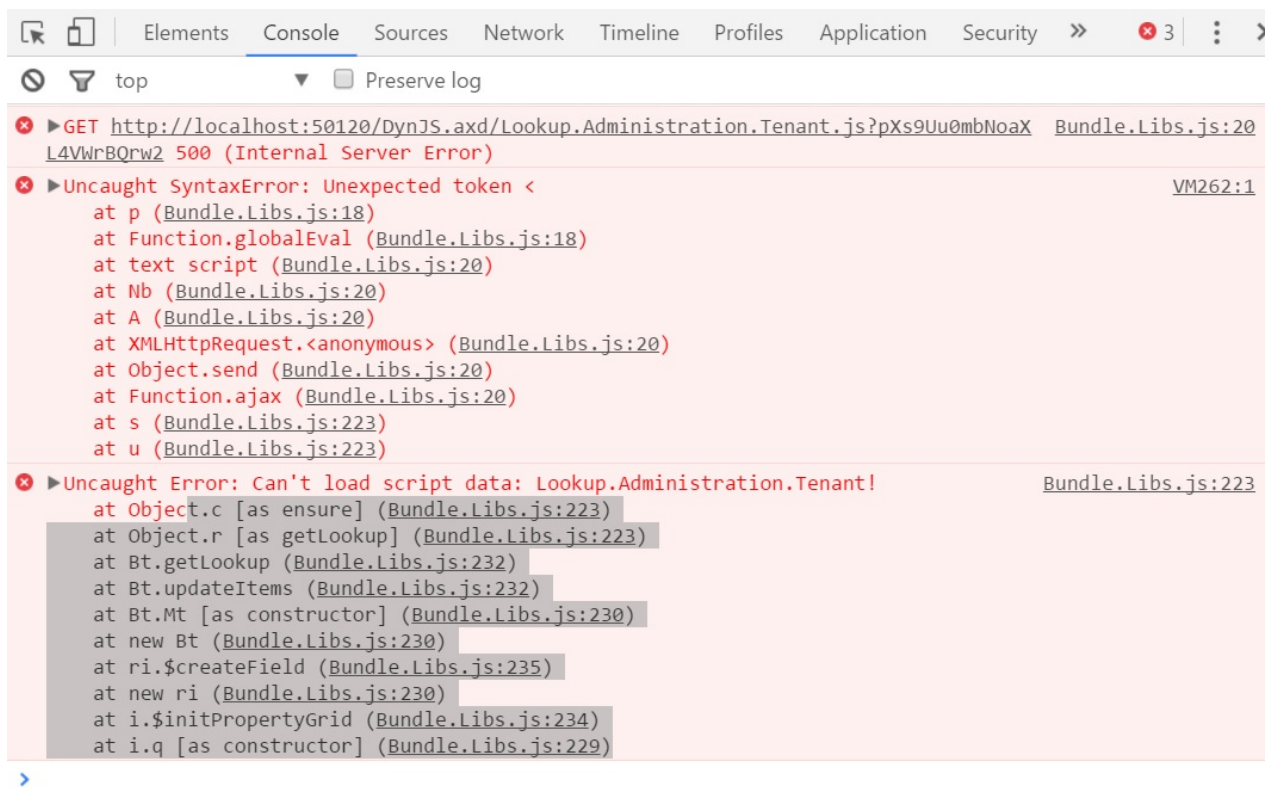
Signout and login with user *tenant2*.

When you open *User Management* page, there may be two different cases you may experience.

In first case, *tenant2* might be able to open user dialog and change his and any other users tenant. This happens if your browser cached the *tenant* lookup.

In the second case, you'll see that *tenant2* can't open User dialog. When you click a user nothing happens.

If you check browser console (whenever such a thing occurs, you should first check browser console for errors), you'll see an error like this:



This is because, our TenantRow has *Administration:Tenants* read permission which is inherited by lookup script.

We could change read permission for tenant lookup script to something else to resolve this error, but in that case *Tenant2* would be able to see and change tenant of himself and any other user including *admin*.

This is not what we wanted.

Let's first prevent him seeing users of other tenants.



## Filtering Users By TenantId

We first need to load and cache user tenant information in `UserDefinition`.

Open `UserDefinition.cs` under `Multitenancy.Web/ Modules/ Administration/ User/ Authentication` and add a `TenantId` property.

```
namespace MultiTenancy.Administration
{
    using Serenity;
    using System;

    [Serializable]
    public class UserDefinition : IUserDefinition
    {
        public string Id { get { return UserId.ToInvariant(); } }
        public string DisplayName { get; set; }
        public string Email { get; set; }
        public short IsActive { get; set; }
        public int UserId { get; set; }
        public string Username { get; set; }
        public string PasswordHash { get; set; }
        public string PasswordSalt { get; set; }
        public string Source { get; set; }
        public DateTime? UpdateDate { get; set; }
        public DateTime? LastDirectoryUpdate { get; set; }
        public int TenantId { get; set; }
    }
}
```

This is the class that is returned when you ask for current user through `Authorization.UserDefinition`.

We also need to modify the code where this class is loaded. In the same folder, edit `UserRetrieveService.cs` and change `GetFirst` method like below:

```
private UserDefinition GetFirst(IDbConnection connection, BaseCriteria criteria)
{
    var user = connection.TrySingle<Entities.UserRow>(criteria);
    if (user != null)
        return new UserDefinition
        {
            UserId = user.UserId.Value,
            Username = user.Username,
            Email = user.Email,
            DisplayName = user.DisplayName,
            IsActive = user.IsActive.Value,
            Source = user.Source,
            PasswordHash = user.PasswordHash,
            PasswordSalt = user.PasswordSalt,
            UpdateDate = user.UpdateDate,
            LastDirectoryUpdate = user.LastDirectoryUpdate,
            TenantId = user.TenantId.Value
        };
    return null;
}
```

Now, it's time to filter listed users by *TenantId*. Open *UserRepository.cs*, locate *MyListHandler* class and modify it like this:

```
private class MyListHandler : ListRequestHandler<MyRow>
{
    protected override void ApplyFilters(SqlQuery query)
    {
        base.ApplyFilters(query);

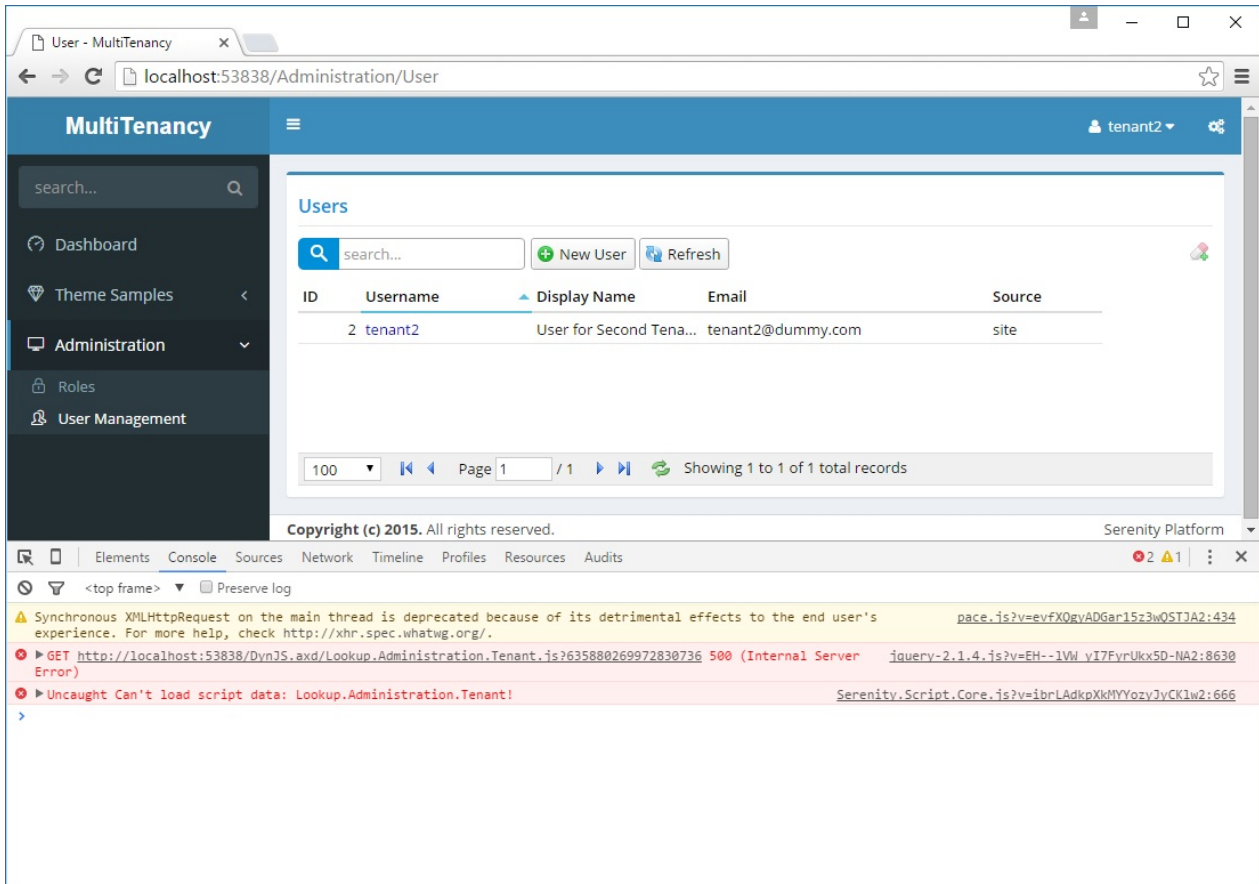
        var user = (UserDefinition)Authorization.UserDefinition;
        if (!Authorization.HasPermission(PermissionKeys.Tenants))
            query.Where(fld.TenantId == user.TenantId);
    }
}
```

Here, we first get a reference to cached user definition of currently logged user.

We check if he has tenant administration permission, which only *admin* will have in the end. If not, we filter listed records by *TenantId*.

# Removing Tenant Dropdown From User Form

After you rebuild, and launch, now user page will be like this:



Yes, he can't see admin user anymore, but something is wrong. When you click *tenant2*, nothing will happen and you'll get an error "Can't load script data: *Lookup.Administration.Tenant*" in browser console:

This error is not related to our recent filtering at repository level. It can't load this lookup script, because current user has no permission to *Tenants* table. But how did he see it last time (in one case)?

He could see it, because we first logged in as *admin* and when we open edit dialog for user, we loaded this lookup script. Browser cached it, so when we logged in with *tenant2* and open edit dialog, it loaded tenants from browser cache.

But this time, as we rebuild project, browser tried to load it from server, and we got this error, as *tenant2* doesn't have this permission. It's ok, we don't want him to have this permission, but how to avoid this error?

We need to remove *Tenant* field from the user form. But we need that field for *admin* user, so we can't simply delete it from *UserForm.cs*. Thus, we need to do it conditionally.

Build the project, transform all templates and add method below to *UserDialog.ts*:

```
protected getPropertyItems() {
    var items = super.getPropertyItems();
    if (!Q.Authorization.hasPermission("Administration:Tenants"))
        items = items.filter(x => x.name != UserRow.Fields.TenantId);
    return items;
}
```

Dialogs gets list of fields it will show in its form by *getPropertyItems* method, which in turn loads them from server side form definition.

Here we exclude *TenantId* field, if current user doesn't have the tenants permission.

This doesn't modify the original user form, it just changes list for this dialog instance.

User *tenant2* can now open the user dialog.

# Securing Tenant Selection At Server Side

When you log in with *tenant2* user and open its edit form, *Tenant* selection dropdown is not displayed, so he can't change his *tenant* right?

Wrong!

If he is an ordinary user, he can't. But if he has some knowledge of how Serenity and its services work, he could.

When you are working with web, you got to take security much more seriously.

It's very easy to create security holes in web applications unless you handle validations both at client side and server side.

Let's demonstrate it. Open Chrome console, while logged in with user *tenant2*.

Copy and paste this into console:

```
Q.serviceCall({
  service: 'Administration/User/Update',
  request: {
    EntityId: 2,
    Entity: {
      UserId: 2,
      TenantId: 1
    }
  }
});
```

Now refresh the user management page, you'll see that *tenant2* can see admin user now!

We called *User Update* service with javascript, and changed *tenant2* user *TenaNntId* to 1 (*Primary Tenant*).

Let's revert it back to *Second Tenant (2)* first, then we'll fix this security hole:

```
Q.serviceCall({
  service: 'Administration/User/Update',
  request: {
    EntityId: 2,
    Entity: {
      UserId: 2,
      TenantId: 2
    }
  }
});
```

Luckily, Serenity provides field level permissions. Edit *UserRow.cs* to let only users with *Administration:Tenants* permission to see and edit tenant information.

```
[LookupEditor(typeof(TenantRow))]
[ReadPermission(PermissionKeys.Tenants)]
public Int32? TenantId
{
  get { return Fields.TenantId[this]; }
  set { Fields.TenantId[this] = value; }
}
```

Now only *admin* can see and update *tenant* field for users.

We didn't have to also set *ModifyPermission* as if a user doesn't have the read permission, he doesn't have the write permission by default.

Build your project, then try typing this into console again:

```
Q.serviceCall({
  service: 'Administration/User/Update',
  request: {
    EntityId: 2,
    Entity: {
      UserId: 2,
      TenantId: 1
    }
  }
});
```

You will now get this error:

```
Tenant field is read only!
```



## Setting TenantId For New Users

While logged in with Tenant2, try to create a new user, *User2*.

You won't get any error but by surprise, you won't see the newly created user in list. What happened to *User2*?

As we set default value for *TenantId* to 1 in migrations, now *User2* has 1 as *TenantId* and is a member of *Primary Tenant*.

We have to set new users *TenantId* to same value with logged in user.

Modify *SetInternalFields* method of *UserRepository* like below:

```
protected override void SetInternalFields()
{
    base.SetInternalFields();

    if (IsCreate)
    {
        Row.Source = "site";
        Row.IsActive = Row.IsActive ?? 1;
        if (!Authorization.HasPermission(Administration.PermissionKeys.Tenants) ||
            Row.TenantId == null)
        {
            Row.TenantId = ((UserDefinition)Authorization.UserDefinition)
                .TenantId;
        }
    }

    if (IsCreate || !Row.Password.IsEmptyOrNull())
    {
        string salt = null;
        Row.PasswordHash = GenerateHash(password, ref salt);
        Row.PasswordSalt = salt;
    }
}
```

Here, we set *TenantId* to the same value with current user, unless he has tenant administration permission.

Now try to create a new user *User2b* and this time you'll see him on the list.



# Preventing Edits To Users From Other Tenants

Remember that user *tenant2* could update his *TenantId* with some service call, and we had to secure it server side.

Similar to this, even if he can't see users from other tenants by default, he can actually retrieve and update them.

Time to hack again.

Open Chrome console and type this:

```
new MultiTenancy.Administration.UserDialog().loadByIdAndOpenDialog(1)
```

What? He could open user dialog for *admin* and update it!

*MultiTenancy.Administration.UserDialog* is the dialog class that is opened when you click a username in user administration page.

We created a new instance of it, and asked to load a user entity by its ID. Admin user has an ID of 1.

So, to load the entity with ID 1, dialog called *Retrieve* service of *UserRepository*.

Remember that we did filtering in *List* method of *UserRepository*, not *Retrieve*. So, service has no idea, if it should return this record from another tenant, or not.

It's time to secure retrieve service in *UserRepository*:

```
private class MyRetrieveHandler : RetrieveRequestHandler<MyRow>
{
    protected override void PrepareQuery(SqlQuery query)
    {
        base.PrepareQuery(query);

        var user = (UserDefinition)Authorization.UserDefinition;
        if (!Authorization.HasPermission(PermissionKeys.Tenants))
            query.Where(fld.TenantId == user.TenantId);
    }
}
```

We did same changes in *MyListHandler* before.

If you try same Javascript code now, you'll get an error:

```
Record not found. It might be deleted or you don't have required permissions!
```

But, we could still update record calling `update` service manually. So, need to secure *MySaveHandler* too.

Change its *ValidateRequest* method like this:

```
protected override void ValidateRequest()
{
    base.ValidateRequest();

    if (IsUpdate)
    {
        var user = (UserDefinition)Authorization.UserDefinition;
        if (Old.TenantId != user.TenantId)
            Authorization.ValidatePermission(PermissionKeys.Tenants);

        // ...
    }
}
```

Here we check if it's an update, and if *TenantId* of record being updated (*Old.TenantId*) is different than currently logged user's *TenantId*. If so, we call *Authorization.ValidatePermission* method to ensure that user has tenant administration permission. It will raise an error if not.

```
Authorization has been denied for this request!
```

## Preventing To Delete Users From Other Tenants

There are delete and undelete handlers in *UserRepository*, and they suffer from similar security holes.

Using similar methods, we need to secure them too:

```
private class MyDeleteHandler : DeleteRequestHandler<MyRow>
{
    protected override void ValidateRequest()
    {
        base.ValidateRequest();

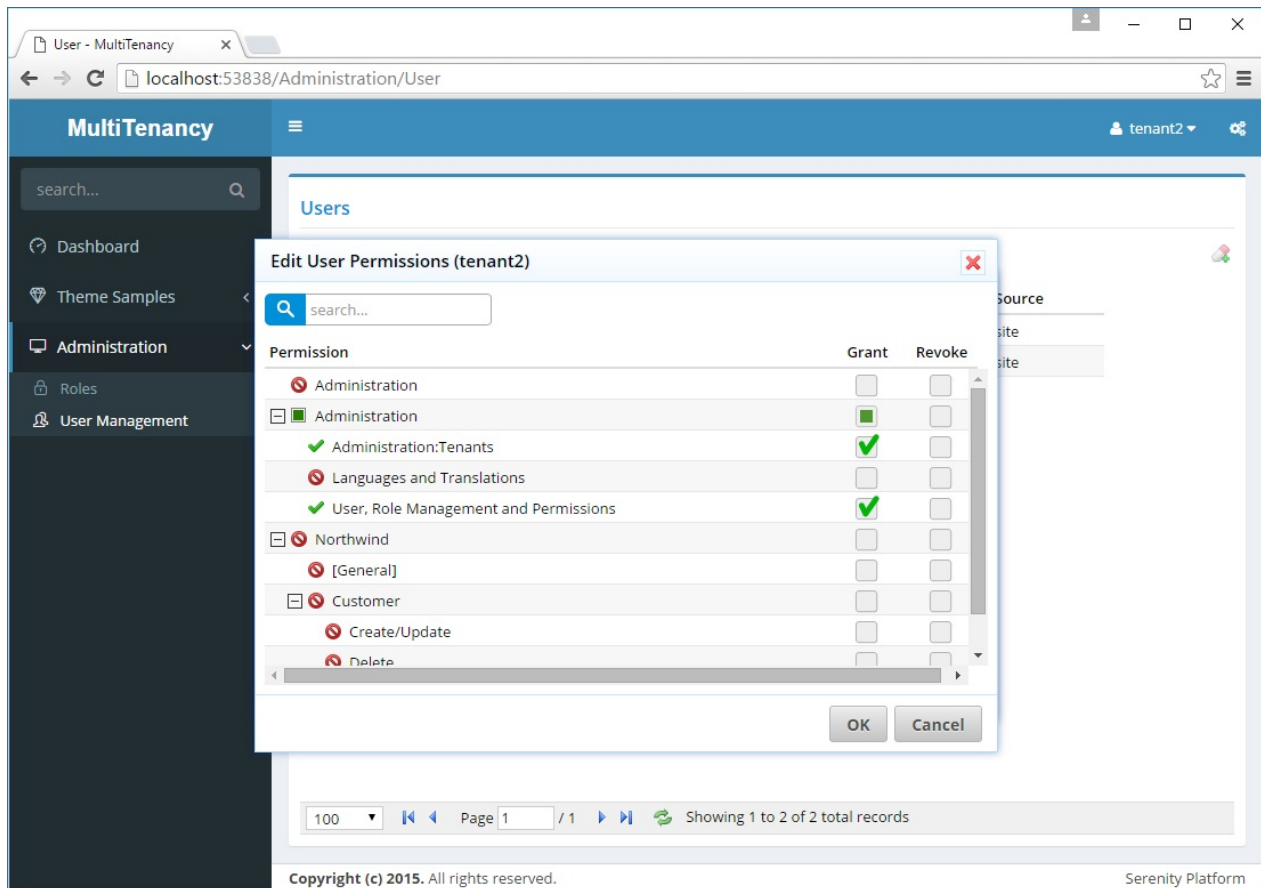
        var user = (UserDefinition)Authorization.UserDefinition;
        if (Row.TenantId != user.TenantId)
            Authorization.ValidatePermission(PermissionKeys.Tenants);
    }
}

private class MyUndeleteHandler : UndeleteRequestHandler<MyRow>
{
    protected override void ValidateRequest()
    {
        base.ValidateRequest();

        var user = (UserDefinition)Authorization.UserDefinition;
        if (Row.TenantId != user.TenantId)
            Authorization.ValidatePermission(PermissionKeys.Tenants);
    }
}
```

# Hiding the Tenant Administration Permission

We now have one little problem. User *tenant2* has permission *Administration:Security* so he can access user and role permission dialogs. Thus, he can grant himself *Administration:Tenants* permission using the permission UI.



Serenity scans your assemblies for attributes like *ReadPermission*, *WritePermission*, *PageAuthorize*, *ServiceAuthorize* etc. and lists these permissions in edit permissions dialog.

We should first remove it from this pre-populated list.

Find method, *ListPermissionKeys* in *UserPermissionRepository.cs*:

```
public ListResponse<string> ListPermissionKeys()
{
    return LocalCache.Get("Administration:PermissionKeys", TimeSpan.Zero, () =>
    {
        //...

        result.Remove(Administration.PermissionKeys.Tenants);
        result.Remove("*");
        result.Remove("?");

        //...
    });
}
```

Now, this permission won't be listed in *Edit User Permissions* or *Edit Role Permissions* dialog.

But, still, he can grant this permission to himself, by some little hacking through *UserPermissionRepository.Update* or *RolePermissionRepository.Update* methods.

We should add some checks to prevent this:

```
public class UserPermissionRepository
{
    public SaveResponse Update(IUnitOfWork uow,
        UserPermissionUpdateRequest request)
    {
        //...
        var newList = new Dictionary<string, bool>(
            StringComparer.OrdinalIgnoreCase);
        foreach (var p in request.Permissions)
            newList[p.PermissionKey] = p.Grant ?? false;

        var allowedKeys = ListPermissionKeys()
            .Entities.ToDictionary(x => x);
        if (newList.Keys.Any(x => !allowedKeys.ContainsKey(x)))
            throw new AccessViolationException();
        //...
    }
}
```

```
public class RolePermissionRepository
{
    public SaveResponse Update(IUnitOfWork uow,
        RolePermissionUpdateRequest request)
    {
        //...
        var newList = new HashSet<string>(
            request.Permissions.ToList(),
            StringComparer.OrdinalIgnoreCase);

        var allowedKeys = new UserPermissionRepository()
            .ListPermissionKeys()
            .Entities.ToDictionary(x => x);
        if (newList.Any(x => !allowedKeys.ContainsKey(x)))
            throw new AccessViolationException();
        //...
```

Here we check if any of the new permission keys that are tried to be granted, are not listed in permission dialog. If so, this is probably a hack attempt.

Actually this check should be the default, even without multi-tenant systems, but usually we trust administrative users. Here, administrators will be only managing their own tenants, so we certainly need this check.

# Making Roles Multi-Tenant

So far, we have made users page work in multi-tenant style. Seems like we did too many changes to make it work. But remember that we are trying to turn a system that is not designed to be multi-tenant into such one.

Let's apply similar principles to the Roles table.

Again, a user in one tenant shouldn't see or modify roles in other tenants and work in isolation.

We start by adding *TenantId* property to *RoleRow.cs*:

```
namespace MultiTenancy.Administration.Entities
{
    //...
    public sealed class RoleRow : Row, IIdRow, INameRow
    {
        [Insertable(false), Updatable(false)]
        public Int32? TenantId
        {
            get { return Fields.TenantId[this]; }
            set { Fields.TenantId[this] = value; }
        }

        //...

        public class RowFields : RowFieldsBase
        {
            //...
            public Int32Field TenantId;
            //...
        }
    }
}
```

Then we'll do several changes in *RoleRepository.cs*:

```
private class MySaveHandler : SaveRequestHandler<MyRow>
{
    protected override void SetInternalFields()
    {
        base.SetInternalFields();

        if (IsCreate)
            Row.TenantId = ((UserDefinition)Authorization.UserDefinition).TenantId;
    }
}
```

```
protected override void ValidateRequest()
{
    base.ValidateRequest();

    if (IsUpdate)
    {
        var user = (UserDefinition)Authorization.UserDefinition;
        if (Old.TenantId != user.TenantId)
            Authorization.ValidatePermission(PermissionKeys.Tenants);
    }
}

private class MyDeleteHandler : DeleteRequestHandler<MyRow>
{
    protected override void ValidateRequest()
    {
        base.ValidateRequest();

        var user = (UserDefinition)Authorization.UserDefinition;
        if (Row.TenantId != user.TenantId)
            Authorization.ValidatePermission(PermissionKeys.Tenants);
    }
}

private class MyRetrieveHandler : RetrieveRequestHandler<MyRow>
{
    protected override void PrepareQuery(SqlQuery query)
    {
        base.PrepareQuery(query);

        var user = (UserDefinition)Authorization.UserDefinition;
        if (!Authorization.HasPermission(PermissionKeys.Tenants))
            query.Where(fld.TenantId == user.TenantId);
    }
}

private class MyListHandler : ListRequestHandler<MyRow>
{
    protected override void ApplyFilters(SqlQuery query)
    {
        base.ApplyFilters(query);

        var user = (UserDefinition)Authorization.UserDefinition;
        if (!Authorization.HasPermission(PermissionKeys.Tenants))
            query.Where(fld.TenantId == user.TenantId);
    }
}
```





# Using Serenity Service Behaviors

If wanted to extend this multi-tenant system to other tables in Northwind, we would repeat same steps we did with Roles. Though it doesn't look so hard, it's too much of manual work.

Serenity provides service behavior system, which allows you to intercept Create, Update, Retrieve, List, Delete handlers and add custom code to them.

Some operations in these handlers, like capture log, unique constraint validation etc. are already implemented as service behaviors.

Behaviors might be activated for all rows, or based on some rule, like having a specific attribute or interface. For example, CaptureLogBehavior activates for rows with [CaptureLog] attribute.

We'll first define an interface *IMultiTenantRow* that will trigger our new behavior. Place this class in file *IMultiTenantRow.cs*, next to *TenantRow.cs*:

```
using Serenity.Data;

namespace MultiTenancy
{
    public interface IMultiTenantRow
    {
        Int32Field TenantIdField { get; }
    }
}
```

Then add this behavior in file *MultiTenantBehavior.cs* next to it:

```
using MultiTenancy.Administration;
using Serenity;
using Serenity.Data;
using Serenity.Services;

namespace MultiTenancy
{
    public class MultiTenantBehavior : IImplicitBehavior,
        ISaveBehavior, IDeleteBehavior,
        IListBehavior, IRetrieveBehavior
    {
        private Int32Field fldTenantId;

        public bool ActivateFor(Row row)
        {
            var mt = row as IMultiTenantRow;
```

```
        if (mt == null)
            return false;

        fldTenantId = mt.TenantIdField;
        return true;
    }

    public void OnPrepareQuery(IRetrieveRequestHandler handler,
        SqlQuery query)
    {
        var user = (UserDefinition)Authorization.UserDefinition;
        if (!Authorization.HasPermission(PermissionKeys.Tenants))
            query.Where(fldTenantId == user.TenantId);
    }

    public void OnPrepareQuery(IListRequestHandler handler,
        SqlQuery query)
    {
        var user = (UserDefinition)Authorization.UserDefinition;
        if (!Authorization.HasPermission(PermissionKeys.Tenants))
            query.Where(fldTenantId == user.TenantId);
    }

    public void OnSetInternalFields(ISaveRequestHandler handler)
    {
        if (handler.IsCreate)
            fldTenantId[handler.Row] =
                ((UserDefinition)Authorization
                    .UserDefinition).TenantId;
    }

    public void OnValidateRequest(ISaveRequestHandler handler)
    {
        if (handler.IsUpdate)
        {
            var user = (UserDefinition)Authorization.UserDefinition;
            if (fldTenantId[handler.Old] != fldTenantId[handler.Row])
                Authorization.ValidatePermission(PermissionKeys.Tenants);
        }
    }

    public void OnValidateRequest(IDeleteRequestHandler handler)
    {
        var user = (UserDefinition)Authorization.UserDefinition;
        if (fldTenantId[handler.Row] != user.TenantId)
            Authorization.ValidatePermission(
                PermissionKeys.Tenants);
    }

    public void OnAfterDelete(IDeleteRequestHandler handler) { }
    public void OnAfterExecuteQuery(IRetrieveRequestHandler handler) { }
    public void OnAfterExecuteQuery(IListRequestHandler handler) { }
    public void OnAfterSave(ISaveRequestHandler handler) { }
```

```
public void OnApplyFilters(IListRequestHandler handler, SqlQuery query) { }
public void OnAudit(IDeleteRequestHandler handler) { }
public void OnAudit(ISaveRequestHandler handler) { }
public void OnBeforeDelete(IDeleteRequestHandler handler) { }
public void OnBeforeExecuteQuery(IRetrieveRequestHandler handler) { }
public void OnBeforeExecuteQuery(IListRequestHandler handler) { }
public void OnBeforeSave(ISaveRequestHandler handler) { }
public void OnPrepareQuery(IDeleteRequestHandler handler, SqlQuery query) { }
public void OnPrepareQuery(ISaveRequestHandler handler, SqlQuery query) { }
public void OnReturn(IDeleteRequestHandler handler) { }
public void OnReturn(IRetrieveRequestHandler handler) { }
public void OnReturn(IListRequestHandler handler) { }
public void OnReturn(ISaveRequestHandler handler) { }
public void OnValidateRequest(IRetrieveRequestHandler handler) { }
public void OnValidateRequest(IListRequestHandler handler) { }
}
}
```

Behavior classes with `ImplicitBehavior` interface decide if they should be activated for a specific row type.

They do this by implementing `ActivateFor` method, which is called by request handlers.

In this method, we check if row type implements `IMultiTenantRow` interface. If not it simply returns false.

Then we get a private reference to `TenantIdField` to reuse it later in other methods.

`ActivateFor` is only called once per every handler type and row. If this method returns true, behavior instance is cached aggressively for performance reasons, and reused for any request for this row and handler type.

Thus, everything you write in other methods must be thread-safe, as one instance is shared by all requests.

A behavior, might intercept one or more of `Retrieve`, `List`, `Save`, `Delete` handlers. It does this by implementing `IRetrieveBehavior`, `IListBehavior`, `ISaveBehavior`, or `IDeleteBehavior` interfaces.

Here, we need to intercept all of these service calls, so we implement all interfaces.

We only fill in methods we are interested in, and leave others empty.

The methods we implement here, corresponds to methods we override in `RoleRepository.cs` in previous section. The code they contain is almost same, except here we need to be more generic, as this behavior will work for any row type implementing `IMultiTenantRow`.

## Reimplementing RoleRepository With Using the Behavior

Now revert every change we made in *RoleRepository.cs*:

```
private class MySaveHandler : SaveRequestHandler<MyRow> { }
private class MyDeleteHandler : DeleteRequestHandler<MyRow> { }
private class MyRetrieveHandler : RetrieveRequestHandler<MyRow> { }
private class MyListHandler : ListRequestHandler<MyRow> { }
```

And add *IMultiTenantRow* interface to *RoleRow*:

```
namespace MultiTenancy.Administration.Entities
{
    //...
    public sealed class RoleRow : Row, IIdRow, INameRow, IMultiTenantRow
    {
        //...
        public Int32Field TenantIdField
        {
            get { return Fields.TenantId; }
        }
        //...
    }
}
```

You should get the same result with much less code. Declarative programming is almost always better.

# Extending Multi-Tenant Behavior To Northwind

As now we have a behavior handling repository details, we just need to add *IMultiTenantRow* interface to rows and add *TenantId* property.

Start with *SupplierRow.cs*:

```
namespace MultiTenancy.Northwind.Entities
{
    //...
    public sealed class SupplierRow : Row,
        IIdRow, INameRow, IMultiTenantRow
    {
        //...
        [Insertable(false), Updatable(false)]
        public Int32? TenantId
        {
            get { return Fields.TenantId[this]; }
            set { Fields.TenantId[this] = value; }
        }

        public Int32Field TenantIdField
        {
            get { return Fields.TenantId; }
        }

        //...

        public class RowFields : RowFieldsBase
        {
            //...
            public readonly Int32Field TenantId;
        }
    }
}
```

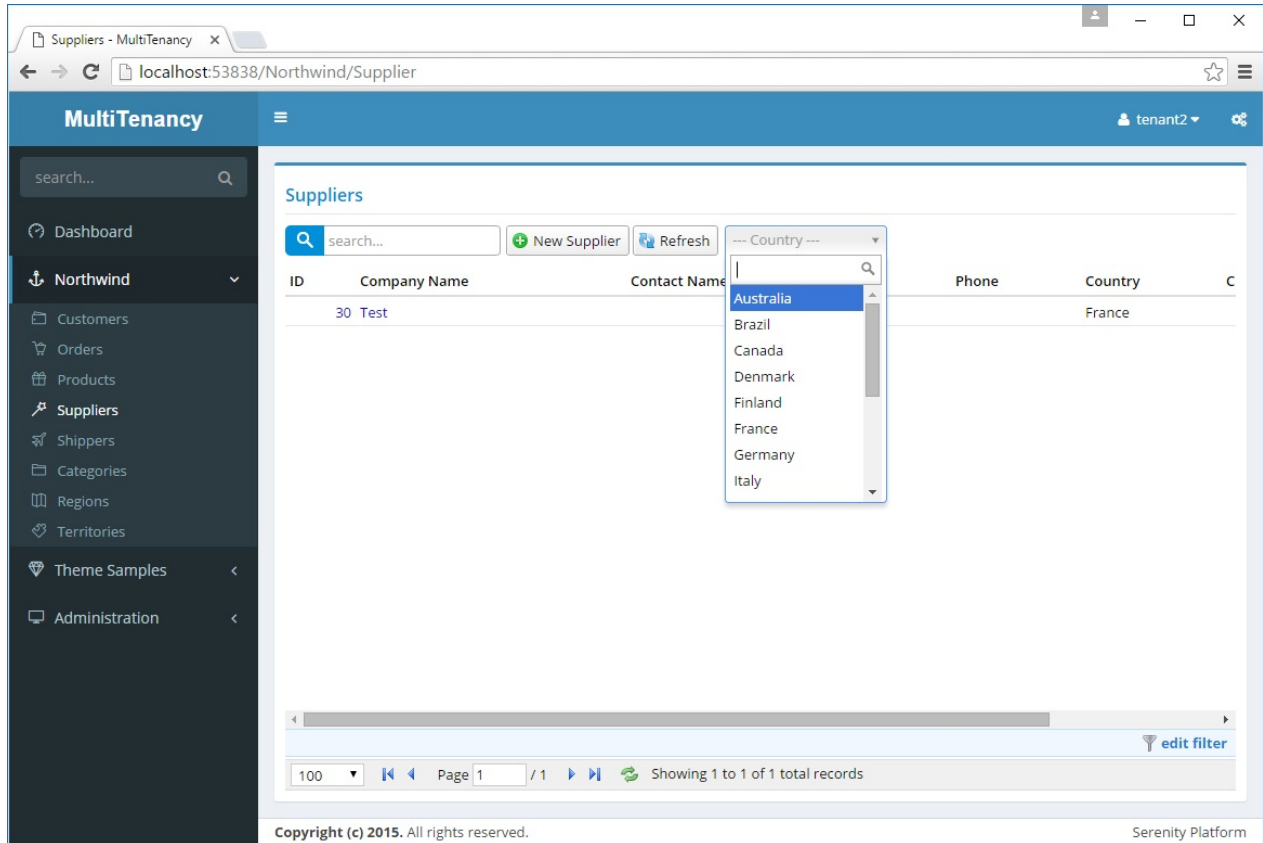
When you these changes in *SupplierRow* and build, you'll see that *tenant2* can't see suppliers from other tenants in suppliers page.

Now repeat these for *EmployeeRow*, *CategoryRow*, *CustomerRow*, *ShipperRow*, *OrderRow*, *ProductRow*, *RegionRow* and *TerritoryRow*.



# Handling Lookup Scripts

If we open *Suppliers* page now, we'll see that *tenant2* can only see suppliers that belongs to its tenant. But on top right of the grid, in country dropdown, all countries are listed:



This data is feed to script side through a dynamic script. It doesn't load this data with *List* services we handled recently.

The lookup script that produces this dropdown is defined in *SupplierCountryLookup.cs*:



```
namespace MultiTenancy.Northwind.Scripts
{
    using Serenity.ComponentModel;
    using Serenity.Data;
    using Serenity.Web;

    [LookupScript("Northwind.SupplierCountry")]
    public class SupplierCountryLookup :
        RowLookupScript<Entities.SupplierRow>
    {
        public SupplierCountryLookup()
        {
            IdField = TextField = "Country";
        }

        protected override void PrepareQuery(SqlQuery query)
        {
            var fld = Entities.SupplierRow.Fields;
            query.Distinct(true)
                .Select(fld.Country)
                .Where(
                    new Criteria(fld.Country) != "" &
                    new Criteria(fld.Country).NotNull());
        }

        protected override void ApplyOrder(SqlQuery query)
        {
        }
    }
}
```

We couldn't use a simple [LookupScript] attribute on a row class here, because there is actually no country table in Northwind database. We are collecting country names from existing records in Supplier table using distinct.

We should filter its query by current tenant.

But this lookup class derives from *RowLookupScript* base class. Let's create a new base class, to prepare for other lookup scripts that we'll have to handle later.

```
namespace MultiTenancy.Northwind.Scripts
{
    using Administration;
    using Serenity;
    using Serenity.Data;
    using Serenity.Web;
    using System;

    public class MultiTenantRowLookupScript<TRow> :
        RowLookupScript<TRow>
        where TRow : Row, IMultiTenantRow, new()
    {
        public MultiTenantRowLookupScript()
        {
            Expiration = TimeSpan.FromDays(-1);
        }

        protected override void PrepareQuery(SqlQuery query)
        {
            base.PrepareQuery(query);
            AddTenantFilter(query);
        }

        protected void AddTenantFilter(SqlQuery query)
        {
            var r = new TRow();
            query.Where(r.TenantIdField ==
                ((UserDefinition)Authorization.UserDefinition).TenantId);
        }

        public override string GetScript()
        {
            return TwoLevelCache.GetLocalStoreOnly("MultiTenantLookup:" +
                this.ScriptName + ":" +
                ((UserDefinition)Authorization.UserDefinition).TenantId,
                TimeSpan.FromHours(1),
                new TRow().GetFields().GenerationKey, () =>
                {
                    return base.GetScript();
                });
        }
    }
}
```

This will be our base class for multi-tenant lookup scripts.

We first set expiration to a negative timespan to disable caching. Why do we have to do this? Because dynamic script manager caches lookup scripts by their keys. But we'll have multiple versions of a lookup script based on TenantId values.

We'll turn off caching at dynamic script manager level and handle caching ourself in *GetScript* method. In *GetScript* method, we are using *TwoLevelCache.GetLocalStoreOnly* to call base method, that generates our lookup script, and cache its result with a cache key including *TenantId*.

See relevant section for more info about *TwoLevelCache* class.

By overriding, *PrepareQuery* method, we are adding a filter by current *TenantId*, just like we did in list service handlers.

Now its time to rewrite our *SupplierCountryLookup* using this new base class:

```
namespace MultiTenancy.Northwind.Scripts
{
    using Serenity.ComponentModel;
    using Serenity.Data;
    using Serenity.Web;

    [LookupScript("Northwind.SupplierCountry")]
    public class SupplierCountryLookup :
        MultiTenantRowLookupScript<Entities.SupplierRow>
    {
        public SupplierCountryLookup()
        {
            IdField = TextField = "Country";
        }

        protected override void PrepareQuery(SqlQuery query)
        {
            var fld = Entities.SupplierRow.Fields;
            query.Distinct(true)
                .Select(fld.Country)
                .Where(
                    new Criteria(fld.Country) != "" &
                    new Criteria(fld.Country).NotNull());

            AddTenantFilter(query);
        }

        protected override void ApplyOrder(SqlQuery query)
        {
        }
    }
}
```

We just called *AddTenantFilter* method manually, because we weren't calling base *PrepareQuery* method here (so it won't be called by base class).

Please first delete *Northwind.DynamicScripts.cs* file, if you have it.

There are several more similar lookup scripts in *CustomerCountryLookup*, *CustomerCityLookup*, *OrderShipCityLookup*, *OrderShipCountryLookup*. I'll do similar changes in them. Change base class to *MultiTenantRowLookupScript* and call *AddTenantFilter* in *PrepareQuery* method.

## Lookup Script Declarations On Rows

We now have one more problem to solve. If you open *Orders* page, you'll see that *Ship Via* and *Employee* filter dropdowns still lists records from other tenants. It is because we defined their lookup scripts by a `[LookupScript]` attribute on their rows.

By default, `LookupScript` generates a lookup instance based on `RowLookupScript<>` type. We need to change it to `MultiTenantRowLookupScript<>` for these multi-tenant rows.

Let's fix employee lookup first. Replace `[LookupScript]` attribute like below in *EmployeeRow*.

```
[LookupScript("Northwind.Employee",
  LookupType = typeof(MultiTenantRowLookupScript<>))]
public sealed class EmployeeRow : Row, IIdRow,
  INameRow, IMultiTenantRow
{
  //...
```

Note that this requires Serenity 2.9.22+

Do similar (add `LookupType`) for *Shipper*, *Product*, *Supplier*, *Category*, *Region* and *Territory* rows.

Now Northwind supports multi-tenancy.

There might be some glitches i missed, report in Serenity Github repository if any.

# Meeting Management (In Progress...)

In this tutorial we are going to develop a meeting management system that will help us keep a track of corporate meetings.

We'll first plan a meeting, with its location, time, agenda and attendees, then send an invitation to those attendees with an e-mail.

Application will also let us store decisions taken in the meeting, and will inform attendees with a meeting report e-mail containing these decisions.

Code for this tutorial will be available at:

<https://github.com/volkanceylan/MeetingManagement>

## Creating Project

Start by creating a new project using Serene template, and name it MeetingManagement.

## Removing Northwind

Remove Northwind using the how-to guide.

# Creating Lookup Tables

Let's start by creating lookup tables we'll need.

Here is a list of these tables:

- Meeting Types (Board Meeting, Weekly Analytics, SCRUM Meeting, Annual Meeting, so on...)
- Locations (where meeting will be held, room numbers, address etc.)
- Agenda Types (what subject(s) an agenda is about, might be multiple)
- Units (which unit is organizing the meeting)
- Contacts (people which would attend meetings, reporters, managers etc.)

We'll use database schema *met* for tables.

Create a new migration under, *Modules/Common/Migrations/DefaultDB* with name *DefaultDB\_20160709\_232400\_MeetingLookups*:

```
using FluentMigrator;

namespace MeetingManagement.Migrations.DefaultDB
{
    [Migration(20160709232400)]
    public class DefaultDB_20160709_232400_MeetingLookups
        : AutoReversingMigration
    {
        public override void Up()
        {
            Create.Schema("met");

            Create.Table("AgendaTypes").InSchema("met")
                .WithColumn("AgendaTypeId").AsInt32()
                    .Identity().PrimaryKey().NotNullable()
                .WithColumn("Name").AsString(100).NotNullable();

            Create.Table("Contacts").InSchema("met")
                .WithColumn("ContactId").AsInt32()
                    .Identity().PrimaryKey().NotNullable()
                .WithColumn("Title").AsString(30).Nullable()
                .WithColumn("FirstName").AsString(50).NotNullable()
                .WithColumn("LastName").AsString(50).NotNullable()
                .WithColumn("Email").AsString(100).NotNullable();

            Create.Table("Locations").InSchema("met")
                .WithColumn("LocationId").AsInt32()
                    .Identity().PrimaryKey().NotNullable()
                .WithColumn("Name").AsString(100).NotNullable()
                .WithColumn("Address").AsString(300).Nullable()
                .WithColumn("Latitude").AsDouble()
                .WithColumn("Longitude").AsDouble();

            Create.Table("MeetingTypes").InSchema("met")
                .WithColumn("MeetingTypeId").AsInt32()
                    .Identity().PrimaryKey().NotNullable()
                .WithColumn("Name").AsString(100).NotNullable();

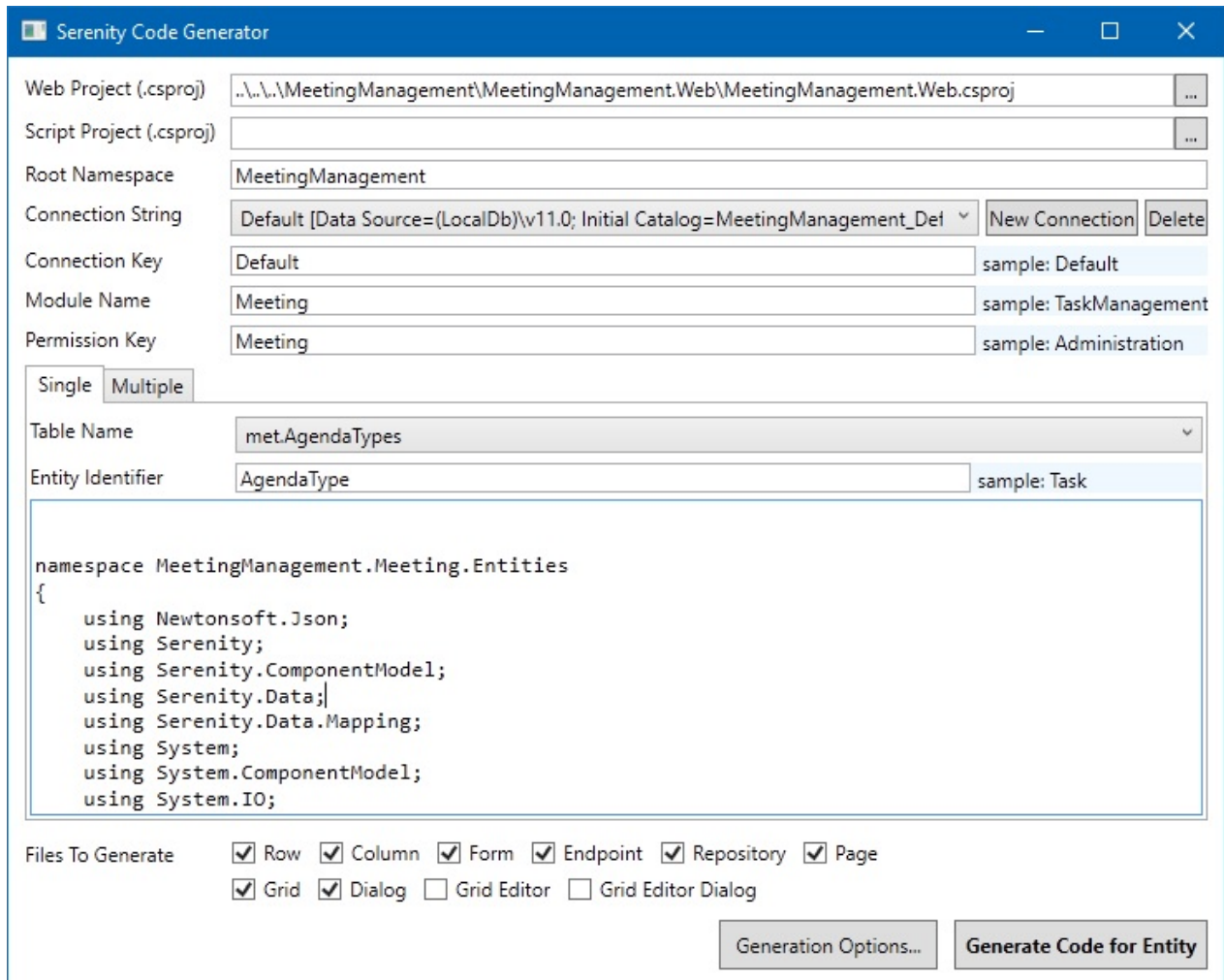
            Create.Table("Units").InSchema("met")
                .WithColumn("UnitId").AsInt32()
                    .Identity().PrimaryKey().NotNullable()
                .WithColumn("Name").AsString(100).NotNullable();
        }
    }
}
```

## Generating Code for Lookup Tables

Our module name will be *Meetings*. We should use non-plural entity identifiers for generated code:

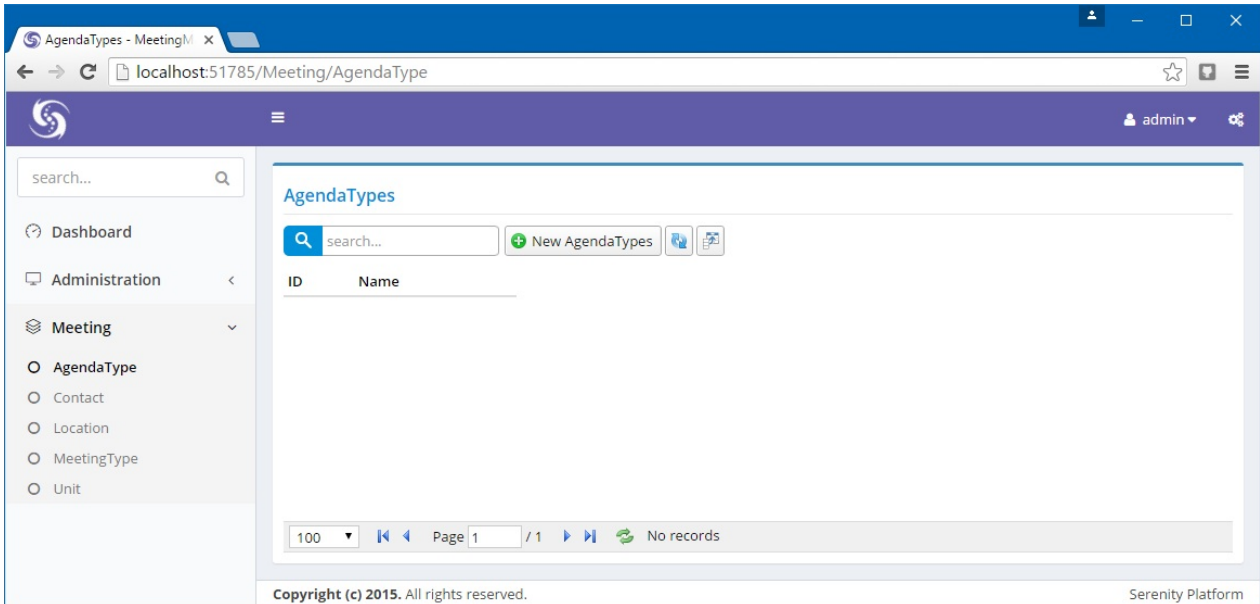
- AgendaTypes => AgendaType
- Contacts => Contact
- Locations => Location
- MeetingTypes => MeetingType
- Units => Unit

Generate code for these 5 tables using the entity identifiers given above:



Generated interface for these tables is fine enough. Just need to do a few cosmetic touches.





## Moving Navigation Links to NavigationItems.cs

Open *AgendaTypePage.cs*, *ContactPage.cs*, *LocationPage.cs*, *MeetingTypePage.cs* and *UnitPage.cs* files and move navigation links at top of them to *NavigationItems.cs*:

```
using Serenity.Navigation;
using Administration = MeetingManagement.Administration.Pages;
using Meeting = MeetingManagement.Meeting.Pages;

[assembly: NavigationLink(1000, "Dashboard",
    url: "~/", permission: "", icon: "icon-speedometer")]

[assembly: NavigationMenu(2000, "Meeting")]
[assembly: NavigationLink(2500, "Meeting/Agenda Types",
    typeof(Meeting.AgendaTypeController))]
[assembly: NavigationLink(2600, "Meeting/Contacts",
    typeof(Meeting.ContactController))]
[assembly: NavigationLink(2700, "Meeting/Locations",
    typeof(Meeting.LocationController))]
[assembly: NavigationLink(2800, "Meeting/Meeting Types",
    typeof(Meeting.MeetingTypeController))]
[assembly: NavigationLink(2900, "Meeting/Units",
    typeof(Meeting.UnitController))]
```

## Setting DisplayName and InstanceName Attributes of Lookup Tables

Open *AgendaTypeRow.cs*, *ContactRow.cs*, *LocationRow.cs*, *MeetingTypeRow.cs* and *UnitRow.cs* files and change *DisplayName* and *InstanceName* attributes like below:

- *AgendaTypeRow* => "Agenda Types", "Agenda Type"

- ContactRow => "Contacts", "Contact"
- LocationRow => "Locations", "Location"
- MeetingTypeRow => "Meeting Types", "Meeting Type"
- UnitRow => "Units", "Unit"

```
[ConnectionKey("Default"), TwoLevelCached,  
  DisplayName("Agenda Types"), InstanceName("Agenda Type")]  
[ReadPermission("Meeting")]  
[ModifyPermission("Meeting")]  
public sealed class AgendaTypeRow : Row, IIdRow, INameRow  
{
```

# How To Guides

# How To: Remove Northwind & Other Samples From Serene

After you take Northwind as a sample, and develop your own project, you would want to remove Northwind module, and other sample artifacts from your project.

Here is how to get rid of them.

We assume your solution name is *MyProject*, so you have *MyProject.Web* project in your solution.

Perform steps below in Visual Studio:

## Removing Project Folders

- Remove *MyProject.Web/Modules/AdminLTE* folder. This will remove all server side code related to theme samples.
- Remove *MyProject.Web/Modules/BasicSamples* folder. This will remove all server side code related to basic samples.
- Remove *MyProject.Web/Modules/Northwind* folder. This will remove all server side code related to Northwind.

## Removing Navigation Items

Navigation items for these modules are moved under relevant modules folder in v2.5.3. If you are using an older version,

- Open *MyProject.Web/Modules/Common/Navigation/NavigationItems.cs*, remove all lines with *Northwind*, *Basic Samples* and *Theme Samples* and remove these two lines:

```
using Northwind = MovieTutorial.Northwind.Pages;  
using Basic = MovieTutorial.BasicSamples.Pages;
```

## Removing Migration Scripts

Remove folder *MyProject.Web/Modules/Common/Migrations/NorthwindDB/* with all files under it.

Remove "Northwind" from following line in *MyProject.Web/App\_Start/SiteInitialization.Migrations.cs*:

```
private static string[] databaseKeys = new[] { "Default", "Northwind" };
```

Also remove *Northwind* connection string from *web.config*.

```
<add name="Northwind" connectionString="Data Source=(LocalDb)\v11.0;  
    Initial Catalog=MovieTutorial_Northwind_v1;  
    Integrated Security=True"  
    providerName="System.Data.SqlClient" />
```

## Removing LESS Entries

- Open *MyProject.Web/Content/site/site.less* file, remove following lines:

```
@import "site.basicsamples.less";  
@import "site.northwind.less";
```

- Remove *MyProject.Web/Content/site/site.basicsamples.less* file.
- Remove *MyProject.Web/Content/site/site.northwind.less* file.
- Open *MyProject.Web/Content/site/rtl.css* file, remove sections with Northwind.

## Removing Localization Texts

- Open *MyProject.Web/Modules/Texts.cs* and remove following lines:

```
public static LocalText NorthwindPhone = "...";  
public static LocalText NorthwindPhoneMultiple = "...";
```

- Remove folder *MyProject.Web/Scripts/site/texts/northwind*
- Remove folder *MyProject.Web/Scripts/site/texts/samples*

## Removing Northwind / Samples Generated Code

- Expand *MyProject.Web/Modules/Common/Imports/ServerTypings/ServerTypings.tt*.  
Select files starting with *Northwind.*, *BasicSamples.* and delete them.

## Removing Northwind Numbers From Dashboard

Open *DashboardPage.cs*, remove these using lines:

```
using Northwind;
using Northwind.Entities;
```

As Dashboard gets numbers from Northwind tables, you should modify *Index()* action like this:

```
[Authorize, HttpGet, Route("~/")]
public ActionResult Index()
{
    var cachedModel = new DashboardPageModel()
    {
    };

    return View(MVC.Views.Common.Dashboard.DashboardIndex, cachedModel);
}
```

You should replace this model with something specific to your site, and modify *DashboardIndex* accordingly.

Open *DashboardIndex.cshtml*, clear *href* attributes containing "Northwind" like:

```
<a href="~/Northwind/Order?shippingState=1"></a>
<a href=""></a>
```

## Building Project and Running T4 (.tt) Templates

- Now rebuild your solution.
- Make sure it is built *successfully* before executing next step.
- Click *Build* menu and click *Transform All Templates*.
- Rebuild your solution again.
- Search for *Northwind*, *Basic Samples* and *Theme Samples* in all solution items. It should find no results.
- Run your project, now Northwind and Sample menus are gone.

## Removing Northwind Tables

Northwind tables are in a separate database, so you can just drop it.



# How To: Update Serenity NuGet Packages

Serene template contains references to following Serenity NuGet packages:

```
Serenity.Core  
Serenity.Data  
Serenity.Data.Entity  
Serenity.Services  
Serenity.Web  
Serenity.CodeGenerator
```

To update Serenity packages to latest version, open package manager console (click View -> Other Windows -> Package Manager Console).

And type following:

```
Update-Package Serenity.Web  
Update-Package Serenity.CodeGenerator
```

Updating these two packages will also update others (because of dependencies).



# How To: Upgrade to Serenity 2.0 and Enable TypeScript

Serenity has TypeScript support starting with version 2.0.

This is a migration guide for users that started with an older Serene template, and wants to use TypeScript features.

If you don't need TypeScript, just update your Serenity packages and it should work as normal.

Even if you won't need TypeScript, it's recommended to perform steps listed here to keep your project up to date. This might also help you avoid future problems as there has been many changes in Serene for TypeScript support.

## Should I Switch To TypeScript?

TypeScript support in Serenity is stable as of writing and is strongly recommended. TypeScript is the future for Serenity applications, as it has a stronger backing at the moment (Microsoft and average number of users).

Also TypeScript feels like native Javascript with proper intellisense, refactoring and compile time type checking.

We've been using Saltaralle with Serenity since start but its future is a bit blurry. It didn't get any updates since it is acquired by Bridge.NET, last June (2015).

Your old code written in Saltaralle will continue to work. It will be supported as long as possible with Serenity for backward compability.

If Bridge.NET v2.0 (next Saltaralle) comes out, we may also try to switch, unless it involves too many changes to handle.

## Migrating Your Serene Application to v2.0

### Check that your solution is building properly

First make sure your solution is properly building.

If possible, take a ZIP backup of solution, as some steps we'll perform might be hard to revert.

## Install TypeScript

Install TypeScript 1.8+ from

<https://www.typescriptlang.org/#download-links>

for your Visual Studio version.

## Update NuGet Packages

Update to 2.0 packages as you'd normally do:

```
Update-Package Serenity.Web
Update-Package Serenity.CodeGenerator
Update-Package Serenity.Script
```

While updating Serenity.Web, VS might show a dialog with text "Your Project has been configured to support TypeScript". Click YES.

## Ensuring Package Updates Caused No Problems

Rebuild your solution again and run it. Open some pages, dialogs etc. and make sure that it is working properly with 2.0 packages.

## Configuring Web Project for TypeScript

Unload MyProject.Web and edit it.

Add lines below after TypeScriptToolsVersion line:

```
// ...
<TypeScriptToolsVersion>1.8</TypeScriptToolsVersion>
<TypeScriptCompileBlocked>True</TypeScriptCompileBlocked>
</PropertyGroup>
<PropertyGroup>
  <TypeScriptCharset>utf-8</TypeScriptCharset>
  <TypeScriptEmitBOM>True</TypeScriptEmitBOM>
  <TypeScriptGeneratesDeclarations>False</TypeScriptGeneratesDeclarations>
  <TypeScriptExperimentalDecorators>True</TypeScriptExperimentalDecorators>
  <TypeScriptOutFile>Scripts\site\Serene.Web.js</TypeScriptOutFile>
  <TypeScriptCompileOnSaveEnabled>False</TypeScriptCompileOnSaveEnabled>
</PropertyGroup>
```

Replace **Serene.Web.js** with your project name.

In the end of same file, you'll see lines like below:

```
<Import Project="..Microsoft.CSharp.targets" />
<Import Project="..Microsoft.WebApplication.targets" />
<Import Project="..Microsoft.TypeScript.targets" />
```

Make sure the line with TypeScript.targets with is under all other targets. Move it under WebApplication.targets if not. VS puts them before Microsoft.WebApplication.targets and somehow it doesn't work that way.

Also, at the bottom of file, you'll find *CompileSiteLess* step, add TSC to end of it:

```
<Target Name="CompileSiteLess" AfterTargets="AfterBuild">
  <Exec Command="&quot;$(ProjectDir)tools\node\lessc.cmd&quot;
    &quot;$(ProjectDir)Content\site\site.less&quot; &gt;
    &quot;$(ProjectDir)Content\site\site.css&quot;">
  </Exec>
  <Exec Command="&quot;$(TscToolPath)\$(TypeScriptToolsVersion)\
    $(TscToolExe)&quot; -project &quot;
    $(ProjectDir)tsconfig.json&quot;" ContinueOnError="true" />
</Target>
```

Save changes, reload the project and follow to next step.

## Adding tsconfig.json File

Add a **tsconfig.json** file to the root of your Web project (where web.config and Global.asax files are) with content like below:

```
{
  "compileOnSave": true,
  "compilerOptions": {
    "preserveConstEnums": true,
    "experimentalDecorators": true,
    "declaration": true,
    "emitBOM": true,
    "sourceMap": true,
    "target": "ES5",
    "outFile": "Scripts/site/Serene.Web.js"
  },
  "exclude": [
    "Scripts/site/Serene.Web.d.ts"
  ]
}
```

Replace **Serene.Web** with your project name.

## Add a Test TypeScript File

Add a **dummy.ts** file under `YourProject.Web/scripts/site/dummy.ts`. Open it and type something like below:

```
namespace MyProject {
    export class Dummy {
    }
}
```

When you save it, there should be a `MyProject.Web.js` file there with content below. If you can't see it, click Show All Files and refresh folder.

```
var MyProject;
(function (MyProject) {
    var Dummy = (function () {
        function Dummy() {
        }
        return Dummy;
    })();
    MyProject.Dummy = Dummy;
})(MyProject || (MyProject = {}));
//# sourceMappingURL=SereneUpgrading.Web.js.map
```

Right click and include that file to your project. Now you can delete `dummy.ts`.

If you are using a version before VS2015 and compile on save is not working, your TS files will be compiled at project build.

## Including `MyProject.Web.js` file in `_LayoutHead.cshtml`

Edit `MyProject.Web/Views/Shared/_LayoutHead.cshtml` and include `MyProject.Web.js` right after `MyProject.Script.js` file:

```
// ...
@Html.Script("~/Scripts/Site/MyProject.Script.js")
@Html.Script("~/Scripts/Site/MyProject.Web.js")
// ...
```

Your project is configured for TypeScript.

## Changing Location for T4 Templates

Serene v2.0 has merged some .TT templates and created new one for TypeScript code generation.

Please make sure your project is building successfully and DON'T CLEAN it while performing these steps, otherwise you may end up with a broken project.

Locate file `YourProject.Web\Modules\Common\Imports\MultipleOutputHelper.ttinclude`

Make a copy of it in same folder with name `CodeGenerationHelpers.ttinclude`

Get latest source of `CodeGenerationHelpers.ttinclude` from address below and copy paste it to `CodeGenerationHelpers.ttinclude` file you just created:

<https://raw.githubusercontent.com/volkanceylan/Serene/master/Serene/Serene.Web/Modules/Common/Imports/CodeGenerationHelpers.ttinclude>

Search and Replace **Serene** with **YourProjectName** in this file if any. There shouldn't be any **Serene** word in this file as of writing.

You may also create a new Serene project with latest version of template to get these files.

## ClientTypes.tt

Create folder `YourProject.Web\Modules\Common\Imports\ClientTypes` and move `ScriptEditorTypes.tt` to there, then rename `ScriptEditorTypes.tt` to `ClientTypes.tt`.

Grab latest source of `ClientTypes.tt` file from address below and copy paste it to `ClientTypes.tt` file you just moved:

<https://raw.githubusercontent.com/volkanceylan/Serene/master/Serene/Serene.Web/Modules/Common/Imports/ClientTypes/ClientTypes.tt>

Search and Replace **Serene** with **YourProjectName** in this file if any.

## ServerTypings.tt

Create folder `YourProject.Web\Modules\Common\Imports\ServerTypings` and move `ScriptFormatterTypes.tt` to there, then rename `ScriptFormatterTypes.tt` to `ServerTypings.tt`.

Grab latest source of `ServerTypings.tt` file from address below and copy paste it to `ServerTypings.tt` file you just moved:

<https://raw.githubusercontent.com/volkanceylan/Serene/master/Serene/Serene.Web/Modules/Common/Imports/ServerTypings/ServerTypings.tt>

Search and Replace **Serene** with **YourProjectName** in this file if any.

## Generate Code

While they are open save `ClientTypes.tt` and `ServerTypings.tt` files, and wait for them to generate codes.

Save project and rebuild.

## Changing Location for FormContexts and ServiceContracts T4 Templates

These two templates are merged into one.

We'll repeat similar steps like in Web project.

Locate file `YourProject.Script\Imports\MultipleOutputHelper.ttinclude`

Make a copy of it in same folder with name `CodeGenerationHelpers.ttinclude`

Get latest source of `CodeGenerationHelpers.ttinclude` from address below (it's different!) and copy paste it to `CodeGenerationHelpers.ttinclude` file you just created:

<https://raw.githubusercontent.com/volkanceylan/Serene/b900c67b4c820284379b9c613b16379bb8c530f3/Serene/Serene.Script/Imports/CodeGenerationHelpers.ttinclude>

*Search and Replace **Serene** with **YourProjectName*** in this file. There must be several.

## ServiceContracts.tt

Rename folder `YourProject.Script\Imports\ServiceContracts` to **ServerImports**. Rename `ServiceContracts.tt` to `ServerImports.tt`.

Grab latest source of `ServerImports.tt` file from address below and copy paste it to `ServerImports.tt` file you just renamed:

<https://raw.githubusercontent.com/volkanceylan/Serene/b900c67b4c820284379b9c613b16379bb8c530f3/Serene/Serene.Script/Imports/ServerImports/ServerImports.tt>

*Search and Replace **Serene** with **YourProjectName*** in this file.

Delete folder `FormContexts` with file `FormContext.tt` in it.

Save **ServerImports.tt** and wait for it to generate code. It might take some time because of some slow down due to Saltaralle.

Rebuild solution and make sure it builds properly without any error.

**Congratulations! Your project is ready for TypeScript and other features.**

## What are These New .tt Files

- **ServerTypings.tt**: generated code for TypeScript, containing Row, Form, Column, Service declarations imported from Server (Web) code. Also contains import classes from YourProject.Script file if any.
- **ServerTypes.tt**: generated code for Saltaralle, containing Row, Form, Column, Service declarations imported from Server (Web) code. There is no import classes from TypeScript yet. So if you want to use some TypeScript class in your Saltaralle code, you need to write import classes manually.
- **ClientTypes.tt**: generated code for Web project, containing Editor and Formatter imports from both TypeScript and Saltaralle.

## How Can I Generate TypeScript Grid/Dialog Code

There is such an option in Serenity CodeGenerator (Sergen) now. Just check \*Generate Grid/Dialog Code in TypeScript (instead of Saltaralle) and it will generate YourDialog.ts and YourGrid.ts files under YourProject.Web/Modules/YourEntity directory, instead of YourGrid.cs and YourDialog.cs in YourProject.Script project.

**Please don't generate code for existing Saltaralle dialogs or grids using Sergen. Otherwise you'll have double YourGrid and YourDialog classes and it may lead to unexpected errors.**

# How To: Authenticate With Active Directory or LDAP

Serene 1.8.12+ has some basic ActiveDirectory / LDAP integration samples.

To enable them, you have to fill one of web.config settings.

For ActiveDirectory add a appSetting key `ActiveDirectory` with contents like below:

```
<add key="ActiveDirectory"
      value="{ Domain: 'youractivedirectorydomain' }" />
```

If this doesn't work for your Active Directory server out of the box, you might have to modify *ActiveDirectoryService* class.

When a AD user tries to login first time, Serene authenticates user with this domain, retrieves user details and inserts a user with type `directory` into users table.

AD password hash and user information is cached for one hour, so for one hour user can login with cached credentials, without even hitting AD.

After that, user information is tried to be updated from AD. If an error occurs, user will be allowed to login with cached credentials.

These details can be seen and modified in *AuthenticationService* class.

To enable LDAP authentication (tested with OpenLDAP) you need to add a appSetting key `LDAP` to web.config:

```
<add key="LDAP"
      value="{
        Host: '123.124.125.126',
        Port: 389,
        DistinguishedName: 'dc=yourdomain, dc=com',
        Username: 'cn=someuserthatcanreadldap,ou=groupofthatuser,
                  dc=yourdomain,dc=com',
        Password: 'passwordofthatuser'
      }"
/>
```

Again, there are many different configurations of LDAP servers out there, so if this doesn't work for you, you might have to modify *LdapDirectoryService* class.





# How To: Use a SlickGrid Formatter

This section is pending update for TypeScript

To use a SlickGrid formatter function, like percent complete bar formatter at *%Complete* column of SlickGrid example:

<http://mleibman.github.io/SlickGrid/examples/example2-formatters.html>

## Including Required Resources

First include javascript file containing these formatters in your *\_LayoutHead.cshtml* file (MyProject.Web/Views/Shared/\_LayoutHead.cshtml):

```
//...
@Html.Script("~/Scripts/jquery.slimscroll.js")
@Html.Script("~/Scripts/SlickGrid/slick.formatters.js")
@Html.Script("~/Scripts/Site/MovieTutorial.Script.js")
//...
```

You also need to include following CSS from example.css (can be inserted in site.less):

```
.percent-complete-bar {
  display: inline-block;
  height: 6px;
  -moz-border-radius: 3px;
  -webkit-border-radius: 3px;
}
```

## Declaring a Serenity DataGrid Formatter

Let's say we have *StudentCourseGrid* with a *CourseCompletion* column that we wan't to use *Slick.Formatters.PercentCompleteBar* formatter with.

```
public class StudentCourseColumns
{
  //...
  [width(200)]
  public Decimal CourseCompletion { get; set; }
}
```

To reference a SlickGrid formatter at server side, you need to declare a formatter type for Serenity grids.

In MyApplication.Script project, next to StudentCourseGrid.cs for example, define a file (PercentCompleteBarFormatter.cs) with contents:

```
using Serenity;
using System;

namespace MyApplication
{
    public class PercentCompleteBarFormatter : ISlickFormatter
    {
        private SlickColumnFormatter formatter =
            Type.GetType("Slick.Formatters.PercentCompleteBar").As<SlickColumnFormatte
r>();

        public string Format(SlickFormatterContext ctx)
        {
            return formatter(ctx.Row, ctx.Cell, ctx.Value, ctx.Column, ctx.Item);
        }
    }
}
```

Replace MyApplication with your root namespace (solution name).

Now you can reference it at server side:

```
public class StudentCourseColumns
{
    //...
    [FormatterType("PercentCompleteBar"), Width(200)]
    public Decimal CourseCompletion { get; set; }
}
```

Rebuild your project and you will see that CourseCompletion column has a percentage bar just like in SlickGrid example.

## Getting Intellisense and Compile Time Checking To Work

To get intellisense for PercentCompleteBarFormatter server side (so to avoid using magic strings), you should transform T4 templates (make sure solution builds successfully before transforming).

After this you can reference it like this server side:

```
public class StudentCourseColumns
{
    //...
    [PercentCompleteBarFormatter, Width(200)]
    public Decimal CourseCompletion { get; set; }
}
```

## Alternate Option (Not Recommended)

It is also possible to set SlickGrid column formatter function directly in script side code without defining a Serenity formatter class, e.g. in *StudentCourseGrid.cs* by overriding its *GetColumns* method:

```
protected override List<SlickColumn> GetColumns()
{
    var columns = base.GetColumns();
    columns.Single(x => x.Field == "CourseCompletion").Formatter =
        Type.GetType("Slick.Formatters.PercentCompleteBar").As<SlickColumnFormatte
r>();
    return columns;
}
```

This is not reusable but saves you from defining a formatter class.

# How To: Add a Row Selection Column

**This section is pending update for TypeScript**

To add a column to select individual rows or all rows, `GridRowSelectionMixin` can be used.

`GridRowSelectionMixin` is available in Serenity 1.6.8+

Sample code:

```
public class MyGrid : EntityGrid<MyRow>
{
    private GridRowSelectionMixin rowSelection;

    public MyGrid(jQueryObject container)
        : base(container)
    {
        rowSelection = new GridRowSelectionMixin(this);
    }

    protected override List<SlickColumn> GetColumns()
    {
        var columns = base.GetColumns();
        columns.Insert(0, GridRowSelectionMixin.CreateSelectColumn(() => rowSelection)
);
        return columns;
    }

    protected override List<ToolButton> GetButtons()
    {
        var buttons = base.GetButtons();

        buttons.Add(new ToolButton
        {
            CssClass = "tag-button",
            Title = "Do Something With Selected Rows",
            OnClick = delegate
            {
                var selectedIDs = rowSelection.GetSelectedKeys();
                if (selectedIDs.Count == 0)
                    Q.NotifyWarning("Please select some rows");
                else
                    Q.NotifySuccess("You have selected " + selectedIDs.Count +
                        " row(s) with ID(s): " + string.Join(", ", selectedIDs));
            }
        });

        return buttons;
    }
}
```

# How To: Setup Cascaded Editors

You might need multi-level cascaded editors like Country => City, Course => Class Name => Subject.

Starting with Serenity 1.8.2, it's rather simple. Lookup editors have this integrated functionality.

For versions before 1.8.2, it was also possible, and there was some samples in Serene, but you had to define some editor classes to make it work.

Let's say we have a database with three tables, Country, City, District:

- **Country Table:** CountryId, CountryName
- **City Table:** CityId, CityName, CountryId
- **District Table:** DistrictId, DistrictName, CityId

First make sure you generate code for all three tables using Seren, and you have a

`[LookupScript]` attribute on all of them:

```
[LookupScript("MyModule.Country")]
public sealed class CountryRow : Row...
{
    [DisplayName("Country Id"), Identity]
    public Int32? CountryId
    {
        get { return Fields.CountryId[this]; }
        set { Fields.CountryId[this] = value; }
    }

    [DisplayName("Country Name"), Size(50), NotNull, QuickSearch]
    public String CountryName
    {
        get { return Fields.CountryName[this]; }
        set { Fields.CountryName[this] = value; }
    }
}
```

```
[LookupScript("MyModule.City")]
public sealed class CityRow : Row...
{
    [DisplayName("City Id"), Identity]
    public Int32? CityId
    {
        get { return Fields.CityId[this]; }
        set { Fields.CityId[this] = value; }
    }

    [DisplayName("City Name"), Size(50), NotNull, QuickSearch]
    public String CityName
    {
        get { return Fields.CityName[this]; }
        set { Fields.CityName[this] = value; }
    }

    [DisplayName("Country"), ForeignKey("Country", "CountryId"), LookupInclude]
    public Int32? CountryId
    {
        get { return Fields.CountryId[this]; }
        set { Fields.CountryId[this] = value; }
    }
}
```



```
[LookupScript("MyModule.District")]
public sealed class DistrictRow : Row...
{
    [DisplayName("District Id"), Identity]
    public Int32? DistrictId
    {
        get { return Fields.DistrictId[this]; }
        set { Fields.DistrictId[this] = value; }
    }

    [DisplayName("District Name"), Size(50), NotNull, QuickSearch]
    public String DistrictName
    {
        get { return Fields.DistrictName[this]; }
        set { Fields.DistrictName[this] = value; }
    }

    [DisplayName("City"), ForeignKey("City", "CityId"), LookupInclude]
    public Int32? CityId
    {
        get { return Fields.CityId[this]; }
        set { Fields.CityId[this] = value; }
    }
}
```

Make sure you add `LookupInclude` attribute to `CityId` field of `DistrictRow`, and `CountryId` field of `CityRow`. We need them to be available at client side, otherwise they are not included by default in lookup scripts.

If you wanted to edit these fields as cascaded lookup editors in a form, e.g. `CustomerForm`, you would set them up like this:

```
[FormScript("MyModule.Customer")]
[BasedOnRow(typeof(Entities.CustomerRow))]
public class CustomerForm
{
    public String CustomerID { get; set; }
    public String CustomerName { get; set; }

    [LookupEditor(typeof(Entities.CountryRow))]
    public Int32? CountryId { get; set; }

    [LookupEditor(typeof(Entities.CityRow),
        CascadeFrom = "CountryId", CascadeField = "CountryId")]
    public Int32? CityId { get; set; }

    [LookupEditor(typeof(Entities.DistrictRow),
        CascadeFrom = "CityId", CascadeField = "CityId")]
    public Int32? DistrictId { get; set; }
}
```

You could also set these attributes in *CustomerRow*

Here, *CascadeFrom* attribute tells city editor, ID of the parent editor that it will bind to (cascade).

When this form is generated, *CountryId* field will be handled with an editor with ID *CountryId*, so we set *CascadeFrom* attribute of *CityId* lookup editor to that ID.

*CascadeField* determines the field to filter cities on. Thus, when country editor value changes, city editor items will be filtered on their *CountryId* properties like this:

```
this.Items.Where(x => x.CountryId == CountryEditorValue)
```

If *CascadeFrom* and *CascadeField* attributes are same, you only need to specify *CascadeFrom*, but i wanted to be explicit here.

If you wanted to add these cascaded editors to filter bar of customer grid, in *CreateToolbarExtensions* method of *CustomerGrid.cs*, do this:

```
AddEqualityFilter<LookupEditor>("CountryId",
    options: new LookupEditorOptions
    {
        LookupKey = "MyModule.Country"
    });

AddEqualityFilter<LookupEditor>("CityId",
    options: new LookupEditorOptions
    {
        LookupKey = "MyModule.City",
        CascadeFrom = "CountryId",
        CascadeField = "CountryId"
    });

AddEqualityFilter<LookupEditor>("DistrictId",
    options: new LookupEditorOptions
    {
        LookupKey = "MyModule.District",
        CascadeFrom = "CityId",
        CascadeField = "CityId"
    });
```

Here i suppose you have CountryId, CityId and DistrictId fields in CustomerRow.

Now you have useful cascaded editors for both editing and filtering.

# How To: Use Recaptcha

To use Recaptcha in login form, follow these steps:

Requires Serenity 1.8.5+

You might also use it for another form, but this is just a sample for login.

First, you need to register a new site for Recaptcha at:

<https://www.google.com/recaptcha/admin>

Once you have your site key, and secret key, enter them in web.config/appSettings section:

```
<add key="Recaptcha" value="{
  SiteKey: '6LeIxAcTAAAAAJcZVRqyHh71UMIEGNQ_MXjiZKhI',
  SecretKey: '6LeIxAcTAAAAAGG-vFI1TnRWxMZNFuojJ4WifJWe' }" />
```

The keys listed above are only for testing purposes. Never use them in production.

Edit LoginForm.cs to add a Recaptcha property:

```
public class LoginForm
{
    [Placeholder("default username is 'admin'")]
    public String Username { get; set; }
    [PasswordEditor, Placeholder("default password for admin user is 'serenity'"), Required(true)]
    public String Password { get; set; }
    [DisplayName(""), Recaptcha]
    public string Recaptcha { get; set; }
}
```

Edit LoginRequest.cs to add a Recaptcha property:

```
public class LoginRequest : ServiceRequest
{
    public string Username { get; set; }
    public string Password { get; set; }
    public string Recaptcha { get; set; }
}
```

Edit Login method under AccountPage.cs to validate the captcha server side:

```
[HttpPost, JsonFilter]
public Result<ServiceResponse> Login(LoginRequest request)
{
    return this.ExecuteMethod(() =>
    {
        request.CheckNotNull();

        if (string.IsNullOrEmpty(request.Username))
            throw new ArgumentNullException("username");

        var username = request.Username;

        // just add line below
        Serenity.Web.RecaptchaValidation.Validate(request.Recaptcha);

        if (WebSecurityHelper.Authenticate(ref username, request.Password, false))
            return new ServiceResponse();

        throw new ValidationError("AuthenticationError",
            Texts.Validation.AuthenticationError);
    });
}
```

## How To: Register Permissions in Serene

Serene shows a list of permissions in user and role permission dialogs. To show your own permissions there, you need to use these permissions with one of the attributes below:

- Attributes that derive from `PermissionAttributeBase`:
  - `ReadPermission`
  - `ModifyPermission`
  - `InsertPermission`
  - `UpdatePermission`
  - `DeletePermission`
- Page and Endpoint Access Control Attributes:
  - `PageAuthorize`
  - `ServiceAuthorize`

These attributes can be used with and located from one of these types:

- On top of `XYZRow` (Read, Write, Insert, Update, Delete permissions)
- On top of `XYZPage` and in action methods (`PageAuthorize`)
- On top of `XYZEndpoint` and in service actions (`ServiceAuthorize`)

When you use a permission key with one of such attributes, Serene will automatically discover them using reflection at application start.

There is a `PermissionKeys` class in Serene. Some users expected that when they write their permission keys in this class, they will be discovered.

But, `PermissionKeys` class is only there for intellisense purposes, it is ignored by Serene.

If you don't use a permission key with any of them but still want to show it in permission dialogs, you can use `RegisterPermission` attribute on assembly (write this anywhere in `YourProject.Web`):

```
[assembly: Serenity.ComponentModel.RegisterPermissionKey("MySpecialPermissionKey")]
```

## Organizing Permission Tree

To create permissions in tree hierarchy, use colon (:) as a separator in your permission keys:

- MyModule:SubModule:General
- MyModule:SubModule:Permission1
- MyModule:SubModule:Permission2

These keys will be shown under MyModule / SubModule category. Thus their category keys will be:

- MyModule:SubModule:

Category keys ends with colon. Don't use permission keys that ends with colon.

Please don't use permission keys that matches category keys. If you use such keys, for example *MyModule:SubModule* it won't be shown under *MyModule / SubModule* category but next to it at same level.

If you need a generic permission for such a category, use something like *MyModule:SubModule:General*.

*General* has no special meaning, you can use Common, Module, View, whatever you like.

## Handling Category Display Texts

As categories are automatically determined from permission keys, they don't have a user friendly display text for them.

You need to add display texts for them using localization system.

If you don't need localization, just add texts to `site.texts.invariant.json`

For example in `site.texts.invariant.json` file, there are such keys:

```
"Permission.Administration:": "Administration",
"Permission.Administration:Security": "User, Role Management and Permissions",
"Permission.Administration:Translation": "Languages and Translations",
"Permission.Northwind:Customer:": "Customers",
"Permission.Northwind:Customer:View": "View",
"Permission.Northwind:Customer:Delete": "Delete",
"Permission.Northwind:Customer:Modify": "Create/Update",
"Permission.Northwind:General": "[General]"
```

The keys ending with colon (:), like *Administration:* and *Customer:* corresponds to categories and these are their display texts.

You need to add texts for categories to invariant language at minimum. You may also add to other languages, if you want localization.





# How To: Use a Third Party Plugin With Serenity

To use a third party / custom plugin with a Serenity application involves no special steps. You may include their scripts and CSS in `_LayoutHead.cshtml`, and follow their documentation.

Especially if you are using TypeScript, there are no special steps involved.

In case of Saltaralle (which is being deprecated), you might have to write some import classes or use dynamic otherwise.

But, if you want that component to work well among other Serenity editors in dialogs, you may try wrapping it into a Serenity widget.

Here we'll take Bootstrap MultiSelect plugin as a sample, and integrate it as an editor into Serenity, similar to LookupEditor.

Here is the documentation and samples for this component:

<http://davidstutz.github.io/bootstrap-multiselect/>

## Getting Script and CSS Files

First we should download its script and CSS files and place them in correct places under `MyProject.Web/scripts/` and `MyProject.Web/content` folders.

This component has a NuGet package but unfortunately it is not in a standard fashion (it doesn't place files into your project folders), so we'll have to download files manually.

Download this script file and put it under `MyProject.Web/Scripts`:

<https://raw.githubusercontent.com/davidstutz/bootstrap-multiselect/master/dist/js/bootstrap-multiselect.js>

Download this CSS file and put it under `MyProject.Web/Content`:

<https://raw.githubusercontent.com/davidstutz/bootstrap-multiselect/master/dist/css/bootstrap-multiselect.css>

## Including Script/Css Files in `_LayoutHead.cshtml`

According to plugin documentation, we should include these files:

```
<!-- Include the plugin's CSS and JS: -->
<script type="text/javascript"
  src="js/bootstrap-multiselect.js">
</script>
<link rel="stylesheet" type="text/css"/
  href="css/bootstrap-multiselect.css" />
```

Open `_LayoutHead.cshtml` under `MyProject.Web/Views/Shared` and include these files:

```
// ...
@Html.Stylesheet("~/Content/bootstrap-multiselect.css")
@Html.Stylesheet("~/Content/serenity/serenity.css")
@Html.Stylesheet("~/Content/site/site.css")
// ...
@Html.Script("~/Scripts/bootstrap-multiselect.js")
@Html.Script("~/Scripts/Site/Serene.Script.js")
@Html.Script("~/Scripts/Site/Serene.Web.js")
```

## Creating BSMultiSelectEditor.ts

Now we need a TypeScript source file to hold our component. We should put it under `MyProject.Web/Scripts` or `MyProject.Web/Modules` directories.

I'll create it under `MyProject.Web/Modules/Common/Widgets` (first you need to create folder `Widgets`)

Create file `BSMultiSelectEditor.ts` at this location:

```

namespace MyProject {
  @Serenity.Decorators.element("<select/>")
  @Serenity.Decorators.registerClass(
    [Serenity.IGetEditValue, Serenity.ISetEditValue])
  export class BSMultiSelectEditor
    extends Serenity.Widget<BSMultiSelectOptions>
    implements Serenity.IGetEditValue, Serenity.ISetEditValue {

    constructor(element: JQuery, opt: BSMultiSelectOptions) {
      super(element, opt);
    }

    public setEditValue(source: any,
      property: Serenity.PropertyItem): void {
    }

    public getEditValue(property: Serenity.PropertyItem,
      target: any): void {
    }
  }

  export interface BSMultiSelectOptions {
    lookupKey: string;
  }
}

```

Here we defined a new editor type, deriving from `Widget`. Our widget takes options of type `BSMultiSelectOptions`, which contains a `lookupKey` option, similar to a `LookupEditor`. It also implements `IGetEditValue` and `ISetEditValue` TypeScript interfaces (this is different than C# interfaces)

```
@Serenity.Decorators.element("<select/>")
```

With above line, we specified that our widget works on a `SELECT` element, as this bootstrap multiselect plugin requires a select element too.

```
@Serenity.Decorators.registerClass(
  [Serenity.IGetEditValue, Serenity.ISetEditValue])
```

Above, we register our TypeScript class, with Saltaralle type system and specify that our widget implements custom value getter and setter methods, corresponding to `getEditValue` and `setEditValue` methods.

Here syntax is a bit terse as we have to handle interop between Saltaralle and TypeScript.

Our constructor and `getEditValue`, `setEditValue` methods are yet empty. We'll fill them in soon.

## Using Our New Editor

Now, build your project and transform templates.

Open CustomerRow.cs and locate Representatives property:

```
[LookupEditor(typeof(EmployeeRow), Multiple = true), NotMapped]
[LinkingSetRelation(typeof(CustomerRepresentativesRow),
    "CustomerId", "EmployeeId")]
public List<Int32> Representatives
{
    get { return Fields.Representatives[this]; }
    set { Fields.Representatives[this] = value; }
}
```

Here we see that Representatives uses a LookupEditor with multiple option true. We'll replace it with our brand new editor:

```
[BSMultiSelectEditor(LookupKey = "Northwind.Employee"), NotMapped]
[LinkingSetRelation(typeof(CustomerRepresentativesRow),
    "CustomerId", "EmployeeId")]
public List<Int32> Representatives
{
    get { return Fields.Representatives[this]; }
    set { Fields.Representatives[this] = value; }
}
```

## Populating Editor With Lookup Items

If you now build your project and open a Customer dialog, you'll see an empty SELECT in place of Customer representatives field.

Let's first fill it with data:

```
export class BSMultiSelectEditor {
    constructor(element: JQuery, opt: BSMultiSelectOptions) {
        super(element, opt);

        let lookup = Q.getLookup(this.options.lookupKey) as Q.Lookup<any>;
        for (let item of lookup.get_items()) {
            let key = item[lookup.get_idField()];
            let text = item[lookup.get_textField()] || '';
            Q.addOption(element, key, text);
        }
    }
}
```

We first get a reference to lookup object specified by our *lookupKey* option.

Lookups has `idField` and `textField` properties, which usually corresponds to fields determined by `IIdRow` and `INameRow` interfaces on your lookup row.

We enumerate all items in lookup and determine key and text properties of those items, using `idField` and `textField` properties.

Now save file, and open Customer dialog again. You'll see that this time options are filled.

## Bootstrap Multi Select Typings

According to documentation we should now call `".multiselect()"` jQuery extension on our select element.

I would cast our SELECT element to `<any>` and call `.multiselect` on it, but i want to write a TypeScript `.d.ts` definition file to reuse `multiselect` with intellisense.

So, under `MyProject.Web/Scripts/typings/bsmultiselect` folder, create a file, `bsmultiselect.d.ts`

```
interface JQuery {
    multiselect(options?: BSMultiSelectOptions | string): JQuery;
}

interface BSMultiSelectOptions {
    multiple?: boolean;
    includeSelectAllOption?: boolean;
    selectAllText?: string;
    selectAllValue?: string | number;
}
```

Here, i have extended JQuery interface which belongs to jQuery itself and is defined in `jquery.d.ts`. In TypeScript you can extend any interface with new methods, properties etc.

I used plugin documentation to define `BSMultiSelectOptions`. The plugin actually has much more options, but for now i keep it short.

## Creating Bootstrap MultiSelect on Our Editor

Now i'll go back to our constructor and initialize a `multiselect` plugin on it:

```

export class BSMultiSelectEditor {
  constructor(element: JQuery, opt: BSMultiSelectOptions) {
    super(element, opt);

    element.attr('multiple', 'multiple')

    let lookup = Q.getLookup(this.options.lookupKey) as Q.Lookup<any>;
    for (let item of lookup.get_items()) {
      let key = item[lookup.get_idField()];
      let text = item[lookup.get_textField()] || '';
      Q.addOption(element, key, text);
    }

    element
      .attr('name', this.uniqueName + "[]")
      .multiselect();
  }
}

```

Open CustomerDialog and you'll see that Representatives has our bootstrap multi select editor.

## Handling GetEditValue and SetEditValue Method

If we don't handle these methods, Serenity won't know how to read or set your editor value, so even if you select some representatives, next time you open the dialog, you'll have empty selections.

```

export class BSMultiSelectEditor {
  //...

  public setEditValue(source: any, property: Serenity.PropertyItem): void {
    this.element.val(source[property.name] || []).multiselect('refresh');
  }

  public getEditValue(property: Serenity.PropertyItem, target: any): void {
    target[property.name] = this.element.val() || [];
  }
}

```

setEditValue is called when editor value needs to be setted. It takes a *source* object, which is usually your entity being loaded in a dialog.

Property parameter is a PropertyItem object that contains details about the field being handled, e.g. our Representatives property. It's *name* field contains field name, e.g. *Representatives*.

Here we have to call `multiselect('refresh')` after setting select value, as multiselect plugin can't know when selections are changed.

`getEditValue` is opposite. It should read edit value and set it in target entity.

Ok, now our custom editor should be working fine.

# How To: Enable Script Bundling

In Serene template there are about 3MB+ of javascript files which are included by default in `_LayoutHead.cshtml`.

This might cause bandwidth and performance problems for some systems, especially when a Serenity based site is accessed from mobile devices.

There are several ways to handle these problems, like minification and gzipping to decrease script size, bundling to pack scripts into fewer files, thus reduce number of requests.

You might prefer to use tools like Webpack, Grunt, Gulp, UglifyJS etc, but in case you want a simpler and effective solution with much less manual steps, Serenity comes with a script bundling and minification / compression system out of the box.

Please note that this feature requires Serenity 2.0.13+

## ScriptBundles.json

First, you need a *ScriptBundles.json* file under *MyProject.Web/scripts/site* folder.

ScriptBundles.json configures which script bundle will contain which files when bundling is turned on.

This file is included by default in Serene template 2.0.13+ and looks like this:

Unless you want to add some custom scripts to bundles, you don't need to modify this file.



```
{
  "Libs": [
    "~/Scripts/pace.js",
    "~/Scripts/rsvp.js",
    "~/Scripts/jquery-{version}.js",
    "~/Scripts/jquery-ui-{version}.js",
    "~/Scripts/jquery-ui-i18n.js",
    "~/Scripts/jquery.validate.js",
    "~/Scripts/jquery.blockUI.js",
    "~/Scripts/jquery.cookie.js",
    "~/Scripts/jquery.json.js",
    "~/Scripts/jquery.autoNumeric.js",
    "~/Scripts/jquery.colorbox.js",
    "~/Scripts/jquery.dialogextendQ.js",
    "~/Scripts/jquery.event.drag.js",
    "~/Scripts/jquery.scrollintoview.js",
    "~/Scripts/jsrender.js",
    "~/Scripts/select2.js",
    "~/Scripts/toastr.js",
    "~/Scripts/SlickGrid/slick.core.js",
    "~/Scripts/SlickGrid/slick.grid.js",
    "~/Scripts/SlickGrid/slick.groupitemmetadataprovider.js",
    "~/Scripts/SlickGrid/Plugins/slick.autotooltips.js",
    "~/Scripts/SlickGrid/Plugins/slick.headerbuttons.js",
    "~/Scripts/SlickGrid/Plugins/slick.rowselectionmodel.js",
    "~/Scripts/SlickGrid/Plugins/slick.rowmovemanager.js",
    "~/Scripts/bootstrap.js",
    "~/Scripts/Saltarelle/mscorlib.js",
    "~/Scripts/Saltarelle/linq.js",
    "~/Scripts/Serenity/Serenity.CoreLib.js",
    "~/Scripts/Serenity/Serenity.Script.UI.js",
    "~/Scripts/Serenity/Serenity.Externals.js",
    "~/Scripts/Serenity/Serenity.Externals.Slick.js",
    "~/Scripts/jquery.cropzoom.js",
    "~/Scripts/jquery.fileupload.js",
    "~/Scripts/jquery.iframe-transport.js",
    "~/Scripts/jquery.slimscroll.js",
    "~/Scripts/mousetrap.js",
    "~/Scripts/fastclick/fastclick.js"
  ],
  "Site": [
    "~/Scripts/adminlte/app.js",
    "~/Scripts/Site/Serene.Script.js",
    "~/Scripts/Site/Serene.Web.js"
  ]
}
```

Here we define two distinct bundles, *Libs* and *Site*, corresponding to *Bundle.Libs.js* and *Bundle.Site.js* dynamic script files.

*Bundle.Site.js* is configured to contain these three JS files (in the listed order):

```
"~/Scripts/adminlte/app.js",  
"~/Scripts/Site/Serene.Script.js",  
"~/Scripts/Site/Serene.Web.js"
```

While *Bundle.Libs.js* contains all other javascript files.

Here we used 2 bundles by default, but it is possible to use one, three or more in case you need a different configuration. Just be careful about dependencies.

Here, the ordering inside a bundle (package) is very important. You must include scripts in the order they appear in your *\_LayoutHead.cshtml*.

When you will add another custom script, make sure that it is placed after all its dependencies.

For example, if you include a jquery plugin before jquery is loaded itself, you'll have errors.

Also make sure that you don't include same file in two bundles.

## Enabling Bundling

You should enable bundling (especially minification) only in production. Otherwise it might become very difficult to debug Javascript.

To enable bundling, just change *Enabled* property of *ScriptBundling* application setting in your web.config to true:

```
<add key="ScriptBundling" value="  
  { Enabled: true, Minimize: false, UseMinJS: false }" />
```

When *Enabled* is false (default) system will do nothing, and you'll work as usual with your script includes. And your page source looks like this:

```
<script src="/Scripts/pace.js?v=..."></script>
<script src="/Scripts/rsvp.js?v=..."></script>
<script src="/Scripts/jquery-2.2.3.js?v=..."></script>
<script src="/Scripts/jquery-ui-1.11.4.js?v=..."></script>
<script src="/Scripts/jquery-ui-i18n.js?v=..."></script>
...
...
...
<script src="/Scripts/adminlte/app.js?v=..."></script>
<script src="/Scripts/Site/Serene.Script.js?v=..."></script>
<script src="/Scripts/Site/Serene.Web.js?v=..."></script>
...
```

When *Enabled* is true, it will become like this one:

```
<script src="/DynJS.axd/Bundle.Libs.js?v=..."></script>
<script src="/DynJS.axd/Bundle.Site.js?v=..."></script>
...
```

These two bundles are generated in memory and contains all other script files configured in ScriptBundles.json file.

They are also compressed with GZIP and cached in memory (in gzipped form), so now our scripts will consume much less bandwidth and will cause fewer requests to server.

Now our script files will consume 600KB, instead of 3000KB before, a %80 reduction, not bad...

## Enabling Minification

After enabling bundling, you may also enable minification of scripts with the same web.config setting. Set *Minimize* property to true:

```
<add key="ScriptBundling" value="
  { Enabled: true, Minimize: true, UseMinJS: false }" />
```

Please note that *Minimize* property only works when *Enabled* is true, thus when bundling is enabled.

UglifyJS library is used for minification. This will be applied before bundling / gzipping so our bundles will become about %40 smaller, but will be much harder to read, so enable this only in production.

Now our bundled and minified script files will consume 375KB, instead of 3000KB before, a %87 reduction, or 1/8 the initial size.

## UseMinJS Setting

Minification might take some time, and first request to your site might take around 5-40 seconds more, depending on speed of your server.

Other requests will not be affected as minification is only performed once at application start.

Anyway, if you still need more performance at first request, you may ask Serenity to reuse already minified files in disk, if they are available.

Set *UseMinJS* to true:

```
<add key="ScriptBundling" value="{ Enabled: true, Minimize: true, UseMinJS: true }" />
```

When this setting is ON, before minifying a file, let's say *jquery-ui-1.11.4.js*, Serenity will first check to see if a *jquery-ui-1.11.4.min.js* already exists in disk. If so, instead of minifying with UglifyJS, it will simply use that file. Otherwise, it will run UglifyJS.

Serene comes with minified versions of almost all libraries, including Serenity scripts, so this setting will speed up initial start time.

There is a little risk that you should be careful about. If you manually modify a library script, make sure you minify it manually and modify its *.min.js* file too, otherwise when bundling is enabled an old version of that script might run at production.

## How Serenity Modifies Your *\_LayoutHead.cshtml* Includes?

If you look at your *\_LayoutHead.cshtml* you might spot lines like these:

```
@Html.Script("~/Scripts/jquery.cropzoom.js")  
@Html.Script("~/Scripts/jquery.fileupload.js")  
@Html.Script("~/Scripts/jquery.iframe-transport.js")
```

When bundling is disabled, these statements generates such HTML code:

```
<script src="/Scripts/jquery.cropzoom.js"></script>  
<script src="/Scripts/jquery.fileupload.js"></script>  
<script src="/Scripts/jquery.iframe-transport.js"></script>
```

Html.Script is an extension method of Serenity, so when bundling is enabled, instead of generating this HTML code, Serenity will first check to see if this script is included in a bundle.

For the first script that is included in a bundle, let's say Bundle.Lib.js, Serenity will generate code below:

```
<script src="/DynJS.axd/Bundle.Libs.js?v=..."></script>
```

But, for other Html.Script calls that is included in same bundle, Serenity will generate nothing. Thus, even though you call Html.Script 50 times, you'll get only one `<script>` output in page code.

## What Is v=p53uqJ... In My Script Tags?

This is a version number, or HASH of your script. Whether bundling is enabled or not, when you use Html.Script, it will add these hash to your script includes. This hash allows browser to cache script until it changes. When content of a script changes, its hash will change too, so browser won't cache and use an older version.

This is the reason you'll never have script caching problems with Serenity apps.

# How To: Debugging with Serenity Sources

Sometimes you might want to debug (or trace into) Serenity sources. There are two ways to do this.

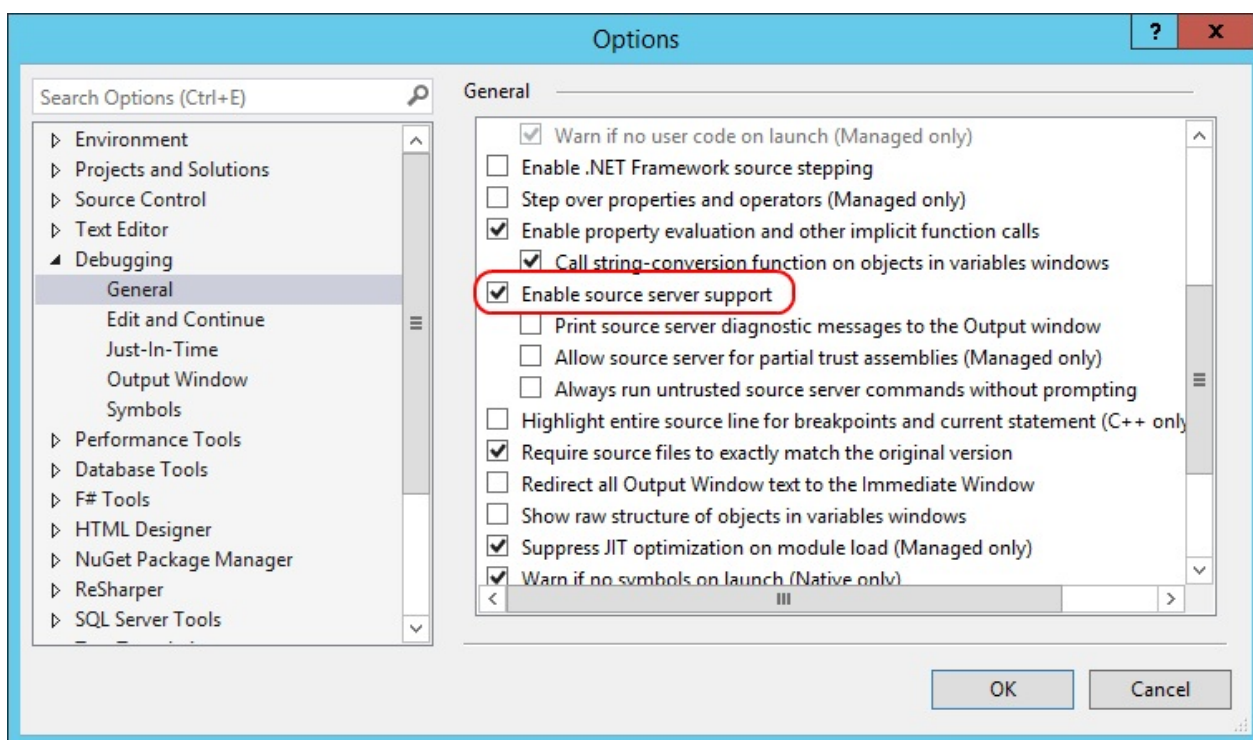
## By Enabling Source Server Support

Serenity NuGet packages already contains .pdb files debugging, which are modified to use GitHub as a symbol source by using excellent GitLink project:

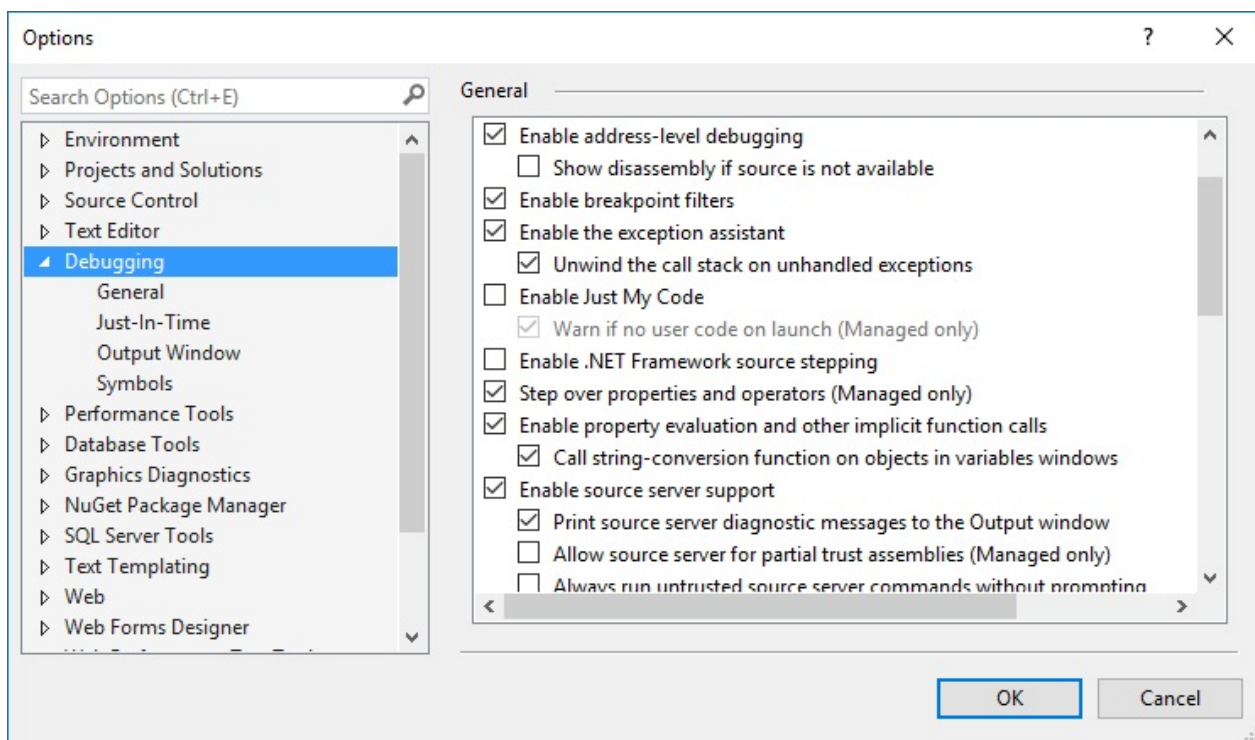
<https://github.com/GitTools/GitLink>

You don't need GitLink to debug, it's just a tool used by Serenity while publishing

To enable source server support, just go to your Visual Studio options, and under Debugging -> General, click **Enable source server support**.



You should also **uncheck** *Enable Just My Code*:



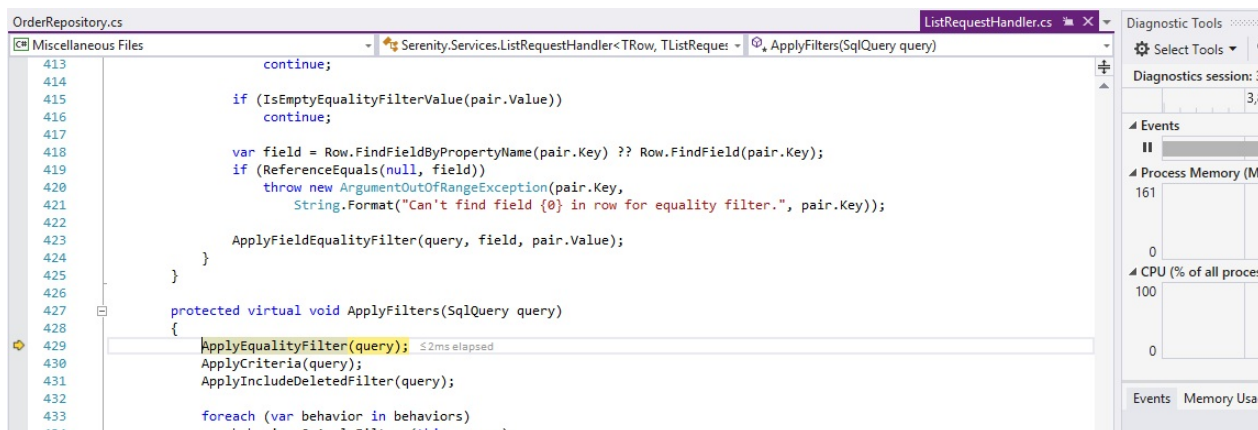
Now put a breakpoint on OrderRepository -> MyListHandler -> ApplyFilters or some other code you like:

```

OrderRepository.cs
Serenite.Web
44 private class MySaveHandler : SaveRequestHandler<MyRow> { }
45     -references
46 private class MyDeleteHandler : DeleteRequestHandler<MyRow> { }
47     -references
48 private class MyUndeleteHandler : UndeleteRequestHandler<MyRow> { }
49     -references
50 private class MyListHandler : ListRequestHandler<MyRow, OrderListRequest>
51 {
52     -references
53     protected override void ApplyFilters(SqlQuery query)
54     {
55         base.ApplyFilters(query);
56
57         if (Request.ProductID != null)
58         {
59             var od = Entities.OrderDetailRow.Fields.As("od");
60
61             query.Where(Criteria.Exists(
62                 query.SubQuery()
63                     .Select("1")
64                     .From(od)
65                     .Where(
66                         od.OrderID == fld.OrderID &
67                         od.ProductID == Request.ProductID.Value)
68                     .ToString()));
69         }
70     }
71 }

```

Launch application in debug mode, navigate to Orders page, and enjoy debugging:



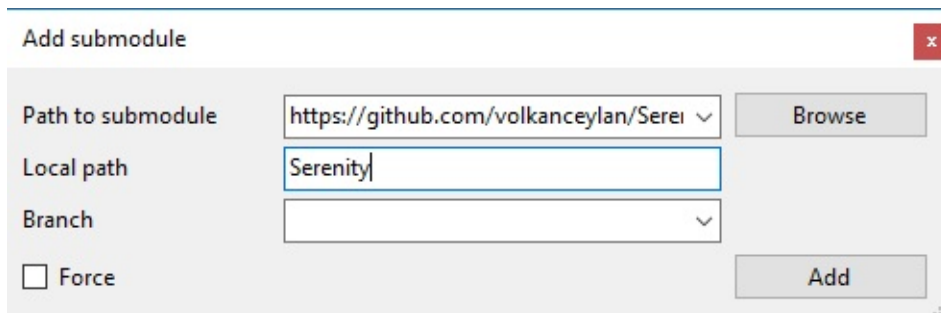
## By Adding Serenity as a SubModule

This option is only recommended for advanced users with a good knowledge of Git, Submodules and .NET in general. You'll also lose the ability to update Serenity and related files simply with NuGet.

I don't recommend this to novice users. If you do this and break your project, sorry but i can't help you.

I assume you have a project named *SereneSample*, and have a GIT repository for it already.

In *GitExtensions*, enter *Repository -> SubModules -> Add submodule*:



Under *Path to submodule* enter:

<https://github.com/volkanceylan/Serenity.git>

Enter *Serenity* as *Local path*.

Then click *Add* to add *Serenity* as a submodule to your repository. Then close the submodules dialog, and return to Visual Studio.

Expand your project references for *SereneSample.Web* and remove following references:



```
Serenity.Core  
Serenity.Data  
Serenity.Data.Entity  
Serenity.Services  
Serenity.Web
```

Right click your solution, click Add -> Existing Project and select Serenity.Core.csproj under Serenity folder.

Repeat it for Serenity.Data, Serenity.Data.Entity, Serenity.Services and Serenity.Web.

Right click your project references, click Add Reference -> Projects -> Solution and check all projects we added above, then click OK.

Now build your solution. There should be no errors.

Unload your project by right clicking it and clicking *Unload*. Then again right click project name and click *Edit*.

Add *Import* statement below, after the last *Import Project* statement in your csproj (there should be 4 *Import Project* statements, 5 after including this one):

```
<Import Project="$(SolutionDir)Serenity\tools\Submodule\Serenity.Submodule.Web.targets" />
```

Under *CompileSiteLess* include this:

```
<Exec Command="&quot;$(ProjectDir)tools\node\lessc.cmd&quot;  
&quot;$(ProjectDir)..\..\Serenity\Serenity.Web\Style\serenity.less&quot;  
&gt; &quot;$(ProjectDir)Content\serenity\serenity.css&quot;">  
</Exec>  
...
```

Save file and reload project.

Now you can use Serenity as a submodule and debug normally.

# Frequently Asked Questions

## Code Generator (Sergen)

### **Should I regenerate code after adding fields to my table:**

It's recommended to only generate code once. You should add new fields to row, column and form classes manually, taking existing fields as a sample.

But if you made too many changes, and want to generate code again it is possible. Sergen will launch Kdiff3 to let you merge changes, so that it won't override the changes you might have made to the code generated before.

---

### **I'm having an error in Sergen about KDiff3. Where to set its location:**

Sergen looks for KDiff3 at its default location under Program Files directory. Install it if you didn't yet.

If Kdiff3 is at another location, edit Serenity.CodeGenerator.config in your solution directory. This is a JSON file containing settings and preferences for Sergen.

## Permissions

### **I want to separate INSERT permission from UPDATE permission :**

Instead of [ModifyPermission] attribute use [InsertPermission] and [UpdatePermission] attributes on rows.

By default, for INSERT, save handler looks for these permissions on row in this order on row:

- 1) InsertPermission
- 2) ModifyPermission
- 3) ReadPermission

Only the first one that is found is checked.

Similarly for UPDATE, save handler looks for these permissions in order on row:

- 1) UpdatePermission

- 2) ModifyPermission
- 3) ReadPermission

For DELETE, delete handler looks for these permissions in order on row:

- 1) DeleteInsertPermission
- 2) ModifyPermission
- 3) ReadPermission

For LIST / RETRIEVE, only one permission is checked:

- 1) ReadPermission

## Publishing and Deployment

### How can i publish Serenity applications:

Serenity applications are x-copy deployable. You just need to setup connection strings after deployment. You might exclude source files from deployment.

Make sure you remove database migration safety check from *RunMigrations* method in *SiteInitialization.Migrations* file.

You can also use publish feature in Visual Studio. Make sure build action for all content files that you use are set to *Content*, and not *None*.

You have to only publish MyApplication.Web, not script project.

Serenity uses a NuGet version of ASP.NET MVC, so there is no need to install MVC on server. If you get some DLL missing error, check that its *Copy Local* option of VS project references is set to *True*.

## Forms and Editors

### How to allow negative values in DecimalEditor:

In *DecimalEditor* attribute set *MinValue* and *MaxValue* properties:

```
[DecimalEditor(MinValue = "-999999999.99", MaxValue = "999999999.99")]  
public Decimal MyProperty { get; set; }
```

Make sure you use same number of digits for min and max value.

### How can i reload/refresh a lookup editor data

Use `Q.ReloadLookup("MyModule.MyLookupKey")` to reload a lookup by its key.

### How to create filter editor for an Enum:

```
AddEqualityFilter<EnumEditor>(SomeRow.Fields.TheEnumField,  
    options: new EnumEditor { EnumKey = "MyModule.MyEnumType" });
```

---

### How to set current date in a date editor in new record mode:

Add `[DefaultValue("today")]` for date, or `[DefaultValue("now")]` for date time editor in form declaration.

Don't do this in row. It may cause errors.

Another option is to do this in dialog, overriding `AfterLoadEntity`:

```
form.MyDateField.AsDate = JsDate.Today;
```

# Troubleshooting

## Initial Setup

**After you create a new Serene application and launch it, login screen doesn't show and you see an error message in console that says *Template.LoginPanel is not found*:**

You probably used an invalid solution name, like MyProject.Something that contains dot (.)

Template system might not be able to locate templates when projects are named this way.

Please don't use dot in solution name. You may rename solution after creation if required.

## Compilation Errors

**I'm getting several *ambiguous reference* errors after adding a file to Script project :**

Remove *System.dll* reference from script project. Visual Studio adds this reference when you use *Add New File* dialog. Saltarelle Compiler doesn't work with such references, as it has a completely different runtime.

Please use copy/paste to create code files in Script project.

**Error: System.ComponentModel.DisplayName attribute exists in both ...\\Serenity.Script.UI.dll and ...\\v2.0...\\System.dll**

Same as above, remove *System.dll* reference from script project.

## Runtime Errors

**I'm getting *NotImplementException* when uploading files, or adding notes:**

Such features requires a table with integer identity column. String/Guid primary key support is added in recent Serenity versions, and some old behaviors doesn't work with such keys.

## SQL and Connections

**When i change page in grid, i'm getting error, "Incorrect syntax near 'OFFSET'. Invalid usage of the option NEXT in the FETCH statement:**

Your SQL server version is 2008 or older. By default, SQL Server connections use SQL2012 dialect. Do something like below for your connections in SiteInitialization.cs and your dialect for all to SqlServer2005 or SqlServer2008:

```
SqlConnection.GetConnectionString("Default").Dialect =  
    SqlServer2008Dialect.Instance;
```

## T4 Template Problems

### **My enum is not transferred to script side, after transforming templates:**

If you use an enum type in a row or service request / response it will be transferred, otherwise it won't by default. If you still want to include this enum, add [ScriptInclude] attribute on top of it.

## Editors and Forms

### **Tried to setup cascaded dropdowns but my second dropdown is always empty:**

Make sure your CascadeField is correct and it matches property name in secondary lookup properly. For example CountryID doesn't match CountryId at script side. You may use nameof() operator like CascadeField = nameof(CityRow.CountryId) to be sure.

A similar problem might occur if you fail to correctly set CascadeFrom option. This corresponds to first dropdown ID in your form. For example, if there are MyCountryId and CustomerCityId properties in the form, CascadeForm should be *CustomerCountryId*. Again, you can use nameof(MyCountryId) to be certain.

CascadeFrom is an editor ID in form, while CascadeField is a field property name in row.

Another possibility is that CascadeField is not included in lookup data that is sent to script side. For example, if second dropdown is city selection, which is connected to a country dropdown through CountryId, make sure that CountryId property in CityRow has a [LookupInclude] attribute on it. By default, only ID and Name properties are sent to script side for lookups.

---

### **Tried to create tabs using a dialog template, but my tab is not shown or empty:**

Make sure you don't put a tab content, inside another one, like DIV inside another tab DIV.

## Master/Detail Editing

**I created a in memory master detail editing similar to one in Movie Tutorial cast editor, but when i update a record, i'm getting duplicate entries:**

Make sure you don't have a [IdProperty] on your EditDialog class. As edit dialogs work in memory with records that doesn't yet have actual IDs, if you use your actual ID property with them, dialog will think that you are adding new records on update (as their actual ID value is always null).

As you see in code below, GridEditorDialog base class uses a fake ID:

```
[IdProperty("__id")]
public abstract class GridEditorDialog<TEntity> : EntityDialog<TEntity>
    where TEntity : class, new()
```

So when you put [IdProperty] to your edit dialog, you're overwriting this fake ID and causing unexpected behavior.

---

**I'm succesfully adding details but later when open an existing record, some view fields are empty:**

Please put [MinSelectLevel(SelectLevel.List)] on your view fields in YourDetailRow.cs. By default, List handlers and MasterDetailBehavior only loads table fields (not view fields) of detail rows.

## Permissions

**My page is not shown in navigation:**

Page access permissions are read from *PageAuthorize* attribute on *Index* action of *XYZPage.cs* file, which is your MVC page controller. Make sure you set this to a permission user has.

---

**I have added a permission to PermissionKeys.cs but it doesn't show in user permissions dialog:**

PermissionKeys class is just for intellisense purposes. See below for information about registering keys.

---

- [How To Register Permissions in Serene](#)
- 

**Changed permission keys on row, but i'm getting an error when i open the page, and no records displayed:**

Your XYZEndpoint.cs also has a `[ServiceAuthorize("SomePermission")]` on it. This is to provide a secondary level security. Replace permission key with the one on Row.

## Localization

**My localizations lost on live site after publishing:**

The translations you made using translation interface are saved to files under `~/App_Data` directory. Either copy these files to live server, or move texts in them to relevant files under `~/Scripts/site/texts`.

---

**I have added some custom local text keys but can't access them from script side:**

Not all translations are transferred to script side. There is a setting in `web.config` with `LocalTextPackages` key, that controls these prefixes. If you look there, you can see that only text keys that are starting with `Db.`, `Dialogs.`, `Forms.` etc are transferred to client side. This is to limit size of texts as not all of them are used in script code.

Either add your own prefix there, or change your keys to start with one of default prefixes.

---

## NuGet Packages and Updates

**I have some errors after updating Select2:**

Please don't update Select2 to a version later than 3.5.1. Recent versions has some known compability problems.

To revert to Select2 3.5.1, type following in package manager console:

```
Update-Package Select2.js -Version 3.5.1
```

## Deployment and Publishing

---



**After publishing project some content is not found, or not displayed:**

If you are using Visual Studio publish, make sure that css, image files etc are included in web project and their build action is set to content.

Another possibility is that IIS\_IUSRS user group can't access files. Check that it has permissions to files in published web folder.

---

**Table not found (e.g. User) errors after publish:**

Serene has a check to avoid running migrations on an arbitrary database. Find this check under *RunMigrations* method of *SiteInitialization.Migrations* file and remove it.

---

**FieldAccessExceptions with message "Cannot set a constant field" :**

Your hosting provider has set your web application pool to medium trust. Ask them to grant high trust, or if possible change provider.

It might be possible to change trust level in web.config if your hosting provider didn't lock it:

```
<configuration>
  <system.web>
    <trust level="Full" />
  </system.web>
</configuration>
```

Serenity initializes field objects with reflection. Under medium trust, it can't do that. You may try replacing all *\*public readonly* field declarations with *"public static"* in *\*Row.cs*, but not sure if this will resolve all problems.

ASP.NET has made Medium trust obsolete, and they won't fix any problems related to this anymore. See <http://stackoverflow.com/questions/16849801/is-trying-to-develop-for-medium-trust-a-lost-cause> It is strongly recommended to change your hosting provider

## Service Locator & Initialization

Serenity uses the service locator pattern to abstract its dependencies and make it possible to work with your chosen libraries and service providers.

For example, Serenity doesn't care about how you store your users, but it can query current user through an abstraction (`IAuthorizationService`, `IUserRetrieveService` etc.)

Similarly you may use Redis, Couchbase, Memcached or any other as distributed cache in your application. Serenity doesn't have a direct dependency on any of their client libraries. As soon as you implement `IDistributedCache` interface and register it with the service locator, Serenity will start working with your NoSQL database.

Some might argue that Service Locator is an anti-pattern that should be avoided. An alternative to it would be the Dependency Injection pattern. But it seems unlogical having to know about every dependency (and dependencies of dependencies...) of an object to just be able to use it (you shouldn't have to know about details of what your mobile operator uses to send a simple SMS). Maybe DI is a sample of over-engineering.

# Dependency Static Class

[namespace: *Serenity.Abstractions*, assembly: *Serenity.Core*]

Dependency class is the service locator of Serenity. All dependencies are queried through its methods:

```
public static class Dependency
{
    public static TType Resolve<TType>() where TType : class;
    public static TType Resolve<TType>(string name) where TType : class;
    public static TType TryResolve<TType>() where TType : class;
    public static TType TryResolve<TType>(string name) where TType : class;

    public static IDependencyResolver SetResolver(IDependencyResolver value);
    public static IDependencyResolver Resolver { get; }
    public static bool HasResolver { get; }
}
```

In your application's start method (e.g. in *global.asax.cs*) you should initialize service locator by setting a dependency resolver (*IDependencyResolver*) implementation (an IoC container) with *SetResolver* method.

## Dependency.SetResolver Method

Configures the dependency resolver implementation to use.

You can use IoC container of your choice but Serenity already includes one based on *Munq*:

```
var container = new MunqContainer();
Dependency.SetResolver(container);
```

*SetResolver* methods return previously configured *IDependencyResolver* implementation, or null if none is configured before.

## Dependency.Resolver Property

Returns currently configured *IDependencyResolver* implementation.

Throws a *InvalidProgramException* if none is configured yet.

## Dependency.HasResolver Property

Returns true if a `IDependencyResolver` implementation is configured through `SetResolver`. Returns false, if not.

## Dependency.Resolve Method

Returns the registered implementation for requested type.

If no implementation is registered, throws a *KeyNotFoundException*.

If no dependency resolver is configured yet, throw a *InvalidProgramException*

Second overload of `Resolve` method accepts a *name* parameter. This should be used if different providers are registered for an interface depending on scope.

For example, Serenity has a `IConfigurationRepository` interface that can have different providers based on setting scope. Some settings might be *Application* scoped (shared between all servers for this application), while some might be *Server* scoped (each server might use a different unique identifier).

So, to retrieve a `IConfigurationRepository` provider for each of these scopes, you would call the method like:

```
var appConfig = Dependency.Resolve<IConfigurationRepository>("Application");  
  
var srvConfig = Dependency.Resolve<IConfigurationRepository>("Server");
```

## Dependency.TryResolve Method

This is functionally equivalent to `Resolve` method with one difference.

If a provider is not registered for requested type, or no dependency resolver is configured yet, `TryResolve` doesn't throw an exception, but instead returns null.

# IDependencyResolver Interface

[namespace: *Serenity.Abstractions*, assembly: *Serenity.Core*]

This interface defines the contract for dependency resolvers which are usually IoC containers that handles mapping between services and providers.

```
public interface IDependencyResolver
{
    TService Resolve<TService>() where TService : class;
    TService Resolve<TService>(string name) where TService : class;
    TService TryResolve<TService>() where TService : class;
    TService TryResolve<TService>(string name) where TService : class;
}
```

All methods are functionally equal to corresponding methods in `Dependency` static class.

# IDependencyRegistrar Interface

[namespace: *Serenity.Abstractions*, assembly: *Serenity.Core*]

Dependency resolvers should implement this interface (IDependencyRegistrar) to register dependencies:

```
public interface IDependencyRegistrar
{
    object RegisterInstance<TType>(TType instance) where TType : class;
    object RegisterInstance<TType>(string name, TType instance) where TType : class;
    object Register<TType, TImpl>() where TType : class where TImpl : class, TType;
    object Register<TType, TImpl>(string name) where TType : class where TImpl : class
    , TType;
    void Remove(object registration);
}
```

MunqContainer and other IoC containers are also dependency registrars (they implement IDependencyRegistrar interface), so you just have to query for it:

```
var registrar = Dependency.Resolve<IDependencyRegistrar>();
registrar.RegisterInstance<ILocalTextRegistry>(new LocalTextRegistry());
registrar.RegisterInstance<IAuthenticationService>(...
```

## IDependencyRegistrar.RegisterInstance Method

Registers a singleton instance of a type (TType, usually an interface) as provider of that type.

```
object RegisterInstance<TType>(TType instance) where TType : class;
```

When you register an object instance with this overload, whenever an implementation of TType is requested, the instance that you registered is returned from dependency resolver. This is similar to *Singleton Pattern*.

```
var registrar = Dependency.Resolve<IDependencyRegistrar>();
registrar.RegisterInstance<ILocalTextRegistry>(new LocalTextRegistry());
```

If there was already a registration for TType, it is overridden.

This overload is the most used method of registering dependencies.

Make sure the provider which you registered is thread safe, as all threads will be using your instance at same time!

RegisterInstance has a less commonly used overload with a *name* parameter:

```
object RegisterInstance<TType>(string name, TType instance) where TType : class;
```

Using this overload, you can register different providers for the same interface, differentiated by a string key.

For example, Serenity has a IConfigurationRepository interface that can have different providers based on setting scope. Some settings might be *Application* scoped (shared between all servers for this application), while some might be *Server* scoped (each server might use a different unique identifier).

So, to register a IConfigurationRepository provider for each of these scopes, you would call the method like:

```
var registrar = Dependency.Resolve<IDependencyRegistrar>();

registrar.RegisterInstance<IConfigurationRepository>(
    "Application", new MyApplicationConfigurationRepository());

registrar.RegisterInstance<IConfigurationRepository>(
    "Server", new MyServerConfigurationRepository());
```

And when querying for these dependencies:

```
var appConfig = Dependency.Resolve<IConfigurationRepository>("Application");
// ...
var srvConfig = Dependency.Resolve<IConfigurationRepository>("Server");
// ...
```

## IDependencyRegistrar.Register Method

Unlike *RegisterInstance*, when a type is registered with this method, every time a provider for that type is requested, a new instance will be returned (so each requestor gets a unique instance).

```
var registrar = Dependency.Resolve<IDependencyRegistrar>();  
registrar.Register<ILocalTextRegistry, LocalTextRegistry>();
```

## IDependencyRegistrar.Remove Method

All registration methods of IDependencyRegistrar interface returns an object which you can later use to remove that registration.

Avoid using this method in ordinary applications as all registrations should be done from a central location and once per lifetime of the application. But this can be useful for unit test purposes.

```
var registrar = Dependency.Resolve<IDependencyRegistrar>();  
var registration = registrar.Register<ILocalTextRegistry, LocalTextRegistry>();  
//...  
registrar.Remove(registration);
```

This is not an undo operation. If you register type C for interface A, while type B was already registered for the same interface, prior registration is overridden and lost. You can't get back to prior state by removing registration of C.



# MunqContainer Class

[namespace: *Serenity*, assembly: *Serenity.Core*]

Serenity includes a slightly modified version of Munq IoC container (<http://munq.codeplex.com/>).

MunqContainer class implements both IDependencyResolver and IDependencyRegistrar interfaces (all containers should).

Once you set a MunqContainer instance as dependency resolver like:

```
var container = new MunqContainer();
Dependency.SetResolver(container);
```

You can access its registration interface by querying for IDependencyRegistrar interface:

```
var registrar = Dependency.Resolve<IDependencyRegistrar>();
```

Here, *registrar* is the same MunqContainer instance (*container*) that we created in prior sample.

If you would like to use another IoC container, you just have to create a class that implements IDependencyResolver and IDependencyRegistrar interfaces using your favorite IoC container.

# CommonInitialization Static Class

[namespace: *Serenity.Web*, assembly: *Serenity.Web*]

If you are going to use defaults in a web environment, instead of doing the service locator setup and some other configuration manually, you may just call *CommonInitialization.Run()* in your application start method. *CommonInitialization* registers default implementations for some of Serenity abstractions.

```
CommonInitialization.Run();
```

If there is already a provider registered for some abstraction, *CommonInitialization* doesn't override them.

This method contains calls to some other methods to initialize Serenity platform defaults:

```
public static class CommonInitialization
{
    public static void Run()
    {
        InitializeServiceLocator();
        InitializeSelfAssemblies();
        InitializeConfigurationSystem();
        InitializeCaching();
        InitializeLocalTexts();
        InitializeDynamicScripts();
    }

    public static void InitializeServiceLocator()
    {
        if (!Dependency.HasResolver)
        {
            var container = new MunqContainer();
            Dependency.SetResolver(container);
        }
    }

    //...
}
```

*CommonInitialization.InitializeServiceLocator* and other methods may also be used individually instead of calling *CommonInitialization.Run*.

*InitializeServiceLocator()*, registers a *MunqContainer* instance as the default *IDependencyResolver* implementation.



# Authentication & Authorization

Serenity uses some abstractions to work with your application's own user authentication and authorization mechanism.

```
Serenity.Abstractions.IAuthenticationService  
Serenity.Abstractions.IAuthorizationService  
Serenity.Abstractions.IPermissionService  
Serenity.Abstractions.IUserRetrieveService
```

As Serenity doesn't have default implementation for these abstractions, you should provide some implementation for them, using dependency registration system.

For example, Serenity Basic Application sample registers them in its `SiteInitialization.ApplicationStart` method like below:

```
var registrar = Dependency.Resolve<IDependencyRegistrar>();  
  
registrar.RegisterInstance<IAuthorizationService>(new Administration.AuthorizationService());  
  
registrar.RegisterInstance<IAuthenticationService>(new Administration.AuthenticationService());  
  
registrar.RegisterInstance<IPermissionService>(new Administration.PermissionService());  
  
registrar.RegisterInstance<IUserRetrieveService>(new Administration.UserRetrieveService());
```

You might want to have a look at those sample implementations before writing your own.

# IAuthenticationService Interface

[namespace: *Serenity.Abstractions*, assembly: *Serenity.Core*]

```
public interface IAuthenticationService
{
    bool Validate(ref string username, string password);
}
```

This is the service that you usually call from your login page to check if entered credentials are correct. Your implementation should return true if username/password pair matches.

A dummy authentication service could be written like this:

```
public class DummyAuthenticationService : IAuthenticationService
{
    public bool Validate(ref string username, string password)
    {
        return username == password;
    }
}
```

This service returns true, if username is equal to specified password (just for demo).

First parameter is passed by reference for you to change username to its actual representation in database before logging in. For example, the user might have entered uppercase `JOE` in login form, but actual username in database could be `Joe`. This is not a requirement, but if your database is case sensitive, you might have problems during login or later phases.

You might register this service from `global.asax.cs / SiteInitialization.ApplicationStart` like:

```
protected void Application_Start(object sender, EventArgs e)
{
    Dependency.Resolve<IDependencyRegistrar>()
        .RegisterInstance(new DummyAuthenticationService());
}
```

And use it somewhere in your login form:

```
void DoLogin(string username, string password)
{
    if (Dependency.Resolve<IAuthenticationService>()
        .Validate(ref username, password))
    {
        // FormsAuthentication.SetAuthenticationTicket etc.
    }
}
```

# IAuthorizationService Interface

[namespace: *Serenity.Abstractions*, assembly: *Serenity.Core*]

This is the interface that Serenity checks through to see if there is a logged user in current request.

```
public interface IAuthorizationService
{
    bool IsLoggedIn { get; }
    string Username { get; }
}
```

Some basic implementation for web applications could be:

```
using Serenity;
using Serenity.Abstractions;

public class MyAuthorizationService : IAuthorizationService
{
    public bool IsLoggedIn
    {
        get { return !string.IsNullOrEmpty(Username); }
    }

    public string Username
    {
        get { return WebSecurityHelper.HttpContextUsername; }
    }
}

/// ...

void Application_Start(object sender, EventArgs e)
{
    Dependency.Resolve<IDependencyRegistrar>()
        .RegisterInstance(new MyAuthorizationService());
}
```

# IPermissionService Interface

[namespace: *Serenity.Abstractions*, assembly: *Serenity.Core*]

A permission is some grant to do some action (entering a page, calling a certain service). In Serenity permissions are some string keys that are assigned to individual users (similar to ASP.NET roles).

For example, if we say some user has `Administration` permission, this user can see navigation links that requires this permission or call services that require the same.

You can also use composite permission keys like `ApplicationID:PermissionID` (for example `orders:Create`), but Serenity doesn't care about application ID, nor permission ID, it only uses the composite permission key as a whole.

```
public interface IPermissionService
{
    bool HasPermission(string permission);
}
```

You might have a table like...

```
CREATE TABLE UserPermissions (
    UserID int,
    Permission nvarchar(20)
}
```

and query on this table to implement this interface.

A simpler sample for applications where there is a `admin` user who is the only one that has the permission `Administration` could be:



```
using Serenity;
using Serenity.Abstractions;

public class DummyPermissionService : IPermissionService
{
    public bool HasPermission(string permission)
    {
        if (Authorization.Username == "admin")
            return true;

        if (permission == "Administration")
            return false;

        return true;
    }
}
```

# IUserDefinition Interface

[namespace: *Serenity*, assembly: *Serenity.Core*]

Most applications store some common information about a user, like ID, display name (nick / fullname), email address etc. Serenity provides a basic interface to access this information in an application independent manner.

```
public interface IUserDefinition
{
    string Id { get; }
    string Username { get; }
    string DisplayName { get; }
    string Email { get; }
    Int16 IsActive { get; }
}
```

Your application should provide a class that implements this interface, but not all of this information is required by Serenity itself. Id, Username and IsActive properties are minimum required.

**Id** can be an integer, string or GUID that uniquely identifies a user.

**Username** should be a unique username, but you can use e-mail addresses as username too.

**IsActive** should return 1 for active users, -1 for deleted users (if you don't delete users from database), and 0 for temporarily disabled (account locked) users.

**DisplayName** and **Email** are optional and currently not used by Serenity itself, though your application may require them.

# IUserRetrieveService Interface

[namespace: *Serenity.Abstractions*, assembly: *Serenity.Core*]

When Serenity needs to access IUserDefinition object for a given user name or user ID, it uses this interface.

```
public interface IUserRetrieveService
{
    IUserDefinition ById(string id);
    IUserDefinition ByUsername(string username);
}
```

In your implementation, it is a good idea to cache user definition objects, as a common WEB application might use this interface repeatedly for same user.

Serenity Basic Application sample has an implementation like below:

```
public class UserRetrieveService : IUserRetrieveService
{
    private static MyRow.RowFields fld { get { return MyRow.Fields; } }

    private UserDefinition GetFirst(IDbConnection connection, BaseCriteria criteria)
    {
        var user = connection.TrySingle<Entities.UserRow>(criteria);
        if (user != null)
            return new UserDefinition
            {
                UserId = user.UserId.Value,
                Username = user.Username,
                //...
            };

        return null;
    }

    public IUserDefinition ById(string id)
    {
        if (id.IsEmptyOrNull())
            return null;

        return TwoLevelCache.Get<UserDefinition>("UserByID_" + id, CacheExpiration.Never, CacheExpiration.OneDay, fld.GenerationKey, () =>
        {
            using (var connection = SqlConnections.NewByKey("Default"))
                return GetFirst(connection,
                    new Criteria(fld.UserId) == id.TryParseID32().Value);
        });
    }

    public IUserDefinition ByUsername(string username)
    {
        if (username.IsEmptyOrNull())
            return null;

        return TwoLevelCache.Get<UserDefinition>("UserByName_" + username, CacheExpiration.Never, CacheExpiration.OneDay, fld.GenerationKey, () =>
        {
            using (var connection = SqlConnections.NewByKey("Default"))
                return GetFirst(connection, new Criteria(fld.Username) == username);
        });
    }
}
```

# Authorization Static Class

[namespace: *Serenity*, assembly: *Serenity.Core*]

Authorization class provides some shortcuts to information which is provided by services like *IAuthorizationService*, *IPermissionService* etc.

For example, instead of writing

```
Dependency.Resolve<IAuthorizationService>().HasPermission("SomePermission")
```

you could use

```
Authorization.HasPermission("SomePermission")
```

```
public static class Authorization
{
    public static bool IsLoggedIn { get; }
    public static IUserDefinition UserDefinition { get; }
    public static string UserId { get; }
    public static string Username { get; }
    public static bool HasPermission(string permission);
    public static void ValidateLoggedIn();
    public static void ValidatePermission(string permission);
}
```

`IsLoggedIn` , `UserDefinition` , `UserId` , `Username` and `HasPermission` make use of corresponding service for you to access information easier about current user.

`ValidateLoggedIn` checks if there is a logged user and if not, throws a `ValidationException` with error code `NotLoggedIn` .

`ValidatePermission` checks if logged user has specified permission and throws a `ValidationException` with error code `AccessDenied` otherwise.

# Configuration System

.NET applications usually stores their configuration in app.config (desktop) or web.config (web) files.

Though, its common practice to store configuration in such files for web applications, sometimes it might be required to store some configuration in a database table to make it available to all servers in a web farm, and set it from one location.

Just like IsolatedStorage has scopes like Application, Machine, User etc, a configuration setting might have different scopes:

- Application - Shared between all servers that runs a web application
- Server - Applies to current server only
- User - Applies to current user only

Number of samples can be increased.

If you have just one server, *Application* and *Server* scopes can be stored in web.config file, but in a web farm setup, Application settings should be stored in a location accessible from all servers (database or shared folder).

User settings are usually stored in database along with a User ID.

Serenity provides an extensible configuration system.

# Defining Configuration Settings

In Serenity platform, configuration settings are just simple classes like:

```
[SettingScope("Application"), SettingKey("Logging")]
private class LogSettings
{
    public LoggingLevel Level { get; set; }
    public string File { get; set; }
    public int FlushTimeout { get; set; }
}
```

If required, default settings can be set in the class constructor.

## SettingScope Attribute

If specified, this attribute determines the scope of settings.

If not specified, default scope is *Application*.

## SettingKey Attribute

If specified, this attribute determines a key for settings (e.g. appSettings key for web.config) class.

If not specified, class name is used as the key.

# ICConfigurationRepository Interface

[namespace: *Serenity.Abstractions*, assembly: *Serenity.Core*]

All applications have some kind of configuration. Scope, storage medium and format for these settings are different from application to application, so Serenity provides IConfigurationRepository interface to abstract access to this configuration.

```
public interface IConfigurationRepository
{
    object Load(Type settingType);
    void Save(Type settingType, object value);
}
```

## ICConfigurationRepository.Load Method

This method returns an instance of settingType. Provider should check SettingKey attribute to determine key for the setting type.

If same provider is registered for multiple scopes, provider should also check for SettingScope attribute.

Provider should return an object instance, even if setting is not found (an object created with settingType's default constructor).

## ICConfigurationRepository.Save Method

Saves an instance of settingType. Provider should check SettingKey attribute to determine key for the setting type.

If same provider is registered for multiple scopes, provider should also check for SettingScope attribute.

This method is optional to implement, as you may not want settings to be changed. In this case, just throw a *NotImplementedException*.



# AppSettingsJsonConfigRepository

[namespace: *Serenity.Configuration*, assembly: *Serenity.Data*]

Most web applications store configuration settings in web.config file, under appSettings section.

Serenity provides a default implementation of IConfigurationRepository that uses appSettings as configuration store.

```
public class AppSettingsJsonConfigRepository : IConfigurationRepository
{
    public void Save(Type settingType, object value)
    {
        throw new NotImplementedException();
    }

    public object Load(Type settingType)
    {
        return LocalCache.Get("ApplicationSetting:" + settingType.FullName,
            TimeSpan.Zero, delegate()
            {
                var keyAttr = settingType.GetCustomAttribute<SettingKeyAttribute>();
                var key = keyAttr == null ? settingType.Name : keyAttr.Value;
                return JSON.Parse(ConfigurationManager.AppSettings[key].TrimToNull() ??
                    "{}", settingType);
            });
    }
}
```

To register this provider manually:

```
var registrar = Dependency.Resolve<IDependencyRegistrar>();
RegisterInstance<IConfigurationRepository>("Application",
    new AppSettingsJsonConfigRepository())
```

When you call *Serenity.Web.CommonInitialization.Run()*, it registers this class as the default provider for IConfigurationRepository (in *Application* scope), if another one is not already registered.

This provider expects settings to be defined in web.config / app.config file in JSON format:

```
<appSettings>
  <add key="Logging" value="{ File: '~\App_Data\Log\App_{0}_{1}.log',
    FlushTimeout: 0, Level: 'Debug' }" />
</appSettings>
```

Out of the box, Serenity contains this configuration provider only. You may take it as a sample, and write another one for your setup (load from database etc.).

It is a good idea to cache returned objects in your implementation to avoid deserialization costs every time settings are read.

# Config Static Class

[namespace: *Serenity*, assembly: *Serenity.Core*]

This is the central location to access your configuration settings. It contains shortcut methods to registered *IConfigurationRepository* provider.

```
public static class Config
{
    public static object Get(Type settingType);
    public static TSettings Get<TSettings>() where TSettings: class, new();
    public static object TryGet(Type settingType);
    public static TSettings TryGet<TSettings>() where TSettings : class, new();
}
```

## Config.Get Method

Used to read configuration settings for specified type.

If no provider is registered for setting type's scope, a *KeyNotFoundException* is raised.

If setting is not found, providers usually return a default instance.

Prefer generic overload to avoid having to cast the returned object.

```
if (Config.Get<LogSettings>().LoggingLevel != LogginLevel.Off)
{
    // ..
}
```

## Config.TryGet Method

Used to read configuration settings for specified type.

Functionally equivalent to *Get*, but while it throws an exception if no configuration provider is registered for the setting scope, *TryGet* returns *null*.

```
if ((Config.TryGet<LogSettings>() ?? new LogSettings()).LoggingLevel != LogginLevel.Of
f)
{
    // ..
}
```

Prefer this method over Get only to avoid exceptions when configuration system is not initialized yet.

Get works on safe-side, and is the recommended method to use.

# Localization

Most web applications must support a variety of languages. Sites like Youtube, Wikipedia, Facebook works in many languages.

First time a user visits such a site, a language for her is automatically chosen depending on the browser language (pre-determined by regional settings).

If automatic selection is not what they expect, users can set their preferred language and this selection is stored in a client-side cookie (or server side user profile table).

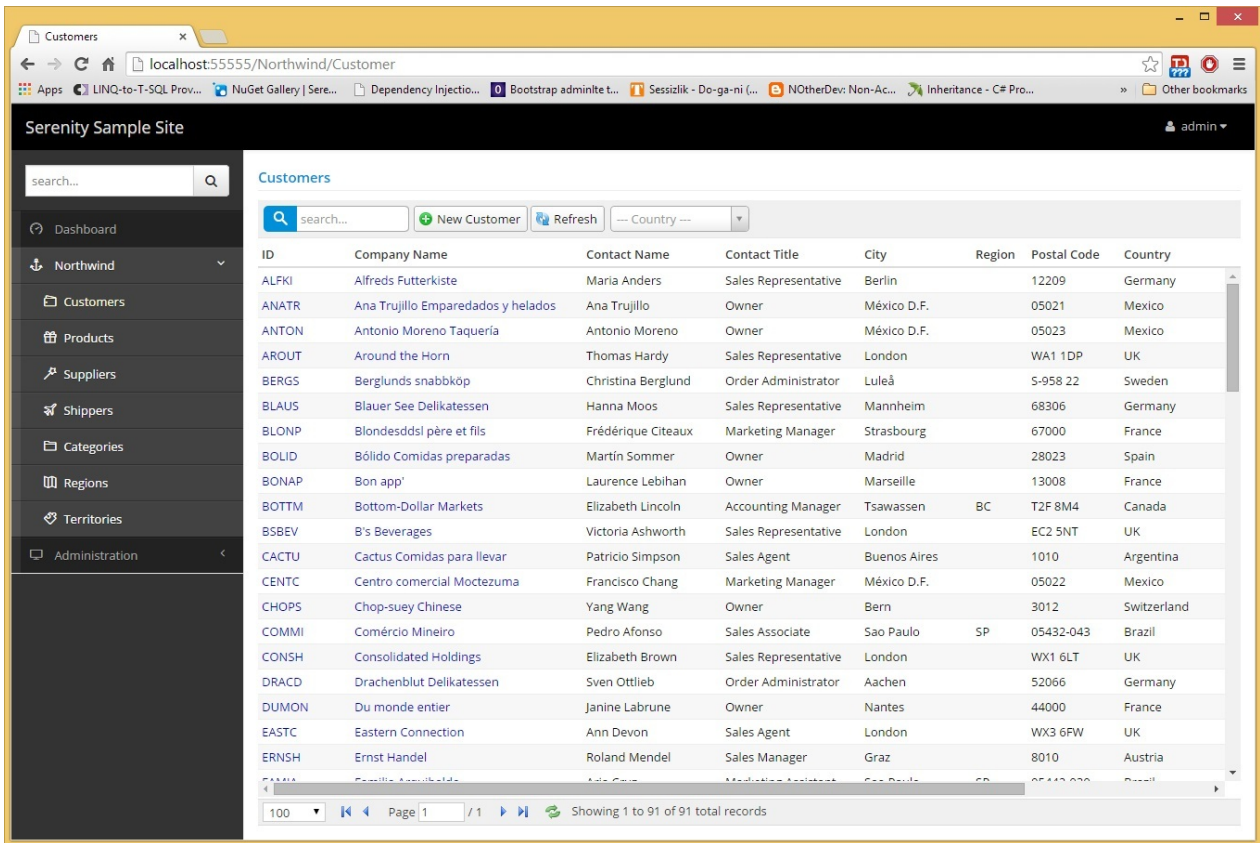
Once a language is chosen, all texts are displayed in the selected language.

Serenity platform is designed with localization in mind from the start.

If you are using *Serenity Basic Application Sample* you can see this yourself by setting your browser language or changing a web.config setting:

```
<system.web>
  <globalization culture="en-US" uiCulture="auto:en-US" />
</system.web>
```

Here, UI culture is set to automatic, and if automatic detection fails, en-US is used as a fallback.



Change this configuration as below, refresh your browser and you will the site in Turkish:

```
<system.web>
  <globalization culture="en-US" uiCulture="tr" />
</system.web>
```

The screenshot shows a web application interface for a customer list. The browser address bar indicates the URL is localhost:5555/Northwind/Customer. The page title is "Serenity Örnek Sitesi". The main content area is titled "Müşteriler" and contains a table with the following data:

Kayıt No	Firma Adı	İlgili Kişi Adı	İlgili Kişi Unvanı	Şehir	Bölge	Posta Kodu	Ülke
ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Berlin		12209	Germany
ANATR	Ana Trujillo Emparedados y helados	Ana Trujillo	Owner	México D.F.		05021	Mexico
ANTON	Antonio Moreno Taqueria	Antonio Moreno	Owner	México D.F.		05023	Mexico
AROUT	Around the Horn	Thomas Hardy	Sales Representative	London		WA1 1DP	UK
BERGS	Berglunds snabbköp	Christina Berglund	Order Administrator	Luleå		S-958 22	Sweden
BLAUS	Blauer See Delikatessen	Hanna Moos	Sales Representative	Mannheim		68306	Germany
BLONP	Blondesddsl père et fils	Frédérique Citeaux	Marketing Manager	Strasbourg		67000	France
BOLID	Bólido Comidas preparadas	Martin Sommer	Owner	Madrid		28023	Spain
BONAP	Bon app'	Laurence Labihan	Owner	Marseille		13008	France
BOTTM	Bottom-Dollar Markets	Elizabeth Lincoln	Accounting Manager	Tsawassen	BC	T2F 8M4	Canada
BSBEV	B's Beverages	Victoria Ashworth	Sales Representative	London		EC2 5NT	UK
CACTU	Cactus Comidas para llevar	Patricio Simpson	Sales Agent	Buenos Aires		1010	Argentina
CENTC	Centro comercial Moctezuma	Francisco Chang	Marketing Manager	México D.F.		05022	Mexico
CHOPS	Chop-suey Chinese	Yang Wang	Owner	Bern		3012	Switzerland
COMMI	Comércio Mineiro	Pedro Afonso	Sales Associate	Sao Paulo	SP	05432-043	Brazil
CONSH	Consolidated Holdings	Elizabeth Brown	Sales Representative	London		WX1 6LT	UK
DRACD	Drachenblut Delikatessen	Sven Ottlieb	Order Administrator	Aachen		52066	Germany
DUMON	Du monde entier	Janine Labrune	Owner	Nantes		44000	France
EASTC	Eastern Connection	Ann Devon	Sales Agent	London		WX3 6FW	UK
ERNSH	Ernst Handel	Roland Mendel	Sales Manager	Graz		8010	Austria
FRANR	Francois Aubert	Mica Garcia	Marketing Assistant	Sao Paulo	SP	05443-030	Brazil

The table is displayed in a web browser window. The page includes a search bar, a "Yeni Müşteri" button, and a "Yenile" button. The footer of the table indicates "Toplam 91 kayıttan 1 ile 91 arasındakiler gösteriliyor".

Here, data is not translated but it is also possible to translate user entered data by some methods like culture extension tables.

# LocalText Class

[namespace: *Serenity*, assembly: *Serenity.Core*]

At the core of string localization is LocalText class.

```
public class LocalText
{
    public LocalText(string key);
    public string Key { get; }
    public override string ToString();
    public static implicit operator string(LocalText localText);
    public static implicit operator LocalText(string key);
    public static string Get(string key);
    public static string TryGet(string key);

    public const string InvariantLanguageID = "";
    public static readonly LocalText Empty;
}
```

Its constructor takes a key parameter, which defines the local text key that it will contain. Some of sample keys are:

- Enums.Month.January
- Enums.Month.December
- Db.Northwind.Customer.CustomerName
- Dialogs.YesButton

Though it is not a rule, it is a good idea to follow this namespace like *dot* convention for local text keys.

At runtime, through ToString() function, the local text key is translated to its representation in the active language (which is *CultureInfo.CurrentCulture*).

```
var text = new LocalText("Dialogs.YesButton");
Console.WriteLine(text.ToString());
```

```
> Yes
```

If a translation is not found in local text table (we will talk about this later), the key itself is returned.



```
var text = new LocalText("Unknown.Local.Text.Key");  
Console.WriteLine(text.ToString());
```

```
> Unknown.Local.Text.Key
```

This is by design, so that developer can determine which translations are missing.

## LocalText.Key Property

Gets the local text key that LocalText instance contains.

## Implicit Conversions From String

LocalText has implicit conversion from String type.

```
LocalText someText = "Dialogs.YesButton";
```

Here *someText* variable references a new LocalText instance with the key *Dialogs.YesButton*. So it is just a shortcut to LocalText constructor.

## Implicit Conversions To String

LocalText has implicit conversion to String type too, but it returns translation instead of the key (just like calling *ToString()* method):

```
var lt = new LocalText("Dialogs.NoButton");  
string text = lt;  
Console.WriteLine(text);
```

```
> No
```

## LocalText.Get Static Method

To access the translation for a local text key without creating a LocalText instance, use Get method:

```
Console.WriteLine(LocalText.Get("Dialogs.YesButton"));
```

```
> Yes
```

*ToString()* method internally calls *Get*

## LocalText.TryGet Static Method

Unlike *Get* method which returns the local text key if no translation is found, *TryGet* returns null. Thus, coalesce operator can be used along with *TryGet* where required:

```
var translation = LocalText.TryGet("Looking.For.This.Key") ?? "Default Text";  
Console.WriteLine(translation);
```

```
> Default Text
```

## LocalText.Empty Field

Similar to *String.Empty*, *LocalText* contains an empty local text object with empty key.

## LocalText.InvariantLanguageID Constant

This is just an empty string for invariant language ID which is the invariant culture language identifier (default language, usually English).

We will talk about language identifiers in the following section.

# Language Identifiers

A language ID is a code that assigns letters and/or numbers as identifiers or classifiers for languages.

Language IDs follow the RFC 1766 standard in the format `<languagecode2>-<country/regioncode2>`, where *languagecode2* is a lowercase two-letter code derived from ISO 639-1 and *country/regioncode2* is an uppercase two-letter code derived from ISO 3166.

Some sample language IDs:

- `en` : English
- `en-US` : English as used in the United States (US is the ISO 3166-1 country code)
- `en-GB` : English as used in the United Kingdom (GB is the ISO 3166-1 country code)
- `es` : Spanish
- `es-AR` : Spanish as used in Argentina

## Invariant Language

Similar to *CultureInfo.InvariantCulture*, invariant language is the default language with empty identifier.

Unless specified otherwise, embedded texts in your assemblies are considered to be written in invariant language.

Though it is usually considered to be English, you may assume your natural language as the invariant language.

# Language Fallbacks

## Neutral Language Fallback

When a translation is not found in `en-US`, it is acceptable to look for a translation in `en` language, as they are closely related.

Two letter language IDs (neutral languages) are implicitly language fallbacks of 4 letter country specific codes.

So `es` is language fallback of `es-AR` and `en` is language fallback of `en-US` and `en-GB`.

## Invariant Language Fallback

Invariant language with empty code is the final fallback of all languages implicitly.

## Implementation

Language fallback functionality should be implemented by the `ILocalTextRegistry` provider (e.g. `LocalTextRegistry` class).

Providers may also support setting language fallbacks explicitly, so you can set `en-US` as language fallback of `en-UK` if needed.

This is how looking up a translation for a local text key works:

- If current language has a translation for the key, return it.
- Check every explicitly defined language fallback for a translation.
- If language ID is a 4 letter country specific code, check neutral language for a translation.
- Check invariant language for a translation.
- Return the key itself or null for `TryGet`.

Let's say we set `en-US` as language fallback of `en-UK`.

If we search for a translation in `en-UK`, it is looked up in this order:

1. `en-UK`
2. `en-US`
3. `en`

#### 4. invariant

# ILocalTextRegistry Interface

[namespace: *Serenity.Abstractions*, assembly: *Serenity.Core*]

LocalText class accesses translations for local text keys through the provider for this interface.

```
public interface ILocalTextRegistry
{
    string TryGet(string languageID, string key);
    void Add(string languageID, string key, string text);
}
```

## ILocalTextRegistry.TryGet Method

Gets translation for the specified key in requested language.

Current language is determined by *CultureInfo.CurrentCulture*.

It is *providers responsibility* to check language fallbacks for the key, if a translation is not found in requested language.

This method returns null if no translation is found in the language hierarchy (from requested language down to invariant language).

## ILocalTextRegistry.Add Method

Adds a translation to the local text table which is internally hold by the local text registry.

The local text table is an in-memory table (dictionary) like:

Key	LanguageID	Text (Translation)
Dialogs.YesButton	en	Yes
Dialogs.YesButton	tr	Evet
Dialogs.NoButton	en	No
Dialogs.NoButton	tr	Hayır

This method doesn't throw an exception if same key/language ID pair is added twice. It simply overrides existing translation.



# LocalTextRegistry Class

[namespace: *Serenity.Localization*, assembly: *Serenity.Core*]

This class is the embedded, default implementation of *ILocalTextRegistry* interface.

```
public class LocalTextRegistry : ILocalTextRegistry
{
    public void Add(string languageID, string key, string text);
    public string TryGet(string languageID, string key);
    public void SetLanguageFallback(string languageID, string languageFallbackID);

    public void AddPending(string languageID, string key, string text);
    public string TryGet(string languageID, string textKey, bool isApprovalMode);

    public Dictionary<string, string> GetAllAvailableTextsInLanguage(
        string languageID, bool pending);
}
```

Add and TryGet implements corresponding methods in *ILocalTextRegistry* interface.

## LocalTextRegistry.SetLanguageFallback Method

Sets language fallback for specified language.

```
var registry = (LocalTextRegistry)(Dependency.Resolve<ILocalTextRegistry>());
registry.SetLanguageFallback('en-UK', 'en-US');
// from now on if a translation is not found in "en-UK" language,
// it will be looked up in "en-US" language first, followed by "en".
```

More information about language fallbacks can be found in relevant section.

## Registering LocalTextRegistry as Provider

This is usually done in your application start method:

```
var registrar = Dependency.Resolve<IDependencyRegistrar>();
registrar.RegisterInstance<ILocalTextRegistry>(new LocalTextRegistry());
```



*CommonInitialization.Run* or *CommonInitialization.InitializeLocalTexts* methods also register a `LocalTextRegistry` instance as the `ILocalTextRegistry` provider, if none is already registered.

# Pending Approval Mode

LocalTextRegistry also supports an optional pending approval mode.

In some sites, translations might be needed to be approved by some moderators before they are published.

So you may add these unapproved texts to your local text registry but want them to be shown only to moderators for them to check how they will look in live site when approved.

LocalTextRegistry allows you to mark some texts as pending approval, and use these translations for only approval contexts (e.g. when a moderator is logged in).

## ILocalTextContext Interface

[namespace: *Serenity.Localization*, assembly: *Serenity.Core*]

```
public interface ILocalTextContext
{
    bool IsApprovalMode { get; }
}
```

Implement this interface and register it through the service locator (Dependency class).

IsApprovalMode property is used to determine if current context is in approval mode (e.g. used by a moderator).

```
public class MyLocalTextContext : ILocalTextContext
{
    public bool IsApprovalMode
    {
        get
        {
            // use some method to determine if current user is a moderator
            return Authorization.HasPermission("Moderation");
        }
    }
}

void ApplicationStart()
{
    Dependency.Resolve<IDependencyRegistrar>()
        .RegisterInstance<ILocalTextContext>(new MyLocalTextContext());
}
```

## LocalTextRegistry.AddPending Method

Adds a translation to local text table for pending approval texts. These texts are only used when current context is in pending approval mode.

## LocalTextRegistry.TryGet Overload With Language and Pending Arguments

```
public string TryGet(string languageID, string textKey, bool isApprovalMode);
```

This overload lets you to get a translation in specified language and optionally using unapproved texts (`isApprovalMode = true`).

Other `TryGet` overload returns unapproved texts only when `ILocalTextContext` provider returns true for `IsApprovalMode` property.

## LocalTextRegistry.GetAllAvailableTextsInLanguage Method

Returns a dictionary of all currently registered translations for all text keys in a language.

Dictionary keys are local text keys, while values are translations.

It also contains texts found from language fallbacks for if a translation is not available in requested language.

# Registering Translations

There are several ways to define local text keys and translations, including:

- Manually through `ILocalTextRegistry.Add` Method
- Declaring nested static classes containing local text objects
- Adding `Description` attribute to enumeration classes
- JSON files in predetermined locations (`~/scripts/serenity/texts`, `~/scripts/site/texts` and `~/App_Data/texts`)

We'll talk about all these methods.

# Manually Registering Translations

You can add translations to local text registry from your application start method.

Sources for these translations might be a database table, xml file, embedded resources etc.

```
void Application_Start()
{
    // ...
    var registry = Dependency.Resolve<ILocalTextRegistry>();
    registry.Add("es", "Dialogs.YesButton", "Sí");
    registry.Add("fr", "Dialogs.YesButton", "Oui");
    // ..
}
```

# Nested Local Texts

Serenity allows you to define nested static classes containing `LocalText` objects to define translations like below:

```
[NestedLocalTexts]
public static partial class Texts
{
    public static class Site
    {
        public static class Dashboard
        {
            public static LocalText WelcomeMessage =
                "Welcome to Serenity BasicApplication home page. " +
                "Use the navigation on left to browse other pages...";
        }
    }

    public static class Validation
    {
        public static LocalText DeleteForeignKeyError =
            "Can't delete record. '{0}' table has records that depends on this one!";

        public static LocalText SavePrimaryKeyError =
            "Can't save record. There is another record with the same {1} value!";
    }
}
```

This definitions allow you to reference localized texts with intellisense support, without having to memorize string keys.

These embedded translation definitions are commonly used to define default translations in invariant language (ultimate fallbacks).

Here is a table of translations that are defined with this `Texts` class:

Key	LanguageID	Text (Translation)
Site.Dashboard.WelcomeMessage		Welcome to Serenity BasicApp...
Validation.DeleteForeignKeyError		Can't delete record...
Validation.SavePrimaryKeyError		Can't save record...

Local text keys are generated from nested static class names with a dot inserted between. Topmost static class (`Texts`) name is ignored though it is a good idea to name it something like `Texts` for consistency.

Unless otherwise stated, language ID for these texts are considered to be the invariant language (empty string).

## NestedLocalTexts Attribute

Topmost class (e.g. *Texts*) for nested local text registration classes must have this attribute.

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple=false)]
public sealed class NestedLocalTextsAttribute : Attribute
{
    public NestedLocalTextsAttribute()
    {
    }

    public string LanguageID { get; set; }
    public string Prefix { get; set; }
}
```

It has two optional attributes, *LanguageID* and *Prefix*.

*LanguageID* allows you to define what language translations are in.

If not specified, translations are considered to be in the invariant language.

It is a good idea to register default texts in invariant language, even if texts are not in English, as invariant language is the eventual language fallback for all languages.

If we used it like:

```
[NestedLocalTexts(LanguageID = "en-US")]
public static partial class Texts
{
    // ..
}
```

LanguageID column in translations table would be "en-US":

Key	LanguageID	Text (Translation)
Site.Dashboard.WelcomeMessage	en-US	Welcome to Serenity BasicApp...
Validation.DeleteForeignKeyError	en-US	Can't delete record...

Prefix attribute value is used as a prefix for local text keys:



```
[NestedLocalTexts(LanguageID = "en-US", Prefix = "APrefix.")]  
public static partial class Texts  
{  
    // ..  
}
```

Key	LanguageID	Text (Translation)
APrefix.Site.Dashboard.WelcomeMessage	en-US	Welcome to Serenity BasicApp...
APrefix.Validation.DeleteForeignKeyError	en-US	Can't delete record...

## NestedLocalTextRegistration Class

[namespace: *Serenity.Localization*, assembly: *Serenity.Core*]

For nested local text definitions to be registered, you need to call *NestedLocalTextRegistration.Initialize()* method in your application start:

```
void Application_Start()  
{  
    NestedLocalTextRegistration.Initialize();  
}
```

CommonInitialization.Run and CommonInitialization.InitializeLocalTexts methods call it by default.

Once it is run, all translations with auto generated keys are added to current *ILocalTextRegistry* provider and *LocalText* instances in nested static classes are replaced with actual *LocalText* instances containing generated keys (they are set through reflection).

# Enumeration Texts

Display text for enumeration values can be specified with Description attribute.

```
namespace MyApplication
{
    public enum Sample
    {
        [Description("First Value")]
        Value1 = 1,
        [Description("Second Value")]
        Value2 = 2
    }
}
```

This enumeration and its Description attributes defines following local text keys and translations:

Key	LanguageID	Text (Translation)
Enums.MyApplication.Sample.Value1		First Value
Enums.MyApplication.Sample.Value2		Second Value

All texts are defined for invariant language ID by default.

You can use these keys to access translated descriptions for enumeration values, or use extension method `GetText()` defined for enumeration types (import namespace `Serenity` to make this extension method available).

```
using Serenity;
//...
Console.WriteLine(MyApplication.Sample.Value1.GetText());
```

```
> First Value
```

## EnumKey Attribute

Enumeration translations use full name of enumeration type as prefix to generate local text keys. This prefix can be overridden with `EnumKeyAttribute`:

```
namespace MyApplication
{
    [EnumKey("Something")]
    public enum Sample
    {
        [Description("First Value")]
        Value1 = 1,
        [Description("Second Value")]
        Value2 = 2
    }
}
```

Now defined keys and translations are:

Key	LanguageID	Text (Translation)
Enums.Something.Value1		First Value
Enums.Something.Value2		Second Value

## EnumLocalTextRegistration Class

[namespace: *Serenity.Localization*, assembly: *Serenity.Core*]

For enumeration local text definitions to be registered, you need to call *EnumLocalTextRegistration.Initialize()* method in your application start:

```
void Application_Start()
{
    EnumLocalTextRegistration.Initialize(ExtensibilityHelper.SelfAssemblies);
}
```

It gets list of assemblies to search for enumeration types. You can pass list of assemblies manually or use *ExtensibilityHelper.SelfAssemblies* which contains all assemblies that reference a Serenity assembly.

CommonInitialization.Run and CommonInitialization.InitializeLocalTexts methods call it by default.

# JSON Local Texts

Serenity supports local text registration through JSON files containing a simple key/value dictionary:

```
{
  "Forms.Administration.User.DisplayName": "Display Name",
  "Forms.Administration.User.Email": "E-mail",
  "Forms.Administration.User.EntitySingular": "User",
  "Forms.Administration.User.EntityPlural": "Users"
}
```

To register all local text keys and translations from JSON files in a folder , call `JsonLocalTextRegistration.AddFromFilesInFolder` with the path:

```
JsonLocalTextRegistration.AddFromFilesInFolder(@"C:\SomeFolder");
```

File names in the folder must follow a convention:

```
{Some Prefix You Choose}.{LanguageID}.json
```

where `{LanguageID}` is two or four letter language code. Use *invariant* as language code for invariant language.

Some sample file names are:

- `site.texts.en-US.json`
- `MyCoolTexts.es.json`
- `user.texts.invariant.json`

Files in a folder are parsed and added to registry in their file name order. Thus for sample file names above, order would be:

1. `MyCoolTexts.es.json`
2. `site.texts.en-US.json`
3. `user.texts.invariant.json`

This order is important as adding a translation in some language with same key overrides prior translation.

# CommonInitialization and Predetermined Folders

*CommonInitialization.Run* and *CommonInitialization.InitializeLocalTexts* calls this method for three predetermined locations under your web site:

1. `~/Scripts/serenity/texts` (serenity translations)
2. `~/Scripts/site/texts` (your application specific translations)
3. `~/App_Data/texts` (user translations made through translation interface)

Prefer using second one for your own files as first one is for Serenity resources.

Third one contains user translated texts. It is recommended to transfer texts from these files to application translation files under `~/Scripts/site/texts` before publishing.

# Caching

Caching is an important part of modern, heavy-traffic applications. Even if your web application isn't getting so much traffic now, it might later and it is a good idea to design it with caching in mind from the start.

- [Local Caching](#)
- [Distributed Caching](#)
- [Two Level Cache](#)

# Local Caching

Serenity provides some caching abstractions and utility functions to make it easier to work with local cache.

The term *local* means that cached items are hold in local memory (thus there is no serialization involved).

When your application is deployed on a web farm, local caching might not be enough or sometimes suitable. We will talk about this scenario in [Distributed Caching](#) section.

# ILocalCache Interface

[namespace: *Serenity.Abstractions*] - [assembly: *Serenity.Core*]

Defines a basic interface to work with the local cache.

```
public interface ILocalCache
{
    void Add(string key, object value, TimeSpan expiration);
    TItem Get<TItem>(string key);
    object Remove(string key);
    void RemoveAll();
}
```

A default implementation of `ILocalCache` ( `Serenity.Caching.HttpRuntimeCache` ) that uses `System.Web.Cache` exists in `Serenity.Web` assembly.

## ILocalCache.Add Method

Adds a value to cache with the specified key. If the key already exists in cache, its value is updated.

Items are hold in cache for `expiration` duration. You can specify `TimeSpan.Zero` for items that shouldn't expire automatically.

Values are added to cache with absolute expiration (thus they expire at a certain time, not sliding expiration).

```
Dependency.Resolve<ILocalCache>.Add("someKey", "someValue", TimeSpan.FromMinutes(5));
```

This method, in its default implementation, uses `HttpRuntime.Cache.Insert` method.

Avoid `HttpRuntime.Cache.Add` method, as it doesn't update value if there is already a key with same key in the cache, and it doesn't even raise an error so you won't notice anything. A mere engineering gem from ASP.NET)

## ILocalCache.Get <TItem> Method

Gets the value corresponding to the specified key in local cache.

If there is no such key in cache, an error may be raised only if `TItem` is of value type. For reference types returned value is `null` .



If value is not of type `TItem`, an exception is thrown.

You may use `object` as `TItem` parameter to prevent errors in case a value doesn't exist, or not of requested type.

### **ILocalCache.Remove Method**

Removes the item with specified key from local cache and returns its value.

No errors thrown if there is no value in cache with the specified key, simply `null` is returned.

```
Dependency.Resolve<ILocalCache>.Remove("someKey");
```

### **ILocalCache.RemoveAll Method**

Removes all items from local cache. Avoid using this except for special situations like unit tests, otherwise performance might suffer.

# LocalCache Static Class

[namespace: Serenity] - [assembly: Serenity.Core]

A static class that contains shortcuts to work easier with the registered ILocalCache provider.

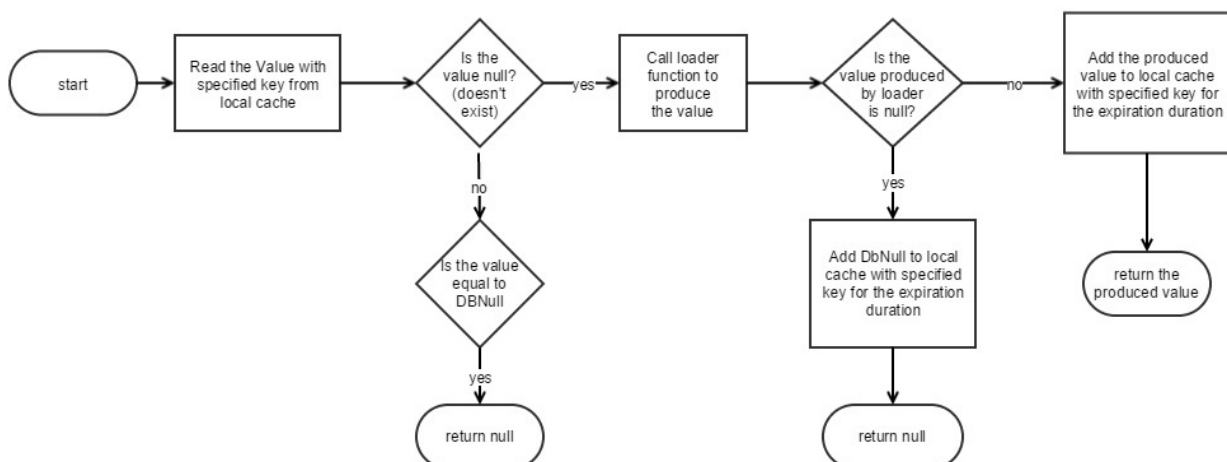
```
public static class LocalCache
{
    public static void Add(string key, object value, TimeSpan expiration);
    public static TItem Get<TItem>(string key, TimeSpan expiration,
        Func<TItem> loader) where TItem : class;
    public static void Remove(string key);
    public static void RemoveAll();
}
```

Add, Remove, and RemoveAll methods are simply shortcuts to corresponding methods in ILocalCache interface, but Get method is a bit different than ILocalCache.Get.

## LocalCache.Get <TItem> Method

Gets the value corresponding to the specified key in local cache.

If there is no such key in cache, uses the loader function to produce value, and adds it to cache with the specified key.



- If the value that exists in cache is DBNull.Value, than null is returned. (This way, if for example a user with an ID doesn't exist in database, repeated querying of database for that ID is prevented)

- If the value exists, but of not type TItem an exception is thrown, otherwise value is returned.
- If the value didn't exist in cache, loader function is called to produce the value (e.g. from database) and...
  - If the value produced by loader function is null, it is stored as DBNull.Value in cache.
  - Otherwise the produced value is added to cache with the specified expiration duration.

# User Profile Caching Sample

Lets assume we have a profile page in our site that is generated using several queries. We might have a model for this page e.g. UserProfile class that contains all profile data for a user, and a GetProfile method that produces this for a particular user id.

```
public class UserProfile
{
    public string Name { get; set; }
    public List<CachedFriend> Friends { get; set; }
    public List<CachedAlbum> Albums { get; set; }
    ...
}
```

```
public UserProfile GetProfile(int userID)
{
    using (var connection = new SqlConnection("..."))
    {
        // load profile by userID from DB
    }
}
```

By making use of LocalCache.Get method, we could cache this information for one hour easily and avoid DB calls every time this information is needed.

```
public UserProfile GetProfile(int userID)
{
    return LocalCache.Get<UserProfile>(
        cacheKey: "UserProfile:" + userID,
        expiration: TimeSpan.FromHours(1),
        loader: delegate {
            using (var connection = new SqlConnection("..."))
            {
                // load profile by userID from DB
            }
        }
    );
}
```

# Distributed Caching

Web applications might require to serve hundreds, thousands or even more users simultaneously. If you didn't take required measures, under such a load, your site might crash or become unresponsive.

Let's say you are showing the last 10 news in your home page and in a minute, in average of a thousand users are visiting this page. For every page view you might be querying your database to display this information:

```
SELECT TOP 10 Title, NewsDate, Subject, Body FROM News ORDER BY NewsDate DESC
```

Even if we think that our home page contains only this information, a site, that gets 10000 visits a minute would run 150 SQL queries per second.

These queries, as their result doesn't differ from user to user (always the last 10 news), might be cached in SQL server side automatically.

But query results consumes some valuable network bandwidth while being transferred from SQL server to your WEB server. As this transfer takes some time (data size / bandwidth) and your connection is kept open during this time, even if your SQL server responded instantly, getting the results wouldn't be so fast. The time to transfer might vary with the size of the news content.

Also as SQL connections which can be kept open simultaneously has a upper limit (connection pool limit) and when you reach that number, the connections start to wait in the queue and block each other.

By taking into account that news don't change every second, we could cache them in our WEB server memory for 5 minutes.

Thus as soon as we transfer news list from SQL database, store them in local cache. For the next 5 minutes, for every user that visits the home page, news list is read from local cache instantly, without even hitting SQL:

```
public List<News> GetNews()
{
    var news = HttpRuntime.Cache["News"] as List<News>;
    if (news == null)
    {
        using (var connection = new SqlConnection("....."))
        {
            news = connection.Query<News>("
                SELECT TOP 10 Title, NewsDate, Subject, Body
                FROM News
                ORDER BY NewsDate DESC")
                .ToList();

            HttpRuntime.Cache.Insert("News", ...,
                TimeSpan.FromMinutes(5), ....);
        }
    }

    return news;
}
```

This takes us from 150 queries per second down to 1/300 queries per second (a query per 300 sec).

Also these news items should be converted to HTML for every visitor. By moving one step further, we could also cache the HTML converted state of the news.

All these cached information is stored in WEB server memory which is the fastest location to access them.

Note that caching something doesn't always mean that your application will work faster. How effectively you use cache is more important than caching alone. It is even possible to slow down your application with caching, if not used properly.

# WEB Farms and Caching

Now let's consider we have a social networking site and have millions of user profiles. Profile pages of some famous users might be getting hundreds or thousands of visits per minute.

To generate a users profile, we would need more than one SQL query (friends, album names and picture counts, profile information, last status etc.).

As long as a user didn't update her profile, the information that is shown on her page would be almost stastic. Thus, a snapshot of profile pages could be cached for 5 minutes or 1 hour etc.

But this might not be enough. We are talking about hundres of millions of profiles and users. Users would be doing much more than just looking at some profile pages. We would need more than one server that are distributed in several geographical locations on earth (a WEB Farm).

At a certain time, all these servers might have cached a very important persons (VIP) profile in local cache. When the VIP makes a change in her profile, all these servers should renew their local cached profile, and this would happen in a few seconds. We now have a problem of load per server instead of load per user.

Actually, once one these of servers loaded the VIP profile from SQL database and cached it, other servers could make use of the same information without hitting database. But, as each server stores cached information in its own local memory, it is not trivial to access this information by other servers.

If we had a shared memory that all servers could access:

Information key	Value
Profile:VeryFamousOne	(Cached information for VeryFamousOne)
Profile:SomeAnother	...
...	...
...	...
Profile:JohnDoe	...

Let's call this memory the distributed cache. If all servers have a look at this common memory before trying DB we would avoid the load per server problem.

```
public CachedProfileInformation GetProfile(string profileID)
{
    var profile = HttpRuntime.Cache["Profil:" + profileID]
        as CachedProfileInformation;

    if (profile == null)
    {
        profile = DistributedCache.Get<CachedProfileInformation>(
            "Profil:" + profileID);

        if (profile == null)
        {
            using (var connection = new SqlConnection("....."))
            {
                profile = GetProfileFromDBWithSomeSQLQueries(profileID)
                    profile, TimeSpan.FromMinutes(5));

                DistributedCache.Set("Profil:" + profileID, profile,
                    TimeSpan.FromHours(1));
            }
        }
    }

    return news;
}
```

You can find many variations of distributed cache systems including Memcached, Couchbase and Redis. They are also called NoSQL database. You can think of them simply as a remote dictionary. They store key/value pairs in their memory and let you access them as fast as possible.

Warning! When it is used properly, distributed cache can improve performance of your application, just like local cache. Otherwise it can have a worse effect than local cache as there is a network transfer and serialization cost involved. Thus "if we keep things in distributed cache our site will run faster" is a myth.

When the cached data becomes too much, one computer memory might be not enough to store all key/value pairs. In this case servers like memcached distribute data by clustering. This could be done by the first letter of keys. One server could hold pairs starting with A, other with B etc. In fact, they use hash of keys for this purpose.



# IDistributedCache Interface

[namespace: *Serenity.Abstractions*, assembly: *Serenity.Core*]

All NoSQL server types provide a similar interface like "store this value for this key", "give me value corresponding to this key" etc.

Serenity provides its distributed cache support through a common interface to not depend on a specific kind of NoSQL database:

```
public interface IDistributedCache
{
    long Increment(string key, int amount = 1);
    TValue Get<TValue>(string key);
    void Set<TValue>(string key, TValue value);
    void Set<TValue>(string key, TValue value, TimeSpan expiration);
}
```

First overload of Set method that takes key and value arguments is used to store a key/value pair in distributed cache.

```
IoC.Resolve<IDistributedCache>().Set("someKey", "someValue");
```

Later we could read back this value using Get method:

```
var value = IoC.Resolve<IDistributedCache>().Get<string>("someKey") // someValue
```

If we wanted to keep some value for a predetermined duration, we could use the second overload of Get method:

```
IoC.Resolve<IDistributedCache>().Set("someKey", "someValue",
    TimeSpan.FromMinutes(10));
```

## IDistributedCache.Increment Method

Operation on distributed cache systems are usually not atomic and they provide no transactional systems at all.

Same key value can be changed by multiple servers at same time and override each others value in random order.

Let's say we needed a unique counter (to generate an ID for example) and synchronize it through distributed cache (to prevent using same ID twice):

```
int GetTheNextIDValue()
{
    var lastID = IoC.Resolve<IDistributedCache>().Get("LastID");
    IoC.Resolve<IDistributedCache>().Set("LastID", lastID + 1);
    return lastID;
}
```

Such a code block won't function as expected. Inside the duration between reading `LastID` value (get) and setting it to increment `LastID` value (set), another server might have read the same `LastID` value. Thus two servers could use same ID value.

For this purpose, you can use Increment method:

```
int GetTheNextIDValue()
{
    return IoC.Resolve<IDistributedCache>().Increment("LastID");
}
```

Increment function acts just like `Interlocked.Increment` method that is used in thread synchronization. It increases an identity value but blocks other requests while doing it, and returns the incremented value. So even if two WEB servers incremented same key in exact same moment, they end up with different ID values.

# Distributed Cache Static Class

[namespace: *Serenity*, assembly: *Serenity.Core*]

DistributedCache class provides shortcuts to methods for currently registered IDistributedCache implementation. So below two lines are functionally equal:

```
IoC.Resolve<IDistributedCache>().Increment("LastID");  
DistributedCache.Increment("LastID");
```

# DistributedCacheEmulator Class

[namespace: *Serenity.Abstractions*, assembly: *Serenity.Core*]

If you don't need a distributed cache now, but you wanted to write code that will work with a distributed cache in the future, you could use the DistributedCacheEmulator class.

DistributedCacheEmulator is also useful for unit tests and development environments (so that developers don't need to access a distributed cache system and work without affecting each other).

DistributedCacheEmulator emulates the IDistributedCache interface in a thread-safe manner by using a in-memory dictionary.

To use DistributedCacheEmulator, you need to register it with the Serenity Service Locator (IDependencyRegistrar). We do it from some method called on application start (global.asax.cs etc):

```
private static void InitializeDependencies()
{
    // ...
    var registrar = Dependency.Resolve<IDependencyRegistrar>();
    registrar.RegisterInstance<IDistributedCache>(new DistributedCacheEmulator());
    // ...
}
```

# CouchbaseDistributedCache Class

[namespace: *Serenity.Caching*, assembly: *Serenity.Caching.Couchbase*]

Couchbase is a distributed database that has Memcached like access interface.

You can get Serenity implementation for this server type in *Serenity.Caching.Couchbase* NuGet package.

Once you register it with the service locator:

```
Dependency.Resolve<IDependencyRegistrar>()  
    .RegisterInstance<IDistributedCache>(new CouchbaseDistributedCache())
```

You can configure *CouchbaseDistributedCache* in application configuration file (with JSON format):

```
<appSettings>  
  <add key="DistributedCache" value='{  
    ServerAddress: "http://111.22.111.97:8091/pools",  
    BucketName: "primary-bucket",  
    KeyPrefix: ""  
  }' />
```

Here *ServerAddress* is Couchbase server address and *BucketName* is the bucket name.

If you wanted to use same server / bucket for more than one application you can put something like `DEV:` , `TEST:`  into *KeyPrefix* setting.

# RedisDistributedCache Class

[namespace: *Serenity.Caching*, assembly: *Serenity.Caching.Couchbase*]

Redis is another in memory database that is also used by StackOverflow for its performance and reliability. They use just one Redis database for all their WEB servers.

You can get Serenity implementation for this server type in Serenity.Caching.Redis NuGet package.

It can be registered just like CouchbaseDistributedCache and configuration is similar (though there is no bucket setting):

```
<appSettings>
  <add key="DistributedCache" value="{
    ServerAddress: 'someredisserver:6379',
    KeyPrefix: ''
  }"/>
/>
```

# Two Level Caching

When you use local (in-memory) caching, one server can cache some information and retrieve it as fast as possible but as other servers can't access that cached data, they have to query for the same information from database.

If you prefer distributed caching to let other servers access cached data as it has some serialization / deserialization and network latency overhead, it may degrade performance in some cases.

There is also another problem with caching that needs to be handled: cache invalidation:

There are only two hard things in Computer Science: cache invalidation and naming things.

-- Phil Karlton

When you cache some information, you have to make sure that, when the source data changes, cached information is invalidated (regenerated or removed from cache).

# Using Local Cache and Distributed Cache in Sync

We might enjoy the best of both worlds by following a simple algorithm:

1. Check for key in local cache.
2. If key exists in local cache return its value.
3. If key doesn't exist in local cache, try distributed cache.
4. If key exists in distributed cache return its value and add it to local cache too.
5. If key doesn't exist in distributed cache, produce it from database, add it to both local cache and distributed cache. Return the produced value.

This way, when a server caches some information in local cache, it also caches it in distributed cache, but this time other servers can re-use information in distributed cache if they don't have a local copy in memory.

Once all servers have a local copy, none of them will need to access distributed cache again, thus, avoiding serialization and latency overhead.

## Validating Local Copies

All looks fine. But now we have a cache invalidation problem. What if in one of the servers cached data is changed. How do we notify them of this change, so that they can invalidate their *locally cached copy*?

We would change the value in distributed cache, but as they don't check distributed cache anymore (shortcut from step 2 in last algorithm), they wouldn't be noticed.

One solution to this problem would be to keep local copies for a certain time, e.g. 5 secs. Thus, when a server changes a cached data, other servers would use out-of-date information for 5 seconds mostly.

This method would help with batch operations that needs same cached information repeatedly. But even if nothing changed in distributed cache, we would have to get a copy from distributed cache to local cache every 5 seconds. If cached data is big, this would increase network bandwidth usage and deserialization cost.

We need a way to know if the data in distributed cache is different from the local copy. There are several ways of it that i can imagine:

- Store hash alongside data in local and distributed cache (slight hash calculation cost)



- Store an incrementing version number of data (how to make sure that two servers doesn't generate same version numbers?)
- Store last time data is set in distributed cache (time sync problems)
- Store a random number (generation) alongside data

Serenity uses generation numbers (random int) as version.

So when we store a value in distributed cache, let's say *SomeCachedKey*, we also store a random number with key *SomeCachedKey\$GENERATION\$*.

Now our prior algorithm becomes this:

1. Check for key in local cache.
2. If key exists in local cache
  - Compare its generation with one in distributed cache
  - If they are equal, return local cached value
  - If they don't match, continue to 4
3. If key doesn't exist in local cache, try distributed cache.
4. If key exists in distributed cache return its value and add it to local cache too, alongside its generation.
5. If key doesn't exist in distributed cache, generate it from database, add it to both local cache and distributed cache with some random generation. Return the produced value.

## Validating Multiple Cached Items In One Shot

You might have cached data produced from some table. There might be more than one key in distributed cache for this table.

Lets say you have a profile table and cached profile items by their User ID values.

When a user's profile information changes, you may try to remove its cached profile from cache. But what if another server or application you don't know about, cached some information that is generated from same user profile data? You may not know what cached information keys exist in distributed cache that depends on some user ID.

Most distributed cache implementations don't provide a way to find all keys that start with some string or it is computationally intensive (as they are dictionary based).

So when you want to expire all items depending on some set of data, it might not be feasible.

While caching items, Serenity allows you to specify a group key, which is used to expire them, when the data that the group depends on changes.

Let's say one application produced *CachedItem17* from a user with ID 17's profile data and we use this ID as a group key (Group17\_Generation):

Key	Value
CachedItem17	cxyzyxzcasd
CachedItem17_Generation	13579
Group17_Generation	13579

Here, random generation (version) for the group is 13579. Along with cached data (CachedItem17), we stored whatever was the group generation when we produced this data (CachedItem17\_Generation).

Suppose that another server, cached *AnotherItem17* from User 17's data:

Key	Value
CachedItem17	cxyzyxzcasd
CachedItem17_Generation	13579
AnotherItem17	uwdsasdas
AnotherItem17_Generation	13579
Group17_Generation	13579

Here, we reused Group17\_Generation, as there was already a group version number in distributed cache, otherwise we would have to generate a new one.

Now, both items in cache (CachedItem17 and AnotherItem17) are valid, because their version numbers matches the group version.

If somebody changed User 17's data and we wanted to expire all cached items related to her, we need to just change the group generation:

Key	Value
CachedItem17	cxyzyxzcasd
CachedItem17_Generation	13579
AnotherItem17	uwdsasdas
AnotherItem17_Generation	13579
Group17_Generation	54237

Now all cached items are expired. Even though they exist in cache, we see that their generations don't match the group generation, so they are not considered valid.

Group keys we use are usually name of the table that data is produced from.

# TwoLevelCache Class

[namespace: *Serenity*] - [assembly: *Serenity.Core*]

Out of the box, TwoLevelCache provides all functionality that we talked about so far and some more.

```
public static class TwoLevelCache
{
    public static TItem Get<TItem>(
        string cacheKey, TimeSpan expiration,
        string groupKey, Func<TItem> loader)
        where TItem : class;

    public static TItem Get<TItem>(
        string cacheKey, TimeSpan localExpiration, TimeSpan remoteExpiration,
        string groupKey, Func<TItem> loader)
        where TItem : class;

    public static TItem GetWithCustomSerializer<TItem, TSerialized>(
        string cacheKey, TimeSpan localExpiration, TimeSpan remoteExpiration,
        string groupKey, Func<TItem> loader,
        Func<TItem, TSerialized> serialize,
        Func<TSerialized, TItem> deserialize)
        where TItem : class
        where TSerialized : class;

    public static TItem GetLocalStoreOnly<TItem>(
        string cacheKey, TimeSpan localExpiration,
        string groupKey, Func<TItem> loader)
        where TItem : class;

    public static void ChangeGlobalGeneration(string globalGenerationKey);
    public static void Remove(string cacheKey);
}
```

## TwoLevelCache.Get Method

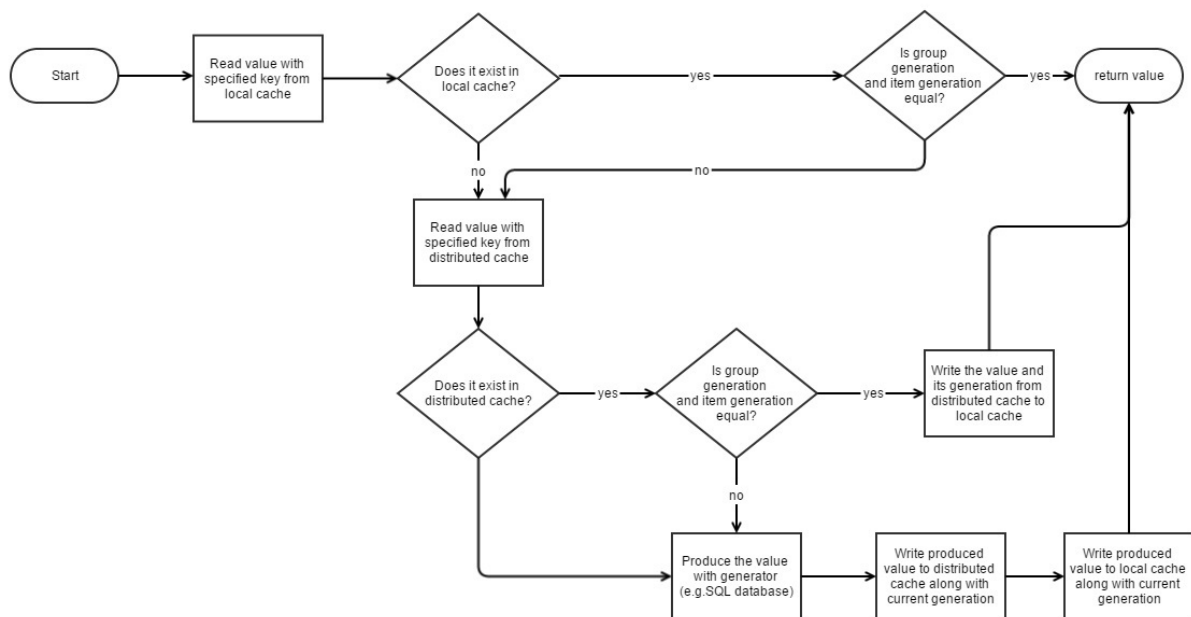
- Tries to read a value from local cache. If it is not found in there (or has an expired version), tries the distributed cache.
- If neither contains the specified key, produces value by calling a loader function and adds the value to local and distributed cache for a given expiration time.

- There are two overloads of the Get method. One that takes expiration time for local and distributed caches separately, and another that has only one expiration parameter for both.
- By using a group key, all items on both cache types that are members of this group can be expired at once (this is generation based expiration, not time).

To avoid checking group generation every time an item that belongs to group is requested, group generation itself is also cached in local cache. Thus, when a generation number changes, local cached items might expire after 5 seconds.

This means that, if you use this strategy in a web farm setup, when a change occurs in one server, other servers might continue to use old local cached data for 5 seconds more.

If this is a problem for your configuration, you should use DistributedCache methods directly instead of depending on TwoLevelCache.



```
CachedProfile GetCachedProfile(int userID)
{
    TwoLevelCache.Get("CachedProfile:" + userID, TimeSpan.FromDays(1), "SomeGroupKey",
        () =>
        {
            using (var connection = new SqlConnection("..."))
            {
                connection.Open();
                return LoadProfileFromDB(connection, userID);
            }
        });
}

CachedProfile LoadProfileFromDB(IDbConnection connection, int userID)
{
    // ...
}
```

## TwoLevelCache.GetWithCustomSerializer Method

TwoLevelCache.Get stores cached data in both local cache and distributed cache. While storing cached items in local cache, serialization is not required (in-memory). But before items are sent to distributed cache, some kind of serialization (binary, json etc.) must be performed (depends on provider and data type).

Sometimes this serialization / deserialization operation can be costly, so you might want to provide your own implementation of these functions for your data type.

GetWithCustomSerializer takes two extra delegate arguments to serialize and deserialize values. You might return a string or byte array from serialization function, and in deserialization take this string or byte array and turn it back into your original data type.

Most providers handle simple types like int, string or byte[] effectively, so for such data types you don't need custom serialization.

## TwoLevelCache.GetLocalStoreOnly Method

If you only want to store items in local cache and not distributed cache, GetLocalStoreOnly can be useful.

When cached data by one server is not helpful for others (changes from server to server), so big or slow to serialize / deserialize, storing such data in distributed cache is not meaningful.

So, why shouldn't you use LocalCache directly in this case?

You could but not if you want to specify a group key, and expire local cached items easily when source data of that group changes (as if they are stored in distributed cache).

## TwoLevelCache.ExpireGroupItems Method

This method allows you to expire all items that are members of one group key. It simply removes group key from local cache and distributed cache, so another version will be generated next time it is queried.

```
TwoLevelCache.ExpireGroupItems("SomeGroupKey");
```

You should call this from methods that change data.

If your entity class has *TwoLevelCached* attribute on it, *Create*, *Update*, *Delete* and *Undelete* handlers do this automatically with *ConnectionKey.TableName* as group key.

## TwoLevelCache.Remove Method

Removes an item and its version from local and distributed cache.

# Entities (Row)

Serenity entity system is a micro-orm that is in love with SQL just like Dapper.

Unlike full blown ORMs like NHibernate / Entity Framework, Serenity provides minimum features required to map and query databases with intellisense, compile time checking and easy refactoring.

Serenity entities are usually named like *XYZRow*. They are subclasses of *Serenity.Data.Row*.

Let's define a simple row class:

```
using Serenity;
using Serenity.ComponentModel;
using Serenity.Data;

public class SimpleRow : Row
{
    public string Name
    {
        get { return Fields.Name[this]; }
        set { Fields.Name[this] = value; }
    }

    public Int32? Age
    {
        get { return Fields.Age[this]; }
        set { Fields.Age[this] = value; }
    }

    public static RowFields Fields = new RowFields().Init();

    public SimpleRow()
        : base(Fields)
    {
    }

    public class RowFields : RowFieldsBase
    {
        public StringField Name;
        public Int32Field Age;
    }
}
```



Yes, it looks a bit more complicated than a simple POCO class. This is required to make some features work without using proxy classes like some ORMs use (Entity Framework, NHibernate etc).

This structure allows us to build queries with zero reflection, do assignment tracking, enable INotifyPropertyChanged when required. It makes it also possible to work with custom, user defined fields.

Rows are JSON serializable, so they can be returned from services without any problems. You don't need extra POCO/DTO classes unless you have a good reason to use them.

Let's study parts of a row declaration.

```
public class SimpleRow : Row
```

Here we define an entity named SimpleRow, which probably maps to a table named `Simple` in database.

Row suffix here is not required, but common practice, and it prevents clashes with other class names.

All entity classes derive from `Serenity.Data.Row` base class.

```
public string Name
{
    get { return Fields.Name[this]; }
    set { Fields.Name[this] = value; }
}
```

Now we declare our first property. This property maps to a database column named `Name` in the `Simple` table.

It is not possible to use an auto property here (like `get; set;` ). Field values must be read and set through a special object called *Field*.

Field objects are very similar to WPF dependency properties. Here is a dependency property declaration sample:

```

public static readonly DependencyProperty MyCustomProperty =
    DependencyProperty.Register("MyCustom", typeof(string), typeof(Window1));

public string MyCustom
{
    get { return this.GetValue(MyCustomProperty) as string; }
    set { this.SetValue(MyCustomProperty, value); }
}

```

Here we define a static dependency property object (`MyCustomProperty`), that contains property metadata and allows us to set and get property value through its *GetValue* and *SetValue* methods. Dependency properties allows WPF to offer features like validation, data binding, animation, and more.

Similar to dependency properties, Field objects contains column metadata and clears way for some features like assignment tracking, building queries without expression trees, change notification etc.

While dependency properties are declared as static members in class they are used, Field objects are declared in a nested class named `RowFields`. This allows to group and reference them easier, without having to add *Field* or *Property* suffix, and keeps our entity clear from field declarations.

```

public Int32? Age
{
    get { return Fields.Age[this]; }
    set { Fields.Age[this] = value; }
}

```

Here is our second property, named `Age` , with type `Int32?` .

Serenity entity properties are always nullable, even if database column is not nullable.

Serenity never use zero in place of null.

This might seem unlogical, if you have a background of other ORMs, but consider this:

Is it not possible for a not null field to have a null value, if you query it through a left/right join? How can you say, if its retrieved value is null or zero in that case?

Reference types are already nullable, so you can't write `String?` .

```

public static RowFields Fields = new RowFields().Init();

```

We noted that field objects are declared in a nested subclass named *RowFields* (usually). Here we are creating its sole static instance. Thus, there is only one *RowFields* instance per row type, and one field instance per row property.

`Init` is an extension method that initializes members of *RowFields*. It creates field objects that are not explicitly initialized.

```
public SimpleRow()
    : base(Fields)
{
}
```

Now we define *SimpleRow*'s parameterless constructor. Base *Row* class requires a *RowFields* instance to work, and we pass our static *Fields* object. So all instances of a row type (*SimpleRow*) share a single *RowFields* (*SimpleRow.RowFields*) instance. This means they share all the metadata.

```
public class RowFields : RowFieldsBase
{
    public StringField Name;
    public Int32Field Age;
}
```

Here we define our nested class that contains field objects. It should be derived from `Serenity.Data.RowFieldsBase`. *RowFieldsBase* is a special class closely related to *Row* that contains table metadata.

We declared a *StringField* and a *Int32Field*. Their type is based on their property types, and they must match exactly.

Their names must also match the property names, or you'll get an initialization error.

We didn't initialize these field objects, so their values are initially null.

Remember that we wrote `new RowFields().Init()` above. This is where field objects are automatically created.

It's also possible to initialize them in *RowFields* constructor manually, but not recommended, except for special customizations.

# Mapping Attributes

Serenity provides some mapping attributes, to match database table, column names with rows.

## Column and Table Mapping Conventions

By default, a row class is considered to match a table in database with the same name, but *Row* suffix removed.

A property is considered to match a column in database with the same name.

Let's say we have such a row definition:

```
public class CustomerRow : Row
{
    public string StreetAddress
    {
        get { return Fields.StreetAddress[this]; }
        set { Fields.StreetAddress[this] = value; }
    }
}
```

If we wrote a query, selecting *StreetAddress* field from *CustomerRow*, it would be generated like below:

```
SELECT
T0.StreetAddress AS [StreetAddress]
FROM Customer T0
```

*CustomerRow* matches table *Customer* by convention. Similarly, *StreetAddress* property matches a column named *StreetAddress*.

`T0` is a special alias assigned to main table by Serenity rows.

As, *StreetAddress* column belongs to main table (*Customer*), it is selected with an expression of `T0.StreetAddress` and with a column alias of `[StreetAddress]`.

Property name is used as a column alias by default

## SqlSettings.AutoQuotedIdentifiers Flag

In some database systems, identifiers are case sensitive.

For example, in Postgres, if you create a column with quoted identifier `"StreetAddress"`, you have to use quotes when selecting it, even if you write `SELECT StreetAddress ...` (same case) it won't work.

You have to use the form `SELECT "StreetAddress"`.

Thus, Postgres users usually prefer lowercase identifiers. But FluentMigrator always quotes identifiers, so we need a workaround to add brackets/quotes to identifiers.

Serenity doesn't quote/bracket column and table names by default, but it has a setting for compability.

If `SqlSettings.AutoQuotedIdentifiers` flag is set to true, previous query would look like this:

```
SELECT
T0.[StreetAddress] AS [StreetAddress]
FROM [Customer] T0
```

This setting defaults to *false* in Serenity for backwards compability, but Serene 1.8.6+ sets it to *true* on application startup.

And if we used Postgres dialect, output would be:

```
SELECT
T0."StreetAddress" AS "StreetAddress"
FROM "Customer" T0
```

## Column Attribute

[namespace: *Serenity.Data.Mapping*] - [assembly: *Serenity.Data*]

You can map a property to some other column name in database using *Column* attribute:

```
public class CustomerRow : Row
{
    [Column("street_address")]
    public string StreetAddress
    {
        get { return Fields.StreetAddress[this]; }
        set { Fields.StreetAddress[this] = value; }
    }
}
```

Now the query becomes:

```
SELECT
T0.street_address AS [StreetAddress]
FROM Customer T0
```

It is also possible to manually add brackets:

```
public class CustomerRow : Row
{
    [Column("[street_address]")]
    public string StreetAddress
    {
        get { return Fields.StreetAddress[this]; }
        set { Fields.StreetAddress[this] = value; }
    }
}
```

```
SELECT
T0.[street_address] AS [StreetAddress]
FROM Customer T0
```

If `SqlSettings.AutoQuotedIdentifiers` is true, brackets are automatically added.

Use `SqlServer` specific brackets ( `[ ]` ) if you need to work with multiple database types. These brackets are converted to dialect specific quotes (double quote, backtick etc.) before running queries.

But, if you only target one type of database, you may prefer using quotes specific to that database type.

## TableName Attribute

[namespace: *Serenity.Data.Mapping*] - [assembly: *Serenity.Data*]

If table name in database is different from row class name, use this attribute:

```
[TableName("TheCustomers")]
public class CustomerRow : Row
{
    public string StreetAddress
    {
        get { return Fields.StreetAddress[this]; }
        set { Fields.StreetAddress[this] = value; }
    }
}
```

```
SELECT
T0.StreetAddress AS [StreetAddress]
FROM TheCustomers T0
```

You may also use brackets or quotes:

```
[TableName("[My Customers]")]
public class CustomerRow : Row
{
    public string StreetAddress
    {
        get { return Fields.StreetAddress[this]; }
        set { Fields.StreetAddress[this] = value; }
    }
}
```

```
SELECT
T0.StreetAddress AS [StreetAddress]
FROM [My Customers] T0
```

Again, prefer brackets for database compability

## Expression Attribute

[namespace: Serenity.Data.Mapping] - [assembly: Serenity.Data]

This attribute is used to specify expression of a non-basic field, e.g. one that doesn't actually exist in database.

There can be several types of such fields.

One example is a Fullname field with a calculated expression like `(T0.[Firstname] + ' ' + T0.[Lastname])`.

```
public class CustomerRow : Row
{
    public string Firstname
    {
        get { return Fields.Firstname[this]; }
        set { Fields.Firstname[this] = value; }
    }

    public string Lastname
    {
        get { return Fields.Lastname[this]; }
        set { Fields.Lastname[this] = value; }
    }

    [Expression("(T0.[Firstname] + ' ' + T0.[Lastname])")]
    public string Fullname
    {
        get { return Fields.Fullname[this]; }
        set { Fields.Fullname[this] = value; }
    }
}
```

Be careful with "+" operator here as it is Sql Server specific. If you want to target multiple databases, you should write the expression as:

```
CONCAT(T0.[Firstname], CONCAT(' ', T0.[Lastname]))
```

Firstname and Lastname are table fields (actual fields in the table), but even if they don't have an expression attribute, they have basic, implicitly defined expressions, `T0.Firstname` and `T0.Lastname` (main table is assigned `T0` alias in Serenity queries).

In this document, when we talk about a *Table Field*, it means a field that actually corresponds to a column in database table.

*View Field* means a field with a calculated expression or a field that originates from another table, like fields that comes from joins in SQL views.

We wrote Fullname expression using `T0` alias before the fields that we reference.

It would probably work without that prefix too. But it is better to use it. When you start to add joins, it is possible to have more than one field with same name and experience ambiguous column errors.

## ForeignKey Attribute

[namespace: Serenity.Data.Mapping] - [assembly: Serenity.Data]



This attribute is used to specify foreign key columns, and add information about primary table and primary field that they are related to.

```
public class CustomerRow : Row
{
    [ForeignKey("Countries", "Id")]
    public string CountryId
    {
        get { return Fields.Firstname[this]; }
        set { Fields.Firstname[this] = value; }
    }
}
```

Here we specified that *CountryId* field in *Customer* table has a foreign key to *Id* field in *Countries* table.

The foreign key doesn't have to exist in database. Serenity doesn't check it.

Serenity can make use of such meta information, even though it doesn't affect generated queries alone.

ForeignKey is more meaningful when used along with the next attribute we'll see.

## LeftJoin Attribute

Where we are querying database, we tend to make many joins because of relations. Most of these joins are LEFT or INNER joins.

With Serenity entities, you'll usually be using LEFT JOINS.

Database admins prefers to define views to make it easier to query a combination of multiple tables, and to avoid writing these joins again and again.

Serenity entities can be used just like SQL views, so you can bring in columns from other tables to an entity, and query it as if they are one big combined table.

```
public class CustomerRow : Row
{
    [ForeignKey("Cities", "Id"), LeftJoin("c")]
    public Int32? CityId
    {
        get { return Fields.CityId[this]; }
        set { Fields.CityId[this] = value; }
    }

    [Expression("c.[Name]")]
    public string CityName
    {
        get { return Fields.CityName[this]; }
        set { Fields.CityName[this] = value; }
    }
}
```

Here we specified that *Cities* table should be assigned alias `c` when joined, and its join type should be `LEFT JOIN`. The join `ON` expression is determined as `c.[Id] == T0.[cityId]` with some help from *ForeignKey* attribute.

LEFT JOIN is preferred as it allows to retrieve all records from *left* table, *Customers*, even if they don't have a `CityId` set.

*CityName* is a view field (not actually a column of *Customer* table), which has an expression `c.Name`. It is clear that *CityName* originates from *Name* field in *Cities* table.

Now, if we wanted to select city names of all customers, our query text would be:

```
SELECT
c.Name AS [CityName]
FROM Customer T0
LEFT JOIN Cities c ON (c.[Id] = T0.CityId)
```

What if we don't have a `CountryId` field in *Customer* table, but we want to bring *Country* names of cities through `CountryId` field in *city* table?

```

public class CustomerRow : Row
{
    [ForeignKey("Cities", "Id"), LeftJoin("c")]
    public Int32? CityId
    {
        get { return Fields.CityId[this]; }
        set { Fields.CityId[this] = value; }
    }

    [Expression("c.[Name]")]
    public string CityName
    {
        get { return Fields.CityName[this]; }
        set { Fields.CityName[this] = value; }
    }

    [Expression("c.[CountryId]"), ForeignKey("Countries", "Id"), LeftJoin("o")]
    public Int32? CountryId
    {
        get { return Fields.CountryId[this]; }
        set { Fields.CountryId[this] = value; }
    }

    [Expression("o.[Name]")]
    public string CountryName
    {
        get { return Fields.CountryName[this]; }
        set { Fields.CountryName[this] = value; }
    }
}

```

This time we did a LEFT JOIN on CountryId field in Cities table. We assigned `o` alias to Countries table and bring in the name field from it.

You can assign any table alias to joins as long as they are not reserved words, and are unique between other joins in the entity. Sergen generates aliases like `jCountry`, but you may rename them to shorter and more natural ones.

Let's select CityName and CountryName fields of all Customers:

```

SELECT
c.[Name] AS [CityName],
o.[Name] AS [CountryName]
FROM Customer T0
LEFT JOIN Cities c ON (c.[Id] = T0.CityId)
LEFT JOIN Countries o ON (o.[Id] = c.[CountryId])

```

We'll see how to build such queries in FluentSQL chapter.

So far, we used `LeftJoin` attribute with properties that has a `ForeignKey` attribute with them.

It is also possible to attach `LeftJoin` attribute to entity classes. This is useful for joins without a corresponding field in main entity.

For example, let's say you have a `CustomerDetails` extension table that stores some extra details of customers (1 to 1 relation). `CustomerDetails` table has a primary key, `CustomerId`, which is actually a foreign key to `Id` field in `Customer` table.

```
[LeftJoin("cd", "CustomerDetails", "cd.[CustomerId] = T0.[Id]")]
public class CustomerRow : Row
{
    [Identity, PrimaryKey]
    public Int32? Id
    {
        get { return Fields.Id[this]; }
        set { Fields.Id[this] = value; }
    }

    [Expression("cd.[DeliveryAddress]")]
    public string DeliveryAddress
    {
        get { return Fields.DeliveryAddress[this]; }
        set { Fields.DeliveryAddress[this] = value; }
    }
}
```

And here what it looks like when you select `DeliveryAddress`:

```
SELECT
cd.[DeliveryAddress] AS [DeliveryAddress]
FROM Customer T0
LEFT JOIN CustomerDetails cd ON (cd.[CustomerId] = T0.[Id])
```

# FieldFlags Enumeration

[namespace: *Serenity.Data.Mapping*] - [assembly: *Serenity.Data*]

Serenity has a set of field flags that controls field behavior.

```
public enum FieldFlags
{
    None = 0,
    Insertable = 1,
    Updatable = 2,
    NotNull = 4,
    PrimaryKey = 8,
    AutoIncrement = 16,
    Foreign = 32,
    Calculated = 64,
    Reflective = 128,
    NotMapped = 256,
    Trim = 512,
    TrimToEmpty = 512 + 1024,
    DenyFiltering = 2048,
    Unique = 4096,
    Default = Insertable | Updatable | Trim,
    Required = Default | NotNull,
    Identity = PrimaryKey | AutoIncrement | NotNull
}
```

An ordinary table field has *Insertable*, *Updatable* and *Trim* flags set by default which corresponds to *Default* combination flag.

## Insertable Flag

*Insertable* flag controls if the field is editable in new record mode. By default, all ordinary fields are considered to be insertable.

Some fields might not be insertable in database table, e.g. identity columns shouldn't have this flags set.

When a field doesn't have this flag, it won't be editable in forms in new record mode. This is also validated in services at repository level.

Sometimes, there might be internal fields that are perfectly valid in SQL INSERT statements, but shouldn't be edited in forms. One example might be a *InsertedByUserId* which should be set on service level, and not by end user. If we would let end user to edit it in forms, this

would be a security hole. Such fields shouldn't have Insertable flag set too.

This means field flags don't have to match database table settings.

## Insertable Attribute

To turn off Insertable flag for a field, put a `[Insertable(false)]` attribute on it:

```
[Insertable(false)]
public string MyField
{
    get { return Fields.MyField[this];
        set { Fields.MyField[this] = value;
            }
}
```

Use *Insertable(true)* to turn it on.

Non insertable fields are not hidden. They are just readonly. If you want to hide them, use `[HideOnInsert]` attribute (Serenity 1.9.8+) or write something like *form.MyField.GetGridField().Toggle(IsNew)* by overriding *UpdateInterface* method of your dialog.

## Updatable Flag

This flag is just like *Insertable* flag, but controls edit record mode in forms and update operations in services. By default, all ordinary fields are considered to be updatable.

## Updatable Attribute

To turn off Updatable flag for a field, put a `[Updatable(false)]` attribute on it:

```
[Updatable(false)]
public string MyField
{
    get { return Fields.MyField[this];
        set { Fields.MyField[this] = value;
            }
}
```

Use *Updatable(true)* to turn it on.

Non updatable fields are not hidden in dialogs. They are just readonly. If you want to hide them, use [HideOnUpdate] attribute (Serenity 1.9.8+) or write something like *form.MyField.GetGridField().Toggle(!IsNew)* by overriding *UpdateInterface* method of your dialog.

## Trim Flag

This flag is only meaningful for string typed fields and controls whether their value should be trimmed before save. All string fields have this flag on by default.

When a field value is empty string or whitespace only, it is trimmed to null.

## TrimToEmpty Flag

Use this flag if you prefer to trim string fields to empty string instead of null.

When a field value is null or whitespace only, it is trimmed to empty string.

## SetFieldFlags Attribute

This attribute can be used on fields to include or exclude a set of flags. It takes a first required parameter to include flags, and a second optional parameter to exclude flags.

To turn on TrimToEmpty flag on a field, we use it like this:

```
[SetFieldFlags(FieldFlags.TrimToEmpty)]
public string MyField
{
    get { return Fields.MyField[this];
        set { Fields.MyField[this] = value;
    }
}
```

To turn off Trim flag:

```
[SetFieldFlags(FieldFlags.None, FieldFlags.TrimToEmpty)]
public string MyField
{
    get { return Fields.MyField[this];
        set { Fields.MyField[this] = value;
    }
}
```

To include TrimToEmpty and Updatable but remove Insertable:

```
[SetFieldFlags(
    FieldFlags.Updatable | FieldFlags.TrimToEmpty,
    FieldFlags.Insertable)]
public string MyField
{
    get { return Fields.MyField[this];
    set { Fields.MyField[this] = value;
    }
}
```

Insertable and Updatable attributes are subclasses of SetFieldFlags attribute.

## NotNull Flag

Use this flag to set fields as not nullable. By default, this flag is set for fields that are not nullable in database, using NotNull attribute.

When a field is not nullable, its corresponding label in forms has a red asterisk and they are required to be entered.

## NotNullable Attribute

This sets the NotNull attribute on a field to ON. Remove attribute to turn it off.

You may also use [Required(false)] to make field not required in forms, even if it is not nullable in database. This doesn't clear the NotNull flag.

## Required Flag

This is a combination of Default and NotNullable flags.

It has no relation to [Required] attribute which controls validation in forms.

## PrimaryKey Flag and PrimaryKey Attribute

Set this for primary key fields in table.

Primary key fields are selected on Key column selection mode in List and Retrieve request handlers.

[PrimaryKey] attribute sets this flag ON.



## AutoIncrement Flag and AutoIncrement Attribute

Set this for fields that are auto incremented on server side, e.g. identity columns, or columns using a generator.

## Identity Flag and Identity Attribute

This is a combination of PrimaryKey, AutoIncrement and NotNull flags, which is common for identity columns.

## Foreign Flag

This flag is set for foreign view fields, that are originating from other tables through a join.

It is automatically set for fields with expressions containing table aliases other than T0.

For example, if a field has an attribute like [Expression("jCountry.CountryName")] it will have this flag.

This has no relation to ForeignKey attribute

## Calculated Flag

If a field has an expression involving more than one field or some mathematical operations, it will have this flag.

This could also be set for fields that are calculated on SQL server side.

## NotMapped Flag and NotMapped Attribute

Corresponds to an unmapped field in Serenity entities. They don't have a corresponding field in database table.

These kinds of fields can be used for temporary calculation, storage and transfer on client and service layers.

## Reflective Flag

This is used for an advanced form of unmapped fields, where they don't have a storage of their own in row, but reflects value of another field in a different form. For example, a field that displays absolute value of a integer field that can be negative.

This should only be used in rare cases for such unmapped fields.

## DenyFiltering Flag

If set, denies filtering operations on a sensitive field. This can be useful for secret fields like PasswordHash, that shouldn't be allowed to be selected or filtered by client side.

## Unique Flag and Unique Attribute

When a field has this flag, its value is checked against existing values in database to be unique.

You can turn on this flag with Unique attribute and determine if this constraint should be checked on service level (before the check in database level to avoid cryptic constraint errors).

# Fluent SQL

Serenity contains a set of query builders for SELECT, INSERT, UPDATE and DELETE statements.

These builders can be used with simple strings or Serenity entity (row) system.

Their output can be executed directly, through a micro-orm like Dapper (which is integrated with Serenity), or Serenity extensions.

# SqlQuery Object

[namespace: *Serenity.Data*] - [assembly: *Serenity.Data*]

SqlQuery is an object to compose dynamic SQL SELECT queries through a fluent interface.

## Advantages

SqlQuery offers some advantages over hand crafted SQL:

- Using IntelliSense feature of Visual Studio while composing SQL
- Fluent interface with minimal overhead
- Reduced syntax errors as the query is checked compile time, not execution time.
- Clauses like Select, Where, Order By can be used in any order. They are placed at correct positions when converting the query to string. Similarly, such clauses can be used more than once and they are merged during string conversion. So you can conditionally build SQL depending on input parameters.
- No need to mess up with parameters and parameter names. All values used are converted to auto named parameters. You can also use manually named parameters if required.
- It can generate a special query to perform paging on server types that doesn't support it natively (e.g. SQL Server 2000)
- With the dialect system, query can be targeted at specific server type and version.
- If it is used along with Serenity entities (it can also be used with micro ORMs like Dapper), helps to load query results from a data reader with zero reflection. Also it does left/right joins automatically.

## How To Try Samples Here

I recommend using LinqPad to try samples given here.

You should add reference to *Serenity.Core*, *Serenity.Data* and *Serenity.Data.Entity* NuGet packages.

Another option is to locate and reference these DLLs directly from a Serene application's *bin* or *packages* directory.

Make sure you add *Serenity* and *Serenity.Data* to Additional Namespace Imports in Query Properties dialog.

## A Simple Select Query

```
void Main()
{
    var query = new SqlQuery();
    query.Select("Firstname");
    query.Select("Surname");
    query.From("People");
    query.OrderBy("Age");

    Console.WriteLine(query.ToString());
}
```

This will result in output:

```
SELECT
Firstname,
Surname
FROM People
ORDER BY Age
```

In the first line of our program, we called `SqlQuery` with its sole parameterless constructor. If `ToString()` was called at this point, the output would be:

```
SELECT FROM
```

`SqlQuery` doesn't perform any syntax validation. It just converts the query you build yourself, by calling its methods. Even if you don't select any fields or call from methods, it will generate this basic `SELECT FROM` statement.

`SqlQuery` can't generate empty queries.

Next, we called `select` method with string parameter `"FirstName"`. Our query is now like this:

```
SELECT Firstname FROM
```

When `Select("Surname")` statement is executed, `SqlQuery` put a comma between previously selected field (*Firstname*) and this one:

```
SELECT Firstname, Surname FROM
```

After executing *From* and *OrderBy* methods, our final output is:

```
SELECT Firstname, Surname FROM People ORDER BY Age
```

## Method Call Order and Its Effects

In previous sample, output wouldn't change even if we reordered *From*, *OrderBy* and *Select* lines. It would change only if we changed order of *Select* statements...

```
void Main()
{
    var query = new SqlQuery();
    query.From("People");
    query.OrderBy("Age");
    query.Select("Surname");
    query.Select("Firstname");

    Console.WriteLine(query.ToString());
}
```

...but, only the column ordering inside the `SELECT` statement would change:

```
SELECT
Surname,
Firstname
FROM People
ORDER BY Age
```

You might use methods like `Select`, `From`, `OrderBy`, `GroupBy` in any order, and can also mix them (e.g. call `Select`, then `OrderBy`, then `Select` again...)

Putting `FROM` at start is recommended, especially when used with Serenity entities, as it helps with auto joins and determining database dialect etc.

## Method Chaining

It is a bit verbose and not so readable to start every line `query.`. Almost all *SqlQuery* methods are chainable, and returns the query itself as result.

We may rewrite the query like this:

```
void Main()
{
    var query = new SqlQuery()
        .From("People")
        .Select("Firstname")
        .Select("Surname")
        .OrderBy("Age");

    Console.WriteLine(query.ToString());
}
```

This feature is similar to jQuery and LINQ enumerable method chaining.

We could even get rid of the query variable:

```
void Main()
{
    Console.WriteLine(
        new SqlQuery()
            .From("People")
            .Select("Firstname")
            .Select("Surname")
            .OrderBy("Age")
            .ToString());
}
```

It is strongly recommended to put every method on its own line, and indent properly for readability and consistency reasons.

## Select Method

```
public SqlQuery Select(string expression)
```

In the samples we had so far, we used the overload of the *Select* method shown above (it has about 11 overloads).

Expression parameter can be a simple field name or an expression like `"FirstName + ' ' + LastName"`

Whenever this method is called, the expression you set is added to the SELECT statement of resulting query with a comma between.

There is also a `SelectMany` method to select multiple fields in one call:

```
public SqlQuery SelectMany(params string[] expressions)
```

For example:

```
void Main()
{
    var query = new SqlQuery()
        .From("People")
        .SelectMany("Firstname", "Surname", "Age", "Gender")
        .ToString();

    Console.WriteLine(query.ToString());
}
```

```
SELECT
Firstname,
Surname,
Age,
Gender
FROM People
```

I'd personally prefer calling `Select` method multiple times.

You might be wondering, why multiple selection is not just another *Select* overload. It's because *Select* has a more commonly used overload to select a column with alias:

```
public SqlQuery Select(string expression, string alias)
```

```
void Main()
{
    var query = new SqlQuery()
        .Select("(Firstname + ' ' + Surname)", "Fullname")
        .From("People")
        .ToString();

    Console.WriteLine(query.ToString());
}
```



```
SELECT
(Firstname + ' ' + Surname) AS [Fullname]
FROM People
```

## From Method

```
public SqlQuery From(string table)
```

SqlQuery.From method should be called at least once (and usually once).

..and it is recommended to be called first.

When you call it a second time, table name will be added to FROM statement with a comma between. Thus, it will be a CROSS JOIN:

```
void Main()
{
    var query = new SqlQuery()
        .From("People")
        .From("City")
        .From("Country")
        .Select("Firstname")
        .Select("Surname")
        .OrderBy("Age");

    Console.WriteLine(query.ToString());
}
```

```
SELECT
Firstname,
Surname
FROM People, City, Country
ORDER BY Age
```

## Using Alias Object with SqlQuery

It is common to use table aliases when number of referenced tables increase and our queries become longer:

```

void Main()
{
    var query = new SqlQuery()
        .From("Person p")
        .From("City c")
        .From("Country o")
        .Select("p.Firstname")
        .Select("p.Surname")
        .Select("c.Name", "CityName")
        .Select("o.Name", "CountryName")
        .OrderBy("p.Age")
        .ToString();

    Console.WriteLine(query.ToString());
}

```

```

SELECT
p.Firstname,
p.Surname,
c.Name AS [CityName],
o.Name AS [CountryName]
FROM Person p, City c, Country o
ORDER BY p.Age

```

Although it works like this, it is better to define `p`, `c`, and `o` as *Alias* objects.

```

var p = new Alias("Person", "p");

```

Alias object is like a short name assigned to a table. It has an indexer and operator overloads to generate SQL member access expressions like `p.Surname`.

```

void Main()
{
    var p = new Alias("Person", "p");
    Console.WriteLine(p + "Surname"); // + operator overload
    Console.WriteLine(p["Firstname"]); // through indexer
}

```

```

p.Surname
p.Firstname

```

Unfortunately C# member access operator (`.`) can't be overridden, so we had to use (`+`). A workaround could be possible with `dynamic`, but it would perform poorly.

Let's modify our query making use of Alias objects:

```
void Main()
{
    var p = new Alias("Person", "p");
    var c = new Alias("City", "c");
    var o = new Alias("Country", "o");

    var query = new SqlQuery()
        .From(p)
        .From(c)
        .From(o)
        .Select(p + "Firstname")
        .Select(p + "Surname")
        .Select(c + "Name", "CityName")
        .Select(o + "Name", "CountryName")
        .OrderBy(p + "Age")
        .ToString();

    Console.WriteLine(query.ToString());
}
```

```
SELECT
p.Firstname,
p.Surname,
c.Name AS [CityName],
o.Name AS [CountryName]
FROM Person p, City c, Country o
ORDER BY p.Age
```

As seen above, result is the same, but the code we wrote is a bit longer. So what is the advantage of using an alias?

If we had a list of constants with field names...

```
void Main()
{
    const string Firstname = "Firstname";
    const string Surname = "Surname";
    const string Name = "Name";
    const string Age = "Age";

    var p = new Alias("Person", "p");
    var c = new Alias("City", "c");
    var o = new Alias("Country", "o");
    var query = new SqlQuery()
        .From(p)
        .From(c)
        .From(o)
        .Select(p + Firstname)
        .Select(p + Surname)
        .Select(c + Name, "CityName")
        .Select(o + Name, "CountryName")
        .OrderBy(p + Age)
        .ToString();

    Console.WriteLine(query.ToString());
}
```

...we would take advantage of intellisense feature and have some more compile time checks.

Obviously, it is not logical and easy to define field names for every query. This should be in a central location, or our entity declarations.

Let's create a poor mans simple ORM using Alias:

```
public class PeopleAlias : Alias
{
    public PeopleAlias(string alias)
        : base("People", alias) { }

    public string ID { get { return this["ID"]; } }
    public string Firstname { get { return this["Firstname"]; } }
    public string Surname { get { return this["Surname"]; } }
    public string Age { get { return this["Age"]; } }
}

public class CityAlias : Alias
{
    public CityAlias(string alias)
        : base("City", alias) { }

    public string ID { get { return this["ID"]; } }
    public string CountryID { get { return this["CountryID"]; } }
    public new string Name { get { return this["Name"]; } }
}

public class CountryAlias : Alias
{
    public CountryAlias(string alias)
        : base("Country", alias) { }

    public string ID { get { return this["ID"]; } }
    public new string Name { get { return this["Name"]; } }
}

void Main()
{
    var p = new PeopleAlias("p");
    var c = new CityAlias("c");
    var o = new CountryAlias("o");
    var query = new SqlQuery()
        .From(p)
        .From(c)
        .From(o)
        .Select(p.Firstname)
        .Select(p.Surname)
        .Select(c.Name, "CityName")
        .Select(o.Name, "CountryName")
        .OrderBy(p.Age)
        .ToString();

    Console.WriteLine(query.ToString());
}
```

Now we have a set of table alias classes with field names and they can be reused in all queries.

This is just a sample to explain aliases. I don't recommend writing such classes. Entities offers much more.

In sample above, we used *SqlQuery.From* overload that takes an *Alias* parameter:

```
public SqlQuery From(Alias alias)
```

When this method is called, table name and its aliased name is added to *FROM* statement of query.

## OrderBy Method

```
public SqlQuery OrderBy(string expression, bool desc = false)
```

OrderBy can also be called with a field name or expression like Select.

If you assign *desc* optional argument as true, `DESC` keyword is appended to the field name or expression.

By default, OrderBy appends specified expressions to end of the ORDER BY statement. Sometimes, you might want to insert an expression/field to start.

For example, you might have a query with some predefined order, but if user orders by a column in a grid, name of the column should be inserted at index 0.

```
public SqlQuery OrderByFirst(string expression, bool desc = false)
```

```
void Main()
{
    var query = new SqlQuery()
        .Select("Firstname")
        .Select("Surname")
        .From("Person")
        .OrderBy("PersonID");

    query.OrderByFirst("Age");

    Console.WriteLine(query.ToString());
}
```

```
SELECT
Firstname,
Surname
FROM Person
ORDER BY Age, PersonID
```

## Distinct Method

```
public SqlQuery Distinct(bool distinct)
```

Use this method to prepend a DISTINCT keyword before SELECT statement.

```
void Main()
{
    var query = new SqlQuery()
        .Select("Firstname")
        .Select("Surname")
        .From("Person")
        .OrderBy("PersonID")
        .Distinct(true);

    Console.WriteLine(query.ToString());
}
```

```
SELECT DISTINCT
Firstname,
Surname
FROM Person
ORDER BY PersonID
```

## GroupBy Method

```
public SqlQuery GroupBy(string expression)
```

GroupBy works similar to OrderBy but it doesn't have a GroupByFirst variant.

```
SELECT
Firstname,
Lastname,
Count(*)
FROM Person
GROUP BY Firstname, LastName
```

```
SELECT
Firstname,
Lastname,
Count(*)
FROM Person
GROUP BY Firstname, LastName
```

## Having Method

```
public SqlQuery Having(string expression)
```

Having can be used with GroupBy (though it doesn't check for GroupBy) and appends expression to the end of HAVING statement.

```
void Main()
{
    var query = new SqlQuery()
        .From("Person")
        .Select("Firstname")
        .Select("Lastname")
        .Select("Count(*)")
        .GroupBy("Firstname")
        .GroupBy("LastName")
        .Having("Count(*) > 5");

    Console.WriteLine(query.ToString());
}
```

```
SELECT
Firstname,
Lastname,
Count(*)
FROM Person
GROUP BY Firstname, LastName
HAVING Count(*) > 5
```



## Paging Operations (SKIP / TAKE / TOP / LIMIT)

```
public SqlQuery Skip(int skipRows)

public SqlQuery Take(int rowCount)
```

SqlQuery has paging methods similar to LINQ Take and Skip.

These are mapped to SQL keywords depending on database type.

As SqlServer versions before 2012 doesn't have a SKIP equivalent, to use SKIP your query should have at least one ORDER BY statement as ROW\_NUMBER() will be used. This is not required if you are using SqlServer 2012+ dialect.

```
void Main()
{
    var query = new SqlQuery()
        .From("Person")
        .Select("Firstname")
        .Select("Lastname")
        .Select("Count(*)")
        .OrderBy("PersonId")
        .Skip(100)
        .Take(50);

    Console.WriteLine(query.ToString());
}
```

```
SELECT
Firstname,
Lastname,
Count(*)
FROM Person
ORDER BY PersonId OFFSET 100 ROWS FETCH NEXT 50 ROWS ONLY
```

In this sample we are using the default SQLServer2012 dialect.

## Database Dialect Support

In our paging sample, SqlQuery used a syntax that is compatible with Sql Server 2012.

With Dialect method, we can change the server type that SqlQuery targets:

```
public SqlQuery Dialect(ISqlDialect dialect)
```

As of writing, these are the list of dialect types supported:

```
FirebirdDialect
PostgresDialect
SqliteDialect
SqlServer2000Dialect
SqlServer2005Dialect
SqlServer2012Dialect
```

If we wanted to target our query to Sql Server 2005:

```
void Main()
{
    var query = new SqlQuery()
        .Dialect(SqlServer2005Dialect.Instance)
        .From("Person")
        .Select("Firstname")
        .Select("Lastname")
        .Select("Count(*)")
        .OrderBy("PersonId")
        .Skip(100)
        .Take(50);

    Console.WriteLine(query.ToString());
}
```

```
SELECT * FROM (
SELECT TOP 150
Firstname,
Lastname,
Count(*), ROW_NUMBER() OVER (ORDER BY PersonId) AS __num__
FROM Person) __results__ WHERE __num__ > 100
```

With SqliteDialect.Instance, output would be:

```
SELECT
Firstname,
Lastname,
Count(*)
FROM Person
ORDER BY PersonId LIMIT 50 OFFSET 100
```

If you are using only one type of database server with your application, you may avoid having to choose a dialect every time you start a query by setting the default dialect:

```
SqlSettings.DefaultDialect = SqliteDialect.Instance;
```

Write code above in your application start method, e.g. `global.asax.cs`.

# Criteria Objects

When you are creating dynamic SQL for SELECT, UPDATE or DELETE, you might have to write complex WHERE statements.

Building these statements using string concatenation is possible but it is tedious to avoid syntax errors and opens your code to SQL injection attacks.

Using parameters might solve SQL injection problems but it involves too much manual work to add parameters.

Luckily, Serenity has a criteria system that helps you build parameterized queries with a syntax similar LINQ expression trees.

Serenity criterias are implemented by utilizing operator overloading features of C#, unlike LINQ which uses expression trees.

Let's write a basic SQL where statement as string first:

```
new SqlQuery()  
    .From("MyTable")  
    .Select("Name")  
    .Where("Month > 5 AND Year < 2015 AND Name LIKE N'%a%'")
```

and same statement using criteria objects:

```
new SqlQuery()  
    .From("MyTable")  
    .Select("Name")  
    .Where(  
        new Criteria("Month") > 5 &  
        new Criteria("Year") < 4 &  
        new Criteria("Name").Contains("a")
```

It looks a bit longer, but it uses parameters

```
SELECT  
    Name  
FROM  
    MyTable  
WHERE  
    Month > @p1 AND  
    Year < @p2 AND  
    Name LIKE N'%a%'
```

and you could write it with intellisense if you had an entity:

```
var m = MyTableRow.Fields;  
new SqlQuery()  
    .From(m)  
    .Select(m.Name)  
    .Where(  
        m.Month > 5 &  
        m.Year < 4 &  
        m.Name.Contains("a")
```

Here we didn't have to use *new Criteria()* because field objects also has operator overloads that builds criteria.

## BaseCriteria Object

BaseCriteria is the base class for all types of criteria objects.

It has overloads for several C# operators, including `>`, `<`, `&`, `|` that can be used to build complex criteria using C# expressions.

BaseCriteria doesn't have a constructor of itself so you need to create one of the objects that derive from it. *Criteria* is the most common one that you might use.

## Criteria Object

Criteria is a simple object that contains an SQL expression as a string, which is usually a field name.

```
new Criteria("MyField")
```

It can also contain an SQL expression (though not recommended this way)

```
new Criteria("a + b")
```

This parameter is not syntax checked, so it is possible to build a criteria with invalid expression:

```
new Criteria("Some invalid expression()//'^^')
```

## AND (&) Operator

It is possible to AND two criteria objects with C# `&` operator:

```
new Criteria("Field1 > 5") &  
new Criteria("Field2 < 4")  
`
```

Please notice that we are not using shortcircuit `&&` operator here.

This creates a new criteria object (BinaryCriteria) with operator (AND) and reference to these two criterias. It doesn't modify original criteria objects.

BinaryCriteria is similar to BinaryExpression in expression trees

It's SQL output would be:

```
Field1 > 5 AND Field2 < 4
```

It is also possible to use C# `&=` operator:

```
BaseCriteria c = new Criteria("Field1 > 5");  
c &= new Criteria("Field2 < 4")
```

BaseCriteria is the base class for all criteria object types. If we used `Criteria c = ...` in the first line, we would have a compile time error on second line as `&` operator returns a BinaryCriteria object, which is not assignable to a Criteria object.

## OR (|) Operator

This is similar to AND operator, though it uses OR.

```
new Criteria("Field1 > 5") |  
new Criteria("Field2 < 4")  
`
```

```
Field1 > 5 OR Field2 < 4
```

## Parenthesis Operator (~)

When you are using several AND/OR statements, you might want to put parenthesis.

```
new Criteria("Field1 > 5") &
(new Criteria("Field2 > 7") | new Criteria("Field2 < 3"))
```

But this won't work with criteria objects, as output of above criteria would be:

```
Field1 > 5 AND Field2 > 7 OR Field2 < 3
```

Information here applies to Serenity versions before 1.9.8. After this version Serenity puts parenthesis around all binary criteria (AND OR etc) even if you don't use parenthesis.

So only use ~ if you want to put an explicit parenthesis somewhere.

What happened to our parenthesis? Let's try putting more parenthesis.

```
new Criteria("Field1 > 5") &
((((new Criteria("Field2 > 7") | new Criteria("Field2 < 3")))))
```

Still:

```
Field1 > 5 AND Field2 > 7 OR Field2 < 3
```

C# doesn't provide a way to overload parenthesis, it just uses them to determine calculation order, so Serenity criteria has no idea if you used them with parenthesis or not.

We have to use a special operator, ~ (which is actually two's complement in C#):

```
new Criteria("Field1 > 5") &
~(new Criteria("Field2 > 7") | new Criteria("Field2 < 3"))
```

Now SQL looks like we hoped before:

```
Field1 > 5 AND (Field2 > 7 OR Field2 < 3)
```

As Serenity 1.9.8+ auto parenthesis binary criteria, above expression would actually be:

```
(Field1 > 5) AND (((Field2 > 7) OR (Field2 < 3)))
```

## Comparison Operators (>, >=, <, <=, ==, !=)

The most of C# comparison operators are overloaded, so you can use them as is with criteria.

```
new Criteria("Field1") == new Criteria("1") &
new Criteria("Field2") != new Criteria("2") &
new Criteria("Field3") > new Criteria("3") &
new Criteria("Field4") >= new Criteria("4") &
new Criteria("Field5") < new Criteria("5") &
new Criteria("Field6") <= new Criteria("6")
```

```
Field1 == 1 AND
Field2 <> 2 AND
Field3 > 3 AND
Field4 >= 4 AND
Field5 < 5 AND
Field6 <= 6
```

## Inline Values

When one side of a comparison operator is a criteria and other side is an integer, string, date, guid etc. value, it is converted a parameter criteria.

```
new Criteria("Field1") == 1 &
new Criteria("Field2") != "ABC" &
new Criteria("Field3") > DateTime.Now &
new Criteria("Field4") >= Guid.NewGuid() &
new Criteria("Field5") < 5L
```

```
Field1 == @p1 AND
Field2 <> @p2 AND
Field3 > @p3 AND
Field4 >= @p4 AND
Field5 < @p5
```

These parameters has corresponding values, when a query containing this criteria is sent to SQL.

Automatic parameter numbering starts from 1 by default, but last number is stored in the query the criteria is used with, so numbers might change.

Let's use this criteria in a query:



```
new SqlQuery()  
  .From("MyTable")  
  .Select("Field999")  
  .Where(new Criteria("FirstOne") >= 999)  
  .Where(new Criteria("SecondOne") >= 999)  
  .Where(  
    new Criteria("Field1") == 1 &  
    new Criteria("Field2") != "ABC" &  
    new Criteria("Field3") > DateTime.Now &  
    new Criteria("Field4") >= Guid.NewGuid() &  
    new Criteria("Field5") < 5L  
  )  
)
```

```
SELECT  
  Field999  
FROM  
  MyTable  
WHERE  
  FirstOne >= @p1 AND -- @p1 = 999  
  SecondOne >= @p2 AND -- @p2 = 999  
  Field1 == @p3 AND -- @p3 = 1  
  Field2 <> @p4 AND -- @p4 = N'ABC'  
  Field3 > @p5 AND -- @p5 = '2016-01-31T01:16:23'  
  Field4 >= @p6 AND -- @p6 = '23123-DEFCD-....'  
  Field5 < @p7 -- @p7 = 5
```

Here the same criteria that listed before, used parameter numbers starting from 3, instead of 1. Because prior 2 numbers were used for other WHERE statements coming before it.

So parameter numbering uses the query as context. You shouldn't make assumptions about what parameter name will be.

## ParamCriteria and Explicit Param Names

If you want to use some explicitly named parameter, you can make use of ParamCriteria:

```
new SqlQuery()  
  .From("SomeTable")  
  .Select("SomeField")  
  .Where(new Criteria("SomeField") <= new ParamCriteria("@myparam"))  
  .Where(new Criteria("SomeOtherField") == new ParamCriteria("@myparam"))  
  .SetParam("@myparam", 5);
```

Here we set param value using SetParam extension of SqlQuery.

We could also declare this param beforehand and reuse it:

```
var myParam = new ParamCriteria("@myparam");

new SqlQuery()
    .From("SomeTable")
    .Select("SomeField")
    .Where(new Criteria("SomeField") <= myParam)
    .Where(new Criteria("SomeOtherField") == myParam)
    .SetParam(myParam.Name, 5);
```

## ConstantCriteria

If you don't want to use parameterized queries, you may put your values as `ConstantCriteria` objects. They will not be converted to auto parameters.

```
new SqlQuery()
    .From("MyTable")
    .Select("MyField")
    .Where(
        new Criteria("Field1") == new ConstantCriteria(1) &
        new Criteria("Field2") != new ConstantCriteria("ABC")
    )
```

```
SELECT
    MyField
FROM
    MyTable
WHERE
    FirstOne >= 1
    SecondOne >= N'ABC'
```

## Null Comparison

In SQL, comparing against NULL values using operators like `==` , `!=` returns NULL. You should use `IS NULL` or `IS NOT NULL` for such comparisons.

Criteria objects don't overload comparisons against null (or object), so you may get errors if you try to write expressions like below:

```
new Criteria("a") == null; // what is type of null?

int b? = null;
new Criteria("c") == b; // no overload for nullable types
```

These could be written using `IsNull` and `Nullable.Value` methods:

```
new Criteria("a").IsNull();
new Criteria("a").IsNotNull();
int? b = 5;
new Criteria("c") == b.Value;
```

If you are desperate to write `Field = NULL`, you could do this:

```
new Criteria("Field") == new Criteria("NULL")
```

## LIKE Operators

Criteria has methods *Like*, *NotLike*, *StartsWith*, *EndsWith*, *Contains*, *NotContains* to help with LIKE operations.

```
new Criteria("a").Like("__C%") &
new Criteria("b").NotLike("D%") &
new Criteria("c").StartsWith("S") &
new Criteria("d").EndsWith("X") &
new Criteria("e").Contains("This") &
new Criteria("f").NotContains("That")
```

```
a LIKE @p1 AND -- @p1 = N'__C%'
b NOT LIKE @p2 AND -- @p2 = N'D%'
c LIKE @p3 AND -- @p3 = 'S%'
d LIKE @p4 AND -- @p4 = N'X%'
e LIKE @p5 AND -- @p5 = N'%This%'
f NOT LIKE @p6 -- @p6 = N'%That%'
```

## IN and NOT IN Operators

Use an inline array to use `IN` or `NOT IN`:

```
new Criteria("A").In(1, 2, 3, 4, 5)
```

```
A IN (@p1, @p2, @p3, @p4, @p5)
-- @p1 = 1, @p2 = 2, @p3 = 3, @p4 = 4, @p5 = 5
```

```
new Criteria("A").NotIn(1, 2, 3, 4, 5)
```

```
A NOT IN (@p1, @p2, @p3, @p4, @p5)
-- @p1 = 1, @p2 = 2, @p3 = 3, @p4 = 4, @p5 = 5
```

You may also pass any enumerable to IN method:

```
IEnumerable<int> x = new int[] { 1, 3, 5, 7, 9 };
new Criteria("A").In(x);
```

```
A IN (1, 3, 5, 7, 9)
-- @p1 = 1, @p2 = 3, @p3 = 5, @p4 = 7, @p5 = 9
```

It is also possible to use a subquery:

```
var query = new SqlQuery()
    .From("MyTable")
    .Select("MyField");

query.Where("SomeID").In(
    query.SubQuery()
        .From("SomeTable")
        .Select("SomeID")
        .Where(new Criteria("Balance") < 0));
```

```
SELECT
    MyField
FROM
    MyTable
WHERE
    SomeID IN (
        SELECT
            SomeID
        FROM
            SomeTable
        WHERE
            Balance < @p1 -- @p1 = 0
    )
```

## NOT Operator

Use C# ! (not) operator to use NOT:

```
!(new Criteria("a") >= 5)
```

```
NOT (a >= @p1) -- @p1 = 5
```

## Usage with Field Objects

We have used Criteria object constructor so far to build criteria. Field objects also has similar overloads, so they can be used in place of them.

For example, using Order, Detail and Customer rows from Northwind sample:

```
var o = OrderRow.Fields.As("o");
var od = OrderDetailRow.Fields.As("od");
var c = CustomerRow.Fields.As("c");
var query = new SqlQuery()
    .From(o)
    .Select(o.CustomerID);

query.Where(
    o.CustomerCountry == "France" &
    o.ShippingState == 1 &
    o.CustomerID.In(
        query.SubQuery()
            .From(c)
            .Select(c.CustomerID)
            .Where(c.Region == "North")) &
    new Criteria(
        query.SubQuery()
            .From(od)
            .Select(Sql.Sum(od.LineTotal.Expression))
            .Where(od.OrderID == o.OrderID)) >= 1000);
```

Its output would be:

```
SELECT
    o.CustomerID AS [CustomerID]
FROM
    Orders o
LEFT JOIN
    Customers o_c ON (o_c.CustomerID = o.CustomerID)
WHERE
    o_c.[Country] = @p2
    AND (CASE WHEN
        o.[ShippedDate] IS NULL THEN 0
        ELSE 1
        END) = @p3
    AND o.CustomerID IN (
        SELECT
            c.CustomerID AS [CustomerID]
        FROM
            Customers c
        WHERE
            c.Region = @p1)
    AND (SELECT
        SUM((od.[UnitPrice] * od.[Quantity] - od.[Discount]))
    FROM
        [Order Details] od
    WHERE
        od.OrderID = o.OrderID) >= @p4
```

# Connections and Transactions

Serenity uses simple ADO.NET data access objects, like `SqlConnection`, `DbCommand` etc.

It provides some basic helpers to create a connection, add parameters, execute queries etc.

## SqlConnection Class

[namespace: *Serenity.Data*, assembly: *Serenity.Data*]

This class contains static functions to create a connection, and control it in a database agnostic way.

## SqlConnection.NewByKey method

```
public static IDbConnection NewByKey(string connectionKey)
```

Use this method to get a new `IDbConnection` for a connection string defined in application configuration file (e.g. `app.config` or `web.config`).

```
using (var connection = SqlConnections.NewByKey("Default"))  
{  
    // ...  
}
```

Try to always wrap connections in a using block...

This reads connection string with "Default" key from `web.config`, and creates a new connection using *ProviderName* information that is also specified in connection setting. For example, if *ProviderName* is "System.Data.SqlClient" this creates a new `SqlConnection` object.

You usually don't have to open connections explicitly as they are automatically opened when needed (as long as you use Serenity extensions).

## SqlConnection.NewFor< TClass > method

If you don't want to memorize connection string keys, but instead reuse information on a row (in form of a *ConnectionKey* attribute), you may prefer this variant.

Looking on top of a Row class, you may spot `ConnectionString` attribute generated by Sergen:

```
[ConnectionString("Northwind")]  
public sealed class CustomerRow : Row, IIdRow, INameRow  
{  
}
```

When you are going to query for customers, instead of hardcoding "Northwind", you may reuse this information from a `CustomerRow`:

```
using (var connection = SqlConnections.NewFor<CustomerRow>())  
{  
    return connection.List<CustomerRow>();  
}
```

This corresponds to `SqlConnections.NewByKey("Northwind")`.

Here we didn't have to open the connection, as `List` extension method opens it automatically.

The class used with this method doesn't have to be a `Row`, any class with a `ConnectionString` attribute would work, even though it would be a row most of the time.

## SqlConnections.New method

```
public static IDbConnection New(string connectionString, string providerName)
```

You may sometimes want to create a connection that doesn't exist in your configuration file.

```
using (var connection = SqlConnections.New(  
    "Data Source=(localdb)\v11.0; Initial Catalog=Northwind;  
    Integrated Security=true", "System.Data.SqlClient"))  
{  
    // ...  
}
```

Here we have to specify connection string and the provider name like "System.Data.SqlClient".

You might be asking yourself "why this method instead of simply typing `new SqlConnection()`", see next topic for advantages of these.

## WrappedConnection



All methods we saw so far returns an IDbConnection object. You'd expect it to be a SqlConnection, FirebirdConnection etc, but thats not exactly true.

The IDbConnection object you receive is a Serenity specific WrappedConnection object that actually contains an underlying SqlConnection or FirebirdConnection etc.

This helps Serenity provide some features like auto open, dialect support, default transactions, unit of work pattern, overriding connections for testability etc.

You may not notice these details while working with returned IDbConnection instances, they'll act just like the underlying connections, but you should prefer SqlConnections methods to create connections, otherwise you might lose some of these listed features.

## UnitOfWork and IUnitOfWork

UnitOfWork is a simple object that just contains a transaction reference. It has two extra events that we can attach to OnCommit and OnRollback.

Let's say we are creating tasks, and some e-mails should be sent in case these tasks are saved to database succesfully.

If we hurry and send these e-mails before transaction is committed, we might end up with e-mails that are sent for non-existent tasks in case transaction fails. So we should only send e-mail if transaction is committed successfully, e.g. in OnCommit event.

You might say then Commit transaction first and send e-mails right after, but what if our Create Task service call is just a step of a larger operation, so we are not controlling the transaction and it should be committed after all steps are success.

Another scenario is about uploading files. This time we are updating some File entity, and let's say we replace an old file with uploaded new file. If we again hurry and delete old file before transaction outcome is clear, and transaction fails eventually, we'll end up with a file entity without an actual old file in disk. So, we should actually delete file and replace it with the new file in OnCommit event, and remove uploaded file in OnRollback event.

```
void SomeBatchOperation()
{
    using (var connection = SqlConnections.NewByKey("Default"))
    using (var uow = new UnitOfWork(connection))
    {
        // here we are in a transaction context
        // create several tasks in transaction
        CreateATask(new TaskRow { ... });
        CreateATask(new TaskRow { ... });
        //...

        // commit the transaction
        // if any exception occurs here or at prior
        // lines transaction will rollback
        // and no e-mails will be sent
        uow.Commit();
    }
}

void CreateATask(IUnitOfWork uow, TaskRow task)
{
    // insert task using connection wrapped inside IUnitOfWork
    // this will automatically run in transaction context
    uow.Connection.Insert(task);

    uow.OnCommit += () => {
        // send e-mail for this task now, this method will only
        // be called if transaction commits successfully
    };

    uow.OnRollback += () => {
        // optional, do something else if it fails
    };
}
```

# Working With Other Database Types

Serenity has a dialect system for working with database types other than Sql Server.

If you need to support multiple database types, just by changing connection strings in web.config, you should be careful about not using database specific functions in expressions and avoid using reserved words.

## Warning About CONCAT and Other Similar Expressions In Rows

Serene has to support a variety of database engines, including MySQL, Postgress etc. These databases don't have a string plus (+) operator like MsSqlServer. Thus, in Northwind, CONCAT function is used in place of '+' operator:

```
[Expression("CONCAT(T0.[FirstName], CONCAT(' ', T0.[LastName]))")]
public String FullName
{
    get { return Fields.FullName[this]; }
    set { Fields.FullName[this] = value; }
}
```

CONCAT is available after Sql Server 2012. So if you are going to use an older version of SQL server, e.g. 2005 or 2008, replace these expressions with such:

```
[Expression("T0.[FirstName] + ' ' + T0.[LastName]")]
public String FullName
{
    get { return Fields.FullName[this]; }
    set { Fields.FullName[this] = value; }
}
```

## Set Database Dialect for Connections

Serenity auto detects dialect for a connection by using the providerName in web.config.

Sometimes, automatic dialect detection using providerName may not work you, or you might want to use SqlServer2000 or SqlServer2005 dialect for some connections.

Even though it is possible to set a default global dialect, this doesn't override automatic detection):

```
SqlSettings.DefaultDialect = SqlServer2005Dialect.Instance;
```

As provider name for "Northwind" and "Default" connections is "System.Data.SqlClient", Serenity will automatically set their dialects to SqlServer2012, even if you override global dialect.

But it is possible to change dialect on connection key basis:

```
public static partial class SiteInitialization
{
    public static void ApplicationStart()
    {
        try
        {
            SqlConnections.GetConnectionString("Default").Dialect =
                SqlServer2005Dialect.Instance;

            SqlConnections.GetConnectionString("Northwind").Dialect =
                SqlServer2005Dialect.Instance;
        }
    }
}
```

It is also possible to set this through an application configuration entry (recommended):

```
<configuration>
  <appSettings>
    <add key="ConnectionSettings" value="{
      Default: {
        Dialect: 'SqlServer2005'
      },
      Northwind: {
        Dialect: 'Postgres'
      }
    }" />
```



## Dialect Based Expressions

Sometimes it might not be possible to use a common expression. For example, Sqlite has no CONCAT operator.

Serenity 2.8.1+ supports dialect based expressions, e.g.

```
[DisplayName("FullName"), QuickSearch]
[Expression("CONCAT(T0.[FirstName], CONCAT(' ', T0.[LastName]))")]
[Expression("(T0.FirstName || ' ' || T0.LastName)", Dialect = "Sqlite")]
public String FullName
{
    get { return Fields.FullName[this]; }
    set { Fields.FullName[this] = value; }
}
```

Here, as the first Expression has no dialect, it will be used for any database type, unless the connection corresponding to this row has dialect of *Sqlite*, e.g. it is a *System.Data.Sqlite* connection.

## How Dialect for Row is Determined

To determine dialect type for a row, the *ConnectionKey* attribute on row is used (if any), otherwise the default dialect (*Sq/Settings.DefaultDialect*) is used.

Expression for a field is determined (fixed) at *application start*, so it is not possible to switch expressions by switching connections or dialects.

It is also possible to specify multiple dialects:

```
[DisplayName("FullName"), QuickSearch]
[Expression("CONCAT(T0.[FirstName], CONCAT(' ', T0.[LastName]))")]
[Expression("T0.[FirstName] + ' ' + T0.[LastName]",
    Dialect = "SqlServer2000,SqlServer2005")]
[Expression("(T0.FirstName || ' ' || T0.LastName)",
    Dialect = "Sqlite,MySQL,Postgres")]
public String FullName
{
    get { return Fields.FullName[this]; }
    set { Fields.FullName[this] = value; }
}
```

## Dialect Matching

ISqlDialect interface has a *ServerType* property. It is *Postgres* for *PostgresDialect*, *Sq/Server* for *Sq/Server2012Dialect*, *Sq/Server2008Dialect* and *Sq/Server2005Dialect*.

For an expression dialect to match a connection dialect, it should start with the *ServerType* and/or the class name of the connection dialect (e.g. *Sq/Server2012Dialect*).

If multiple dialect types match a targeted expression, the one with the longest name matches.

Let's say we wrote these two expressions:

```
[Expression("CONCAT(T0.[FirstName], T0.[LastName])", Dialect = "SqlServer")]  
[Expression("T0.[FirstName] + T0.[LastName]", Dialect = "SqlServer200")]
```

If connection dialect is *Sq/Server2008*, both expressions would match, but as *Sq/Server200* is a longer match than *Sq/Server*, second expression will be used.

If connection dialect is *Sq/Server2012*, only the first expression would match.

# PostgreSQL

## Registering Npgsql Provider

PostgreSQL has a .NET provider named Npgsql. You need to first install it in MyProject.Web:

```
Install-Package Npgsql -Project MyProject.Web
```

If you didn't install this provider in GAC/machine.config before, or don't want to install it there, you need to register it in web.config file:

```
<configuration>
  // ...
  <system.data>
    <DbProviderFactories>
      <remove invariant="Npgsql"/>
      <add name="Npgsql Data Provider"
          invariant="Npgsql"
          description=".Net Data Provider for PostgreSQL"
          type="Npgsql.NpgsqlFactory, Npgsql, Culture=neutral,
              PublicKeyToken=5d8b90d52f46fda7"
          support="FF" />
    </DbProviderFactories>
  </system.data>
  // ...
```

## Setting Connection Strings

Next step is to replace connection strings for databases you want to use with Postgres:

```
Make sure you replace connection string parameters with values for your server
```

```
<connectionStrings>
  <add name="Default" connectionString="
      Server=127.0.0.1;Database=serene_default_v1;
      User Id=postgres;Password=yourpassword;"
      providerName="Npgsql" />

  <add name="Northwind" connectionString="
      Server=127.0.0.1;Database=serene_northwind_v1;
      User Id=postgres;Password=yourpassword;"
      providerName="Npgsql" />
</connectionStrings>
```



## Setting Connection Strings - .Net Core appsettings.json

```
"Data": {
  "Default": {
    "ConnectionString": "Server=127.0.0.1;Database=serene_default_v1;User Id=postgres;Password=yourpassword;",
    "ProviderName": "Npgsql"
  },
  "Northwind": {
    "ConnectionString": "Server=127.0.0.1;Database=serene_northwind_v1;User Id=postgres;Password=yourpassword;",
    "ProviderName": "Npgsql"
  }
},
```

Please use lowercase database names like `serene_default_v1` as Postgres will always convert it to lowercase.

Provider name must be `Npgsql` for Serenity to auto-detect dialect.

## Notes About Identifier Case Sensitivity

PostgreSQL is case sensitive for identifiers.

FluentMigrator automatically quotes all identifiers, so tables and column names in database will be quoted and case sensitive. This might cause problems when tables/columns are tried to be selected without quoted identifiers.

One option is to always use lowercase identifiers in migrations, but such naming scheme won't look so nice for other database types, thus we didn't prefer this way.

To prevent such problems with Postgres, Serenity has an automatic quoting feature, to resolve compability with Postgres/FluentMigrator, which should be enabled in application start method of SiteInitialization.cs:

```
public static void ApplicationStart()
{
    try
    {
        SqlSettings.AutoQuotedIdentifiers = true;
        Serenity.Web.CommonInitialization.Run();
    }
}
```

Make sure it is before `CommonInitialization.Run` line

This setting automatically quotes column names in entities, but not manually written expressions (with `Expression` attribute for example).

Use brackets `[]` for identifiers in expressions if you want to support multiple database types. Serenity will automatically convert brackets to database specific quote type before running queries.

You might also prefer to use double quotes in expressions, but it might not be compatible with other databases like MySQL.

## Registering PostgreSQL DbProviderFactory

Open the `Startup.cs` file under `{SerenityProject}/Initialization/` and uncomment the last line, as shown below.

```
public static void RegisterDataProviders()
{
    // DbProviderFactories.RegisterFactory("System.Data.SqlClient",
    //   SqlConnectionFactory.Instance);
    // DbProviderFactories.RegisterFactory("Microsoft.Data.Sqlite",
    //   Microsoft.Data.Sqlite.SqliteFactory.Instance);
    // to enable FIREBIRD: add FirebirdSql.Data.FirebirdClient reference, set connectio
ns, and uncomment line below
    // DbProviderFactories.RegisterFactory("FirebirdSql.Data.FirebirdClient",
    //   FirebirdSql.Data.FirebirdClient.FirebirdClientFactory.Instance);
    // to enable MYSQL: add MySql.Data reference, set connections, and uncomment line b
elow
    // DbProviderFactories.RegisterFactory("MySql.Data.MySqlClient",
    //   MySql.Data.MySqlClient.MySqlClientFactory.Instance);
    // to enable POSTGRES: add Npgsql reference, set connections, and uncomment line be
low
    DbProviderFactories.RegisterFactory("Npgsql", Npgsql.NpgsqlFactory.Instance);
}
```

## Setting Default Dialect

This step is optional.

Serenity automatically determines which dialect to use, by looking at *providerName* of connection strings.

It can even work with multiple database types at the same time.

For example, Northwind might stay in Sql Server, while Default database uses PostgreSQL.

But, if you are going to use only one database type per site, you can register which you are going to use by default in SiteInitialization:

```
public static void ApplicationStart()
{
    try
    {
        SqlSettings.DefaultDialect = PostgresDialect.Instance;
        SqlSettings.AutoQuotedIdentifiers = true;
        Serenity.Web.CommonInitialization.Run();
    }
}
```

Default dialect is used when the dialect for a connection / entity etc. couldn't be auto determined.

This setting doesn't override automatic detection, it is just used as fallback.

## Launching Application

Now launch your application, it should automatically create databases, if they are not created manually before.

## Configuring Code Generator

Sergen doesn't have reference to PostgreSQL provider, so if you want to use it to generate code, you must also register this provider with it.

Sergen.exe is an exe file, so you can't add a NuGet reference to it. We need to register this provider in application config file.

It is also possible to register the provider in GAC/machine.config and skip this step completely.

Locate Sergen.exe, which is under a folder like *packages/Serenity.CodeGenerator.1.8.6/tools* and create a file named `Sergen.exe.config` next to it with contents below:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.data>
    <DbProviderFactories>
      <remove invariant="Npgsql"/>
      <add name="Npgsql Data Provider"
          invariant="Npgsql"
          description=".Net Data Provider for PostgreSQL"
          type="Npgsql.NpgsqlFactory, Npgsql, Culture=neutral,
              PublicKeyToken=5d8b90d52f46fda7"
          support="FF" />
    </DbProviderFactories>
  </system.data>
  <appSettings>
    <add key="LoadProviderDLLs" value="Npgsql.dll"/>
  </appSettings>
</configuration>
```

Also copy Npgsql.dll to same folder where Sergen.exe resides. Now Sergen will be able to generate code for your Postgres tables.

You might want to remove `[public].` prefix for default schema from tablename/column expressions in generated rows if you want to be able to work with multiple databases.

# MySql

## .NET Framework

### Registering MySql Provider

MySQL has a .NET provider named `MySql.Data`. You need to first install it in `MyProject.Web`:

```
Install-Package MySql.Data -Project MyProject.Web
```

If you didn't install this provider in `GAC/machine.config` before, or don't want to install it there, you need to register it in `web.config` file (`MySql.Data` NuGet package already does this on install):

```
<configuration>
  // ...
  <system.data>
    <DbProviderFactories>
      <remove invariant="MySql.Data.MySqlClient"/>
      <add name="MySQL Data Provider"
          invariant="MySql.Data.MySqlClient"
          description=".Net Framework Data Provider for MySQL"
          type="MySql.Data.MySqlClient.MySqlClientFactory,
              MySql.Data, Culture=neutral,
              PublicKeyToken=c5687fc88969c44d" />
    </DbProviderFactories>
  </system.data>
  // ...
```

### Setting Connection Strings

Next step is to replace connection strings for databases you want to use with `MySql`:

```
Make sure you replace connection string parameters with values for your server
```

```
<connectionStrings>
  <add name="Default" connectionString="
    Server=localhost; Port=3306; Database=Serene_Default_v1;
    Uid=root; Pwd=yourpass"
    providerName="MySql.Data.MySqlClient" />

  <add name="Northwind" connectionString="
    Server=localhost; Port=3306; Database=Serene_Northwind_v1;
    Uid=root; Pwd=yourpass"
    providerName="MySql.Data.MySqlClient" />
</connectionStrings>
```

Provider name must be `MySql.Data.MySqlClient` for Serenity to auto-detect dialect. Read notes above to override default dialect.

MySQL uses lowercase database (schema) and table names, but doesn't have the case sensitivity problem we talked about Postgres.

## Configuring Code Generator

Sergen doesn't have reference to MySQL provider, so if you want to use it to generate code, you must also register this provider with it.

Sergen.exe is an exe file, so you can't add a NuGet reference to it. We need to register this provider in application config file.

It is also possible to register the provider in GAC/machine.config and skip this step completely.

Locate Sergen.exe, which is under a folder like `packages/Serenity.CodeGenerator.1.8.6/tools` and create a file named `Sergen.exe.config` next to it with contents below:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.data>
    <DbProviderFactories>
      <remove invariant="MySQL.Data.MySqlClient"/>
      <add name="MySQL Data Provider"
          invariant="MySQL.Data.MySqlClient"
          description=".Net Framework Data Provider for MySQL"
          type="MySQL.Data.MySqlClient.MySqlClientFactory,
              MySQL.Data, Culture=neutral,
              PublicKeyToken=c5687fc88969c44d" />
    </DbProviderFactories>
  </system.data>
  <appSettings>
    <add key="LoadProviderDLLs" value="MySQL.Data.dll"/>
  </appSettings>
</configuration>
```

Also copy `MySQL.Data.dll` to same folder where `Sergen.exe` resides. Now `Sergen` will be able to generate code for your MySQL tables.

## .NET Core

### Registering MySQL Provider

MySQL has a .NET provider named `MySQL.Data`. You need to first install it in `MyProject.AspNetCore`:

Open `project.json` and add package as follows:

```
{
  "dependencies": {
    // ...
    "Serenity.FluentMigrator.Runner": "1.6.903",
    "MySQL.Data": "7.0.6-IR31"
  },
}
```

Make sure you have `Serenity.FluentMigrator.Runner 1.6.903+`

Open `Initialization/Startup.cs` file, register this factory in `Serenity`:

```
DbProviderFactories.RegisterFactory(  
    "System.Data.SqlClient", SqlConnectionFactory.Instance);  
  
DbProviderFactories.RegisterFactory(  
    "MySql.Data.MySqlClient",  
    MySql.Data.MySqlClient.MySqlClientFactory.Instance);
```

## Configuring Code Generator

As of writing *dotnet-sergen* doesn't yet support any databases other than Sql Server.



# Sqlite

## Registering Sqlite Provider

Sqlite has a .NET provider named System.Data.Sqlite. You need to first install it in MyProject.Web:

```
Install-Package System.Data.SQLite.Core -Project MyProject.Web
```

If you didn't install this provider in GAC/machine.config before, or don't want to install it there, you need to register it in web.config file:

```
<configuration>
  // ...
  <system.data>
    <DbProviderFactories>
      <remove invariant="System.Data.SQLite"/>
      <add name="SQLite Data Provider"
          invariant="System.Data.SQLite"
          description=".Net Framework Data Provider for SQLite"
          type="System.Data.SQLite.SQLiteFactory, System.Data.SQLite"/>
    </DbProviderFactories>
  </system.data>
  // ...
```

## Setting Connection Strings

Next step is to replace connection strings for databases you want to use with Sqlite:

```
<connectionStrings>
  <add name="Default" connectionString=
    "Data Source=|DataDirectory|Serene_Default_v1.sqlite;"
    providerName="System.Data.Sqlite" />
  <add name="Northwind" connectionString=
    "Data Source=|DataDirectory|Serene_Northwind_v1.sqlite;"
    providerName="System.Data.Sqlite" />
</connectionStrings>
```

## Applying Sqlite Changes to Serene

Sqlite provider has been added recently, so if you already have an application, you'll need to get latest version of *SiteInitialization.Migrations.cs* from latest template / github repository to get Sqlite support.

Provider name must be `System.Data.Sqlite` for Serenity to auto-detect dialect. Read notes above to override default dialect.

I'm not sure why, but while FluentMigrator creates Northwind database for Sqlite first time, it takes some time.

## Configuring Code Generator

Sergen doesn't have reference to Sqlite provider, so if you want to use it to generate code, you must also register this provider with it.

Sergen.exe is an exe file, so you can't add a NuGet reference to it. We need to register this provider in application config file.

It is also possible to register the provider in GAC/machine.config and skip this step completely.

Locate Sergen.exe, which is under a folder like `packages/Serenity.CodeGenerator.1.8.6/tools` and create a file named `Sergen.exe.config` next to it with contents below:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.data>
    <DbProviderFactories>
      <remove invariant="System.Data.SQLite"/>
      <add name="SQLite Data Provider"
          invariant="System.Data.SQLite"
          description=".Net Framework Data Provider for SQLite"
          type="System.Data.SQLite.SQLiteFactory, System.Data.SQLite"/>
    </DbProviderFactories>
  </system.data>
  <appSettings>
    <add key="LoadProviderDLLs" value="Sqlite.Data.dll"/>
  </appSettings>
</configuration>
```

Also copy `System.Data.SQLite.dll` and its `x86` and `x64` folders under `bin` directory to same folder where `Sergen.exe` resides. Now Sergen will be able to generate code for your Sqlite tables.

# Oracle

Oracle support is available for Serene 2.2.2+

## Registering Oracle Provider

Oracle has a managed .NET provider named Oracle.ManagedDataAccess. You need to first install it in MyProject.Web:

```
Install-Package Oracle.ManagedDataAccess -Project MyProject.Web
```

If you didn't install this provider in GAC/machine.config before, or don't want to install it there, you need to register it in web.config file (Oracle.ManagedDataAccess NuGet package already does this on install):

```
<configuration>
  // ...
  <system.data>
    <DbProviderFactories>
      <remove invariant="Oracle.ManagedDataAccess.Client"/>
      <add name="ODP.NET, Managed Driver"
          invariant="Oracle.ManagedDataAccess.Client"
          description="Oracle Data Provider for .NET, Managed Driver"
          type="Oracle.ManagedDataAccess.Client.OracleClientFactory,
              Oracle.ManagedDataAccess, Version=4.121.2.0, Culture=neutral,
              PublicKeyToken=89b483f429c47342"/>
    </DbProviderFactories>
  </system.data>
  // ...
</configuration>
```

## Creating Databases

Serene can't autcreate database (tablespace) for Oracle. You might create them yourself, or use a script like below (i used this for XE):

```
CREATE TABLESPACE Serene_Default_v1_TABS
  DATAFILE 'Serene_Default_v1_TABS.dat' SIZE 10M AUTOEXTEND ON;
CREATE TEMPORARY TABLESPACE Serene_Default_v1_TEMP
  TEMPFILE 'Serene_Default_v1_TEMP.dat' SIZE 5M AUTOEXTEND ON;
CREATE USER Serene_Default_v1
  IDENTIFIED BY somepassword
  DEFAULT TABLESPACE Serene_Default_v1_TABS
  TEMPORARY TABLESPACE Serene_Default_v1_TEMP;
GRANT CREATE SESSION TO Serene_Default_v1;
GRANT CREATE TABLE TO Serene_Default_v1;
GRANT CREATE SEQUENCE TO Serene_Default_v1;
GRANT CREATE TRIGGER TO Serene_Default_v1;
GRANT UNLIMITED TABLESPACE TO Serene_Default_v1;

CREATE TABLESPACE Serene_Northwind_v1_TABS
  DATAFILE 'Serene_Northwind_v1_TABS.dat' SIZE 10M AUTOEXTEND ON;
CREATE TEMPORARY TABLESPACE Serene_Northwind_v1_TEMP
  TEMPFILE 'Serene_Northwind_v1_TEMP.dat' SIZE 5M AUTOEXTEND ON;
CREATE USER Serene_Northwind_v1
  IDENTIFIED BY somepassword
  DEFAULT TABLESPACE Serene_Northwind_v1_TABS
  TEMPORARY TABLESPACE Serene_Northwind_v1_TEMP;

GRANT CREATE SESSION TO Serene_Northwind_v1;
GRANT CREATE TABLE TO Serene_Northwind_v1;
GRANT CREATE SEQUENCE TO Serene_Northwind_v1;
GRANT CREATE TRIGGER TO Serene_Northwind_v1;
GRANT UNLIMITED TABLESPACE TO Serene_Northwind_v1;
```

## Setting Connection Strings

You might want to configure your data sources for ORACLE. I used Express Edition (XE) here:

```
<configuration>
  <oracle.manageddataaccess.client>
    <version number="*">
      <dataSources>
        <dataSource alias="XE"
          descriptor="
            (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
              (HOST=localhost)(PORT=1521))
              (CONNECT_DATA=(SERVICE_NAME=XE))) "/>
        </dataSource>
      </dataSources>
    </version>
  </oracle.manageddataaccess.client>
</configuration>
```

Next step is to replace connection strings for databases you want to use with Oracle:

Make sure you replace connection string parameters with values for your server

```
<connectionStrings>
  <add name="Default" connectionString="
    Data Source=XE;User Id=Serene_Default_v1;Password=somepassword;"
    providerName="Oracle.ManagedDataAccess.Client" />
  <add name="Northwind" connectionString="
    Data Source=XE;User Id=Serene_Northwind_v1;Password=somepassword;"
    providerName="Oracle.ManagedDataAccess.Client" />
</connectionStrings>
```

Provider name must be `Oracle.ManagedDataAccess.Client` for Serenity to auto-detect dialect. Read notes above to override default dialect.

## Configuring Code Generator

Sergen doesn't have support for Oracle yet, hopefully coming soon...



# Service Endpoints

In Serenity, Service Endpoints are a subclass of ASP.NET MVC controllers.

Here is an excerpt from Northwind OrderEndpoint:

```
namespace Serene.Northwind.Endpoints
{
    [RoutePrefix("Services/Northwind/Order"), Route("{action}")]
    [ConnectionFactory("Northwind"), ServiceAuthorize(Northwind.PermissionKeys.General)]
    public class OrderController : ServiceEndpoint
    {
        [HttpPost]
        public SaveResponse Create(IUnitOfWork uow, SaveRequest<MyRow> request)
        {
            return new MyRepository().Create(uow, request);
        }

        public ListResponse<MyRow> List(IDbConnection connection, ListRequest request)
        {
            return new MyRepository().List(connection, request);
        }
    }
}
```

## Controller Naming and Namespace

Our class has name *OrderController*, even though its file is named *OrderEndpoint.cs*. This is due to a ASP.NET MVC limitation (which i don't find so logical) that all controllers must end with *Controller* suffix.

If you don't end your controller class name with this suffix, your actions will simply won't work. So be very careful with this.

I also did this mistake several times and it cost me hours.

Namespace of this class (*Serene.Northwind.Endpoints*) is not important at all, though we usually put endpoints under *MyProject.Module.Endpoints* namespace for consistency.

Our OrderController derives from ServiceEndpoint (and should), which provides this MVC controller with not so usual features that we'll see shortly.

## Routing Attributes

```
[RoutePrefix("Services/Northwind/Order"), Route("{action}")]
```

Routing attributes above, which belongs to ASP.NET attribute routing, configures base address for our service endpoint. Our actions will be available under "mysite.com/Services/Northwind/Order".

Please avoid classic ASP.NET MVC routing, where you configured all routes in `ApplicationStart` method with `routes.AddRoute` etc. It was really hard to manage.

All Serenity service endpoints uses `/Services/Module/Entity` addressing scheme by default. Again even though you'd be able to use another address scheme, this is recommended for consistency and basic conventions.

## ConnectionKey Attribute

This attribute specifies which connection key in your application configuration file (e.g. `web.config`) should be used to create a connection when needed.

Let's see when and how this auto created connection is used:

```
public ListResponse<MyRow> List(IDbConnection connection, ListRequest request)
{
    return new MyRepository().List(connection, request);
}
```

Here we see that this action takes a `IDbConnection` parameter. You can't send a `IDbConnection` to an MVC action from client side. So who creates this connection?

Remember that our controller derives from `ServiceEndpoint`? So `ServiceEndpoint` understands that our action requires a connection. It checks `[ConnectionKey]` attribute on top of controller class to determine connection key, creates a connection using `SqlConnections.NewByKey()`, executes our action with this connection, and when action ends executing, closes the connection.

You'd be able to remove this connection parameter from the action and create it manually:

```
public ListResponse<MyRow> List(ListRequest request)
{
    using (var connection = SqlConnections.NewByKey("Northwind"))
    {
        return new MyRepository().List(connection, request);
    }
}
```



Actually this is what ServiceEndpoint does behind the scenes.

Why not use this feature while platform handles this detail automatically? One reason would be when you need to open a custom connection that is not listed in the config file, or open a dynamic one depending on some conditions.

We have another method that takes IUnitOfWork (transaction), instead of IDbConnection parameter:

```
public SaveResponse Create(IUnitOfWork uow, SaveRequest<MyRow> request)
{
    return new MyRepository().Create(uow, request);
}
```

Here situation is similar. ServiceEndpoint again creates a connection, but this time starts a transaction on it (IUnitOfWork), calls our action method, and when it returns will commit transaction automatically. Again, if it fails, would rollback it.

Here is the manual version of the same thing:

```
public SaveResponse Create(SaveRequest<MyRow> request)
{
    using (var connection = SqlConnections.NewByKey("Northwind"))
    using (var uow = new UnitOfWork(connection))
    {
        var result = new MyRepository().Create(uow, request);
        uow.Commit();
        return result;
    }
}
```

So, ServiceEndpoint handles something that takes 8 lines in 1 line of code.

## When To Use IUnitOfWork / IDbConnection

By convention, Serenity action methods that modify some state (CREATE, UPDATE etc.) should run inside a transaction, thus take an IUnitOfWork parameter, and ones that are read only operations (LIST, RETRIEVE) should use IDbConnection.

If your service method takes a IUnitOfWork parameter, this is a signature that your method might modify some data.

## About [HttpPost] Attribute

You may have noticed that Create, Update, Delete etc methods has this attribute while List, Retrieve etc. not.

This attribute limits Create, Update, Delete actions to HTTP POST only. It doesn't allow them to be called by HTTP GET.

This is because, these methods are ones that modify some state, e.g. insert, update, delete some records from DB, so they shouldn't be allowed to be called unintentionally, and their results shouldn't be allowed to be cached.

This also has some security implications. Actions with GET method might be subject to some attacks.

List, Retrieve doesn't modify anything, so they are allowed to be called with GET, e.g. typing in a browser address bar.

Even though, List, Retrieve can be called by GET, Serenity always calls services using HTTP POST when you use its methods, e.g. Q.CallService, and will turn of caching to avoid unexpected results.

## ServiceAuthorize Attribute

Our controller class has ServiceAuthorize attribute:

```
ServiceAuthorize(Northwind.PermissionKeys.General)
```

This attribute is similar to ASP.NET MVC [Authorize] attribute but it checks only that user is logged in, and throws an exception otherwise.

If used with no parameters, e.g. [ServiceAuthorize()] this attribute also checks that user is logged in.

When you pass it a permission key string, it will check that user is logged in and also has that permission.

```
ServiceAuthorize("SomePermission")
```

If user is not granted "SomePermission", this will prevent him from executing any endpoint method.

There is also [PageAuthorize] attribute that works similar, but you should prefer [ServiceAuthorize] with service endpoints, because its error handling is more suitable for services.

While [PageAuthorize] **redirects** user to the Login page, if user doesn't have the permission, ServiceAuthorize returns a more suitable **NotAuthorized service error**.

It's also possible to use [ServiceAuthorize] attribute on actions, instead of controller:

```
[ServiceAuthorize("SomePermissionThatIsRequiredForCreate")]  
public SaveResponse Create(SaveRequest<MyRow> request)
```

## About Request and Response Objects

Except the specially handled IUnitOfWork and IDbConnection parameters, all Serenity service actions takes a single request parameter and returns a single result.

```
public SaveResponse Create(IUnitOfWork uow, SaveRequest<MyRow> request)
```

Let's start with the result. If you have some background on ASP.NET MVC, you'd know that controllers can't return arbitrary objects. They must return objects that derive from *ActionResult*.

But our *SaveResponse* derives from *ServiceResponse* which is just an ordinary object:

```
public class SaveResponse : ServiceResponse  
{  
    public object EntityId;  
}  
  
public class ServiceResponse  
{  
    public ServiceError Error { get; set; }  
}
```

How this is possible? Again ServiceEndpoint handles this detail behind the scenes. It transforms our SaveResponse to a special action result that returns JSON data.

We don't have to worry about this detail as long as our response object derives from ServiceResponse and is JSON serializable.

Again, our request object is also an ordinary class that derives from a basic ServiceRequest:

```
public class SaveRequest<TEntity> : ServiceRequest, ISaveRequest
{
    public object EntityId { get; set; }
    public TEntity Entity { get; set; }
}

public class ServiceRequest
{
}
```

ServiceEndpoint takes the HTTP request content which is usually JSON, deserializes it into our *request* parameter, using a special MVC action filter (JsonFilter).

If you want to use some custom actions, your methods should also follow this philosophy, e.g. take just one request (deriving from ServiceRequest) and return one response (deriving from ServiceResponse).

Let's add a service method that returns count of all orders greater than some amount:

```
public class MyOrderCountRequest : ServiceRequest
{
    public decimal MinAmount { get; set; }
}

public class MyOrderCountResponse : ServiceResponse
{
    public int Count { get; set; }
}

public class OrderController : ServiceEndpoint
{
    public MyOrderCountResponse MyOrderCount(IDbConnection connection,
        MyOrderCountRequest request)
    {
        var fld = OrderRow.Fields;
        return new MyOrderCountResponse
        {
            Count = connection.Count<OrderRow>(fld.TotalAmount >= request.MinAmount);
        };
    }
}
```

Please follow this pattern and try not to add more parameters to action methods. Serenity follows message based pattern, with only one request object, that can be extended later by adding more properties.

Don't do this (which is called RPC - Remote procedure call style):

```
public class OrderController : ServiceEndpoint
{
    public decimal MyOrderCount(IDbConnection connection,
        decimal minAmount, decimal maxAmount, ....)
    {
        // ...
    }
}
```

Prefer this (message based services):

```
public class MyOrderCountRequest : ServiceRequest
{
    public decimal MinAmount { get; set; }
    public decimal MaxAmount { get; set; }
}

public class OrderController : ServiceEndpoint
{
    public MyOrderCountResponse MyOrderCount(IDbConnection connection,
        MyOrderCountRequest request)
    {
        // ...
    }
}
```

This will avoid having to remember parameter order, will make your request objects extensible without breaking backwards compability, and have many more advantages that you may notice later.

## Why Endpoint Methods Are Almost Empty

We usually delegate actual work to our repository layer:

```
public ListResponse<MyRow> List(IDbConnection connection, ListRequest request)
{
    return new MyRepository().List(connection, request);
}
```

Remember that ServiceEndpoint has a direct dependency to ASP.NET MVC. This means that any code you write inside a service endpoint will have a dependency to ASP.NET MVC, and thus web environment.

You may not be able to reuse any code you wrote here, from let's say a desktop application, or won't be able to isolate this code into a DLL that doesn't have a reference to WEB libraries.

But if you really don't have such a requirement, you can remove repositories all together and write all your code inside the endpoint.

Some people might argue that entities, repositories, business rules, endpoints etc. should all be in their own isolated assemblies. In theory, and for some scenarios this might be valid, but some (or most) users don't need so much isolation, and may fall into YAGNI (you aren't gonna need it) category.

# ListRequestHandler

This is the base class that handles List requests originating from client side, e.g. from grids.

Let's first sample when and how this class handles list requests:

1. First a list request must be triggered from client side. Possible options are:
  - a) You open a list page that contains a grid. Right after your grid object is created it builds up a ListRequest object, based on currently visible columns, initial sort order, filters etc. and submits it to server side.
  - b) User clicks on a column header to sort, clicks paging buttons or refresh button to trigger same events in option A.
  - c) You might manually call a list service using XYZService.List method.
2. A service request (AJAX) to MVC XYZController (in file XYZEndpoint.cs) arrives at server. Request parameters are deserialized from JSON into a ListRequest object.
3. XYZEndpoint calls XYZRepository.List method with retrieved ListRequest object.
4. XYZRepository.List method creates a subclass of ListRequestHandler (XYZRepository.MyListHandler) and invokes its Process method with the ListRequest.
5. ListRequestHandler.Process method builds up a dynamic SQL query, based on the ListRequest, metadata in its entity type (Row) and other information and executes it.
6. ListRequestHandler.Process returns a ListResponse with Entities member that contains rows to be returned.
7. XYZEndpoint receives this ListResponse, returns it from action.
8. ListResponse is serialized to JSON, sent back to client
9. Grid receives entities, updates its displayed rows and other parts like paging status.

We'll cover how grids build and submit a list request in another chapter. Let's focus on ListRequestHandler for now.

## List Request Object

First we should have a look at what members a ListRequest object have:

```
public class ListRequest : ServiceRequest, IIncludeExcludeColumns
{
    public int Skip { get; set; }
    public int Take { get; set; }
    public SortBy[] Sort { get; set; }
    public string ContainsText { get; set; }
    public string ContainsField { get; set; }
    public Dictionary<string, object> EqualityFilter { get; set; }
    [JsonConverter(typeof(JsonSafeCriteriaConverter))]
    public BaseCriteria Criteria { get; set; }
    public bool IncludeDeleted { get; set; }
    public bool ExcludeTotalCount { get; set; }
    public ColumnSelection ColumnSelection { get; set; }
    [JsonConverter(typeof(JsonStringHashSetConverter))]
    public HashSet<string> IncludeColumns { get; set; }
    [JsonConverter(typeof(JsonStringHashSetConverter))]
    public HashSet<string> ExcludeColumns { get; set; }
}
```

## ListRequest.Skip and ListRequest.Take Parameters

These options are used for paging and similar to Skip and Page extensions in LINQ.

There is one little difference about Take. If you Take(0), LINQ will return you zero records, while Serenity will return ALL records. There is no point in calling a LIST service and requesting 0 records.

So, SKIP and TAKE has default values of 0, and they are simply ignored when 0 / undefined.

```
// returns all customers as Skip and Take are 0 by default
CustomerService.List(new ListRequest
{
}, response => {});
```

If you have a grid that has page size 50 and switch to page number 4, SKIP will be 200 while TAKE is 50.

```
// returns customers between row numbers 201 and 250 (in some default order)
CustomerService.List(new ListRequest
{
    Skip = 200,
    Take = 50
}, response => {});
```

These parameters are converted to relevant SQL paging statements based on SQL dialect.



## ListRequest.Sort Parameter

This parameter takes an array to sort results on. Sorting is performed by generating SQL.

SortBy parameter expects a list of *SortBy* objects:

```
[JsonConverter(typeof(JsonSortByConverter))]
public class SortBy
{
    public SortBy()
    {
    }

    public SortBy(string field)
    {
        Field = field;
    }

    public SortBy(string field, bool descending)
    {
        Field = field;
        Descending = descending;
    }

    public string Field { get; set; }
    public bool Descending { get; set; }
}
```

When calling a List method of XYZRepository server side to sort by Country then City descending, you might do it like this:

```
new CustomerRepository().List(connection, new ListRequest
{
    SortBy = new[] {
        new SortBy("Country"),
        new SortBy("City", descending: true)
    }
});
```

SortBy class has a custom JsonConverter so when building a list request client side, you should use a simple string array:

```
// CustomerEndpoint and thus CustomerRepository is accessed from
// client side (YourProject.Script) through CustomerService class static methods
// which is generated by ServiceContracts.tt
CustomerService.List(connection, new ListRequest
{
    SortBy = new[] { "Country", "City DESC" }
}, response => {});
```

This is because ListRequest class definition at client side has a bit different structure:

```
[Imported, Serializable, PreserveMemberCase]
public class ListRequest : ServiceRequest
{
    public int Skip { get; set; }
    public int Take { get; set; }
    public string[] Sort { get; set; }
    // ...
}
```

Column names used here should be Property names of corresponding fields. Expressions are not accepted. E.g. this won't work!:

```
CustomerService.List(connection, new ListRequest
{
    SortBy = new[] { "t0.FirstName + ' ' + t0.LastName" }
}, response => {});
```

## ListRequest.ContainsText and ListRequest.ContainsField Parameters

These parameters are used by quick search functionality which is search input on top left of grids.

When only ContainsText is specified and ContainsField is empty, searching is performed on all fields with [QuickSearch] attribute on them.

It is possible to define some specific field list to perform searches on grid client side, by overriding GetQuickSearchField() methods. So when such a field is selected in quick search input, search is only performed on that column.

If you set ContainsField to a field name that doesn't have QuickSearch attribute on it, system will raise an exception. This is for security purposes.

As usual, searching is done with dynamic SQL by LIKE statements.

```
CustomerService.List(connection, new ListRequest
{
    ContainsText = "the",
    ContainsField = "CompanyName"
}, response => {});
```

```
SELECT ... FROM Customers t0 WHERE t0.CompanyName LIKE '%the%'
```

If ContainsText is null or empty string it is simply ignored.

## ListRequest.EqualityFilter Parameter

EqualityFilter is a dictionary that allows quick equality filtering by some fields. It is used by quick filter dropdowns on grids (ones that are defined with AddEqualityFilter helper).

```
CustomerService.List(connection, new ListRequest
{
    EqualityFilter = new JsDictionary<string, object> {
        { "Country", "Germany" }
    }
}, response => {});
```

```
SELECT * FROM Customers t0 WHERE t0.Country = "Germany"
```

Again, you should use property names as equality field keys, not expressions. Serenity doesn't allow any arbitrary SQL expressions from client side, to prevent SQL injections.

Please note that null and empty string values are simply ignored, similar to ContainsText, so it's not possible to filter for empty or null values with EqualityFilter. Such a request would return all records:

```
CustomerService.List(connection, new ListRequest
{
    EqualityFilter = new JsDictionary<string, object> {
        { "Country", "" }, // won't work, empty string is ignored
        { "City", null }, // won't work, null is ignored
    }
}, response => {});
```

Use Criteria parameter if you intent to filter customers with empty countries.

## ListRequest.Criteria

This parameter accepts criteria objects similar to server side Criteria objects we talked about in Fluent SQL chapter. Only difference is, as these criteria objects are sent from client side, they have to be validated and can't contain any arbitrary SQL expressions.

Service request below will only return customers with empty country or null city values

```
CustomerService.List(connection, new ListRequest
{
    Criteria = new Criteria("Country") == "" |
        new Criteria("City").IsNull()
}, response => {});
```

You could set Criteria parameter of generated ListRequest that is about to be submitted in your XYZGrid.cs like below:

```
protected override bool OnViewSubmit()
{
    // only continue if base class didn't cancel request
    if (!base.OnViewSubmit())
        return false;

    // view object is the data source for grid (SlickRemoteView)
    // this is an EntityGrid so view.Params is a ListRequest
    var request = (ListRequest)view.Params;

    // we use " &= " here because otherwise we might clear
    // filter set by an edit filter dialog if any.

    request.Criteria &=
        new Criteria(ProductRow.Fields.UnitsInStock) > 10 &
        new Criteria(ProductRow.Fields.CategoryName) != "Condiments" &
        new Criteria(ProductRow.Fields.Discontinued) == 0;

    return true;
}
```

You could also set other parameters of ListRequest in your grids similarly.

## ListRequest.IncludeDeleted

This parameter is only useful with rows that implements IIsActiveDeletedRow interface. If row has such an interface, list handler by default only returns rows that are not deleted (IsActive != -1). It is a way to not delete rows actually but mark them as deleted.

If this parameter is True, list handler will return all rows without looking at IsActive column.

Some grids for such rows have a little eraser icon on top right to toggle this flag, thus show deleted records or hide them (default).

## ListRequest.ColumnSelection Parameter

Serenity tries hard to load only required columns of your entities from SQL server to limit network traffic to minimum between SQL Server < - > WEB Server and thus keep data size transferred to client as low as possible.

ListRequest has a ColumnSelection parameter for you to control the set of columns loaded from SQL.

ColumnSelection enumeration has following values defined:

```
public enum ColumnSelection
{
    List = 0,
    KeyOnly = 1,
    Details = 2,
}
```

By default grid requests records from List service in "ColumnSelection.List" mode (can be changed). Thus, its list request looks like this:

```
new ListRequest
{
    ColumnSelection = ColumnSelection.List
}
```

In *ColumnSelection.List* mode, ListRequestHandler returns *table* fields, thus fields that actually belong to the table, not view fields that are originating from joined tables.

One exception is *expression* fields that only contains reference to *table* fields, e.g. (*t0.FirstName + ' ' + t0.LastName*). ListRequestHandler also loads such fields.

*ColumnSelection.KeyOnly* only includes ID / primary key fields.

*ColumnSelection.Details* includes all fields, including view ones, unless a field is explicitly excluded or marked as "sensitive", e.g. a password field.

Dialogs loads edited records in Details mode, thus they also include view fields.

## ListRequest.IncludeColumns Parameter

We told that grid requests records in *List* mode, so loads only *table* fields, then how it can show columns that originate from other tables?

Grid sends list of visible columns to List service with *IncludeColumns*, so these columns are *included* in selection even if they are view fields.

In memory grids can't do this. As they don't call services directly, you have to put `[MinSelectLevel(SelectLevel.List)]` to view fields that you want to load for in memory detail grids.

If you have a ProductGrid that shows SupplierName column its actual ListRequest looks like this:

```
new ListRequest
{
    ColumnSelection = ColumnSelection.List,
    IncludeColumns = new List<string> {
        "ProductID",
        "ProductName",
        "SupplierName",
        "..."
    }
}
```

Thus, these extra view fields are also included in *selection*.

If you have a grid that should only load visible columns for performance reasons, override its ColumnSelection level to *KeyOnly*. Note that non-visible table fields won't be available in client side row.

## ListRequest.ExcludeColumns Parameter

Opposite of IncludeColumns is ExcludeColumns. Let's say you have a `nvarchar(max)` *Notes* field on your row that is never shown in the grid. To lower network traffic, you may choose to NOT load this field in product grid:

```
new ListRequest
{
    ColumnSelection = ColumnSelection.List,
    IncludeColumns = new List<string> {
        "ProductID",
        "ProductName",
        "SupplierName",
        "..."
    },
    ExcludeColumns = new List<string> {
        "Notes"
    }
}
```

OnViewSubmit is a good place to set this parameter (and some others):

```
protected override bool OnViewSubmit()
{
    if (!base.OnViewSubmit())
        return false;

    var request = (ListRequest)view.Params;
    request.ExcludeColumns = new List<string> { "Notes" };
    return true;
}
```

## Controlling Loading At Server Side

You might want to exclude some fields like *Notes* from *ColumnSelection.List*, without excluding it explicitly in grid. This is possible with *MinSelectLevel* attribute:

```
[MinSelectLevel(SelectLevel.Details)]
public String Note
{
    get { return Fields.Note[this]; }
    set { Fields.Note[this] = value; }
}
```

There is a *SelectLevel* enumeration that controls when a field is loaded for different *ColumnSelection* levels:

```
public enum SelectLevel
{
    Default = 0,
    Always = 1,
    Lookup = 2,
    List = 3,
    Details = 4,
    Explicit = 5,
    Never = 6
}
```

*SelectLevel.Default*, which is the default value, corresponds to *SelectLevel.List* for table fields and *SelectLevel.Details* for view fields.

By default, table fields have a select level of *SelectLevel.List* while view fields have *SelectLevel.Details*.

*SelectLevel.Always* means such a field is selected for any column selection mode, **even if** it is explicitly excluded using *ExcludeColumns*.

*SelectLevel.Lookup* is obsolete, avoid using it. Lookup columns are determined with [LookupInclude] attribute.

*SelectLevel.List* means such a field is selected for ColumnSelection.List and ColumnSelection.Details modes or if it is explicitly included with IncludeColumns parameter.

*SelectLevel.Details* means such a field is selected for ColumnSelection.Details mode, or if it is explicitly included with IncludeColumns parameter.

*SelectLevel.Explicit* means such a field shouldn't be selected in any mode, unless it is explicitly included with IncludeColumns parameter. Use this for fields that are not meaningful for grids or edit dialogs.

*SelectLevel.Never* means never load this field! Use it for fields that shouldn't be sent to client side, like a password hash.



# Widgets

Serenity Script UI layer's component classes (control) are based on a system that is similar to *jQuery UI's Widget Factory*, but redesigned for C#.

You can find more information about jQuery UI widget system here:

<http://learn.jquery.com/jquery-ui/widget-factory/>

<http://msdn.microsoft.com/en-us/library/hh404085.aspx>

Widget, is an object that is attached to an HTML element and extends it with some behaviour.

For example, IntegerEditor widget, when attached to an INPUT element, makes it easier to enter numbers in the input and validates that the entered number is a correct integer.

Similarly, a Toolbar widget, when attached to a DIV element, turns it into a toolbar with tool buttons (in this case, DIV acts as a placeholder).

# ScriptContext Class

C#, doesn't support global methods, so jQuery's `$` function can't be used as simply in Saltarelle as it is in Javascript.

A simple expression like `$('#SomeElementId')` in Javascript corresponds to Saltarelle C# code `jQuery.Select("#SomeElementId")` .

As a workaround, *ScriptContext* class can be used:

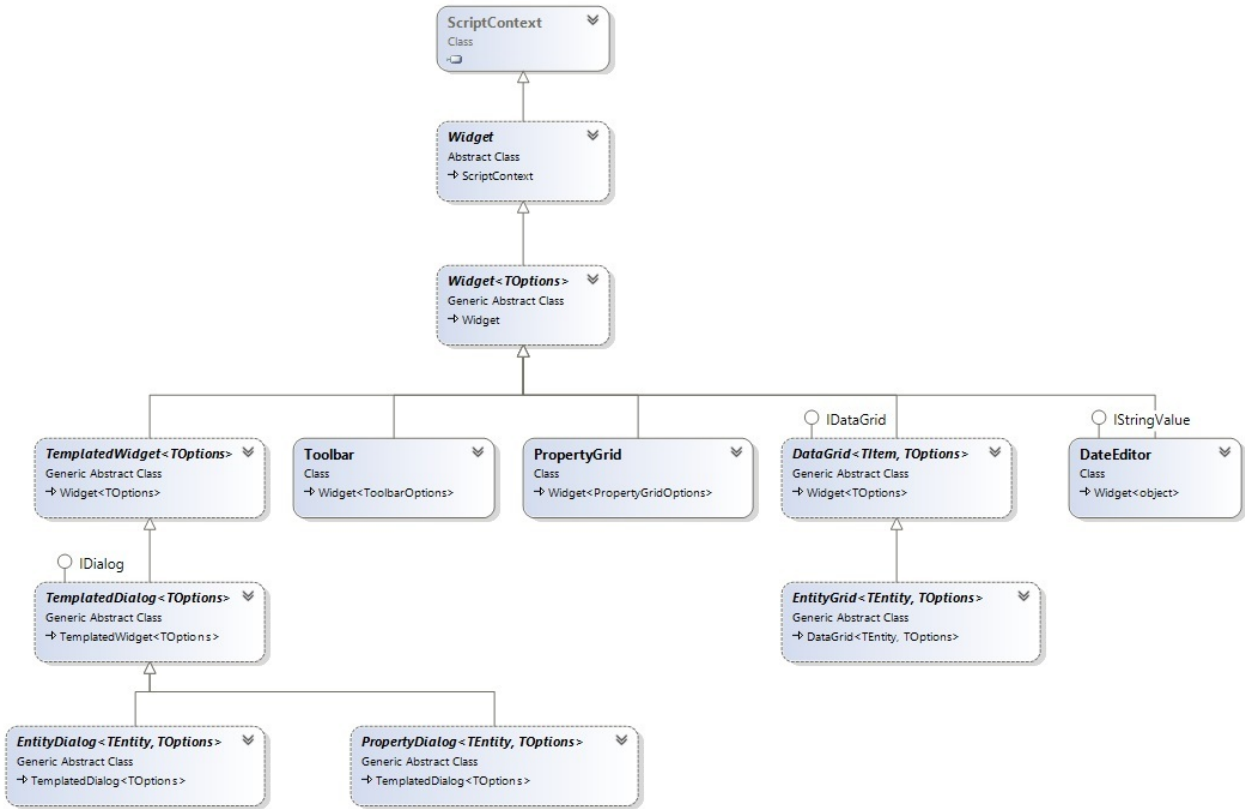
```
public class ScriptContext
{
    [InlineCode("${p}")]
    protected static jQueryObject J(object p);
    [InlineCode("${p}, {context}")]
    protected static jQueryObject J(object p, object context);
}
```

As `$` is not a valid method name in C#, `J` is chosen instead. In subclasses of *ScriptContext*, `jQuery.Select()` function can be called briefly as `J()` .

```
public class SampleClass : ScriptContext
{
    public void SomeMethod()
    {
        J("#SomeElementId").AddClass("abc");
    }
}
```

# Widget Class

## Widget Class Diagram



## A sample Widget

Let's build a widget, that increases a DIV's font size everytime it is clicked:

```
namespace MySamples
{
    public class MyCoolWidget : Widget
    {
        private int fontSize = 10;

        public MyCoolWidget(jQueryObject div)
            : base(div)
        {
            div.Click(e => {
                fontSize++;
                this.Element.Css("font-size", fontSize + "pt");
            });
        }
    }
}
```

```
<div id="SomeDiv">Sample Text</div>
```

We can create this widget on an HTML element, like:

```
var div = jQuery.Select("#SomeDiv");
new MyCoolWidget(div);
```

## Widget Class Members

```
public abstract class Widget : ScriptContext
{
    private static int NextWidgetNumber = 0;

    protected Widget(jQueryObject element);
    public virtual void Destroy();

    protected virtual void OnInit();
    protected virtual void AddCssClass();

    public jQueryObject Element { get; }
    public string WidgetName { get; }
    public string UniqueName { get; }
}
```

### ***Widget.Element*** Property

Classes derived from Widget can get the element, on which they are created, by the `Element` property.

```
public jQueryObject Element { get; }
```

This property has type of `jQueryObject` and returns the element, which is used when the widget is created. In our sample, container DIV element is referenced as `this.Element` in the click handler.

## HTML Element and Widget CSS Class

When a widget is created on an HTML element, it does some modifications to the element.

First, the HTML element gets a CSS class, based on the type of the widget.

In our sample, `.s-MyCoolWidget` class is added to the `DIV` with ID `#SomeDiv`.

Thus, after widget creation, the DIV looks similar to this:

```
<div id="SomeDiv" class="s-MyCoolWidget">Sample Text</div>
```

This CSS class is generated by putting a `s-` prefix in front of the widget class name (it can be changed by overriding `Widget.AddCssClass` method).

## Styling the HTML Element With Widget CSS Class

Widget CSS class can be used to style the HTML element that the widget is created on.

```
.s-MyCoolWidget {  
    background-color: red;  
}
```

## Getting a Widget Reference From an HTML Element with the *jQuery.Data* Function

Along with adding a CSS class, another information about the widget is added to the HTML element, though it is not obvious on markup. This information can be seen by typing following in Chrome console:

```
> $('#SomeDiv').data()  
  
> Object { MySamples_MyCoolWidget: $MySamples_MyCoolWidget }
```

Thus, it is possible to get a reference to a widget that is attached to an HTML element, using `$.data` function. In C# this can be written as:

```
var myWidget = (MyCoolWidget)(J("#SomeDiv").GetDataValue('MySamples_MyCoolWidget'));
```

## WidgetExtensions.GetWidget Extension Method

Instead of the prior line that looks a bit long and complex, a Serenity shortcut can be used:

```
var myWidget = J("#SomeDiv").GetWidget<MyCoolWidget>();
```

This piece of code returns the widget if it exists on HTML element, otherwise throws an exception:

```
Element has no widget of type 'MySamples_MyCoolWidget'!
```

## WidgetExtensions.TryGetWidget Extension Method

TryGetWidget can be used to check if the widget exists, simply returning `null` if it doesn't:

```
var myWidget = $("#SomeDiv").TryGetWidget<MyCoolWidget>();
```

## Creating Multiple Widgets on an HTML Element

Only one widget of the same class can be attached to an HTML element.

An attempt to create a secondary widget of the same class on a element throws the following error:

```
The element already has widget 'MySamples_MyCoolWidget'.
```

Any number of widgets from different classes can be attached to a single element as long as their behaviour doesn't affect each other.

## Widget.UniqueName Property

Every widget instance gets a unique name like `MySamples_MyCoolWidget3` automatically, which can be accessed by `this.UniqueName` property.

This unique name is useful as a ID prefix for the HTML element and its descendant elements which may be generated by the widget itself.

It can also be used as an event class for `$.bind` and `$.unbind` methods to attach / detach event handlers without affecting other handlers, which might be attached to the element:

```
jQuery("body").Bind("click." + this.UniqueName, delegate { ... });  
...  
jQuery("body").Unbind("click." + this.UniqueName);
```

## Widget.Destroy Method

Sometimes releasing an attached widget might be required without removing the HTML element itself.

Widget class provides `Destroy` method for the purpose.

In default implementation of the Destroy method, event handlers which are assigned by the widget itself are cleaned (by using UniqueName event class) and its CSS class ( `.s-WidgetClass` ) is removed from the HTML element.

Custom widget classes might need to override Destroy method to undo changes on HTML element and release resources (though, no need to detach handlers that are attached previously with UniqueName class)

Destroy method is called automatically when the HTML element is detached from the DOM. It can also be called manually.

If destroy operation is not performed correctly, memory leaks may occur in some browsers.

## Widget < TOptions > Generic Class

If a widget requires some additional initialization options, it might be derived from the

```
Widget< TOptions > class.
```

The options passed to the constructor can be accessed in class methods through the protected field `options` .

```
public abstract class Widget< TOptions > : Widget
    where TOptions: class, new()
{
    protected Widget(jQueryObject element, TOptions opt = null) { ... }
    protected readonly TOptions options;
}
```



# TemplatedWidget Class

A widget that generates a complicated HTML markup in its constructor or other methods might lead to a class with much spaghetti code that is hard to maintain. Besides, as markup lies in program code, it might be difficult to customize output.

```
public class MyComplexWidget : Widget
{
    public MyComplexWidget(jQueryObject div)
        : base(div)
    {
        var toolbar = J("<div>")
            .Attribute("id", this.UniqueName + "_MyToolbar")
            .AppendTo(div);

        var table = J("<table>")
            .AddClass("myTable")
            .Attribute("id", this.UniqueName + "_MyTable")
            .AppendTo(div);

        var header = J("<thead/>").AppendTo(table);
        var body = J("<tbody/>").AppendTo(table);
        ...
        ...
        ...
    }
}
```

Such problems can be avoided by using HTML templates. For example, lets add the following template into the HTML page:

```
<script id="Template_MyComplexWidget" type="text/html">
<div id="~_MyToolbar">
</div>
<table id="~_MyTable">
    <thead><tr><th>Name</th><th>Surname</th>...</tr></thead>
    <tbody>...</tbody>
</table>
</script>
```

Here, a `SCRIPT` tag is used, but by specifying its type as `"text/html"`, browser won't recognize it as a real script to execute.

By making use of `TemplatedWidget`, lets rewrite previous spaghetti code block:

```
public class MyComplexWidget : TemplatedWidget
{
    public MyComplexWidget(jQueryObject div)
        : base(div)
    {
    }
}
```

When this widget is created on an HTML element like following:

```
<div id="SampleElement">
</div>
```

You'll end up with such an HTML markup:

```
<div id="SampleElement">
    <div id="MySamples_MyComplexWidget1_MyToolbar">
    </div>
    <table id="MySamples_MyComplexWidget1_MyTable">
        <thead><tr><th>Name</th><th>Surname</th>...</tr></thead>
        <tbody>...</tbody>
    </table>
</div>
```

TemplatedWidget automatically locates the template for your class and applies it to the HTML element.

## TemplatedWidget ID Generation

If you watch carefully, in our template we specified ID for descendant elements as

```
~_MyToolbar and ~_MyTable .
```

But when this template is applied to the HTML element, resulting markup contained ID's of **MySamples\_MyComplexWidget1\_MyToolbar** and **MySamples\_MyComplexWidget1\_MyTable** instead.

TemplatedWidget replaces prefixes like `~_` with the widget's `UniqueName` and underscore ("`_`") ( `this.idPrefix` contains the combined prefix).

Using this strategy, even if the same widget template is used in a page for more than one HTML element, their ID's won't conflict with each other as they will have unique ID's.

## TemplatedWidget.ByID Method

As TemplateWidget appends a unique name to them, the ID attributes in a widget template can't be used to access elements after widget creation.

Widget's unique name and an underscore should be prepended to the original ID attribute in the template to find an element:

```
public class MyComplexWidget : TemplatedWidget
{
    public MyComplexWidget(jQueryObject div)
        : base(div)
    {
        J(this.uniqueName + "_" + "Toolbar").AddClass("some-class");
    }
}
```

TemplatedWidget's ByID method can be used instead:

```
public class MyComplexWidget
{
    public MyComplexWidget(jQueryObject div)
        : base(div)
    {
        ByID("Toolbar").AddClass("some-class");
    }
}
```

## TemplatedWidget.GetTemplateName Method

In the last sample `MyComplexWidget` located its template automatically.

TemplatedWidget makes use of a convention to find its template (convention based programming). It inserts `Template_` prefix before the class name and searches for a `SCRIPT` element with this ID attribute ( `Template_MyComplexWidget` ) and uses its HTML content as a template.

If we wanted to use another ID like following:

```
<script id="TheMyComplexWidgetTemplate" type="text/html">
    ...
</script>
```

An error like this would be seen in the browser console:

```
Can't locate template for widget 'MyComplexWidget' with name 'Template_MyComplexWidget'!
```

We might fix our template ID or ask the widget to use our custom ID:

```
public class MyComplexWidget
{
    protected override string GetTemplateName()
    {
        return "TheMyComplexWidgetTemplate";
    }
}
```

## TemplatedWidget.GetTemplate Method

`GetTemplate` method might be overridden to provide a template from another source or specify it manually:

```
public class MyCompleWidget
{
    protected override string GetTemplate()
    {
        return $("#TheMyComplexWidgetTemplate").GetHtml();
    }
}
```

## Q.GetTemplate Method and Server Side Templates

Default implementation for `TemplatedWidget.GetTemplate` method calls `GetTemplateName` and searches for a `SCRIPT` element with that ID.

If no such `SCRIPT` element is found, `Q.GetTemplate` is called with the same ID.

An error is thrown if neither returns a result.

`Q.GetTemplate` method provides access to templates defined on the server side. These templates are compiled from files with `.template.cshtml` extension in `~/Views/Template` or `~/Modules` folders or their subfolders.

For example, we could create a template for `MyComplexWidget` in a server side file like `~/Views/Template/SomeFolder/MyComplexWidget.template.cshtml` with the following content:

```
<div id="~_MyToolbar">
</div>
<table id="~_MyTable">
  <thead><tr><th>Name</th><th>Surname</th>...</tr></thead>
  <tbody>...</tbody>
</table>
```

Template file name and extension is important while its folder is simply ignored.

By using this strategy there would be no need to insert widget templates into the page markup.

Also, as such server side templates are loaded on the first use (*lazy loading*) and cached in the browser and the server, page markup doesn't get polluted with templates for widgets that we might never use in a specific page. Thus, server side templates are favored over inline SCRIPT templates.

## TemplatedDialog Class

TemplatedWidget's subclass TemplatedDialog makes use of jQuery UI Dialog to create in-page modal dialogs.

Unlike other widget types TemplatedDialog creates its own HTML element, which it will be attached to.

# Attributes

## Visible Attribute

**namespace:** *Serenity.ComponentModel*, **assembly:** *Serenity.Core*

Controls visibility of a column or form field.

It is also possible to hide a field by passing *false* as its value, but `[Hidden]` attribute is recommended.

```
public class SomeColumns
{
    [Visible]
    public string ExplicitlyVisible { get; set; }
    [Visible(false)]
    public string ExplicitlyHidden { get; set; }
}
```

- User might still show the column by using the column picker if any.

## Hidden Attribute

**namespace:** *Serenity.ComponentModel*, **assembly:** *Serenity.Core*

Hides a column or form field.

This is just a subclass of *VisibleAttribute* with *false* value.

```
public class SomeColumns
{
    [Hidden]
    public string HiddenColumn { get; set; }
}
```

- User might still show the column by using the column picker if any.

## HideOnInsert Attribute

**namespace:** *Serenity.ComponentModel*, **assembly:** *Serenity.Core*

Controls whether a field is visible on new record mode.

- This only works with forms, not columns.

```
public class SomeColumns
{
    [HideOnInsert]
    public string HideMeOnInsert { get; set; }
    [HideOnInsert(false)]
    public string DontHideMeOnInsert { get; set; }
}
```

## HideOnUpdate Attribute

**namespace:** *Serenity.ComponentModel*, **assembly:** *Serenity.Core*

Controls whether a field is visible on edit record mode.

- This only works with forms, not columns.

```
public class SomeColumns
{
    [HideOnUpdate]
    public string HideMeOnUpdate { get; set; }
    [HideOnUpdate(false)]
    public string DontHideMeOnUpdate { get; set; }
}
```

## Insertable Attribute

**namespace:** *Serenity.ComponentModel*, **assembly:** *Serenity.Core*

Controls if a property is editable in new record mode.

- When used on row fields, turns on or off the Insertable flag.
- It has no effect on columns

```
public class SomeForm
{
    [Insertable(false)]
    public string ReadOnlyOnInsert { get; set; }
}
```



## Updatable Attribute

**namespace:** *Serenity.ComponentModel*, **assembly:** *Serenity.Core*

Controls if a property is editable in edit record mode.

- When used on row fields, turns on or off the Updatable flag.
- It has no effect on columns

```
public class SomeForm
{
    [Updatable(false)]
    public string ReadOnlyOnUpdate { get; set; }
}
```

## DisplayName Attribute

**namespace:** *System.ComponentModel*, **assembly:** *System*

Determines default title for grid columns or form fields.

```
public class SomeForm
{
    [DisplayName("Title for Some Field")]
    public string SomeField { get; set; }
}
```

- DisplayName attribute cannot be used on Enum members, so you have to use Description attribute
- Titles set with this attribute is considered to be in *invariant* language.

This is not a Serenity attribute, it resides in .NET System assembly.

## Description Attribute

**namespace:** *System.ComponentModel*, **assembly:** *System*

Determines default title for enum members.

```
public class SomeEnum
{
    [Description("Title for Value 1")]
    Value1 = 1,
    [Description("Value 2")]
    Value2 = 2
}
```

- Titles set with this attribute is considered to be in *invariant* language.

This is not a Serenity attribute, it resides in .NET System assembly.

## DisplayFormat Attribute

**namespace:** *Serenity.ComponentModel*, **assembly:** *Serenity.Core*

Sets the display format for a column.

- This has no effect on editors! It is only for Display, **NOT Editing**. For editing, you have to change culture in web.config (not UI culture).
- Display format strings are specific to column data and formatter type.
- If column is a Date or DateTime column, its default formatter accepts custom DateTime format strings like *dd/MM/yyyy*.
- We don't suggest setting DisplayFormat for dates explicitly, use culture setting (not UI culture) in *web.config* unless a column has to display date/time in a different order than the default.
- You may also use following standard format strings:
  - **"d"**: *dd/MM/yyyy* where DMY order changes based on current culture.
  - **"g"**: *dd/MM/yyyy HH:mm* where DMY order changes based on current culture.
  - **"G"**: *dd/MM/yyyy HH:mm:ss* where DMY order changes based on current culture.
  - **"s"**: *yyydd-MM-ddTHH:mm:ss* ISO sortable date time format.
  - **"u"**: *yyydd-MM-ddTHH:mm:ss.fffZ* ISO 8601 UTC.
- If column is an integer, double or decimal it accepts .NET custom numeric format strings.

```
public class SomeColumns
{
    [DisplayFormat("d")]
    public DateTime DateWithCultureDMYOrder { get; set; }
    [DisplayFormat("dd/MM/yyyy")]
    public DateTime DateWithConstantDMYOrder { get; set; }
    [DisplayFormat("g")]
    public DateTime DateTimeToMinWithCultureDMYOrder { get; set; }
    [DisplayFormat("dd/MM/yyyy HH:mm")]
    public DateTime DateTimeToMinConstantDMYOrder { get; set; }
    [DisplayFormat("G")]
    public DateTime DateTimeToSecWithCultureDMYOrder { get; set; }
    [DisplayFormat("dd/MM/yyyy HH:mm:ss")]
    public DateTime DateTimeToSecWithConstantDMYOrder { get; set; }
    [DisplayFormat("s")]
    public DateTime SortableDateTime { get; set; }
    [DisplayFormat("u")]
    public DateTime ISO8601UTC { get; set; }
    [DisplayFormat("#,##0.00")]
    public Decimal ShowTwoZerosAfterDecimalWithGrouping { get; set; }
    [DisplayFormat("0.00")]
    public Decimal ShowTwoZerosAfterDecimalNoGrouping { get; set; }
}
```

## Placeholder Attribute

**namespace:** *Serenity.ComponentModel*, **assembly:** *Serenity.Core*

Sets a placeholder for a form field.

- Placeholder is shown inside the editor with gray color when editor value is empty.
- Only basic input based editors and Select2 supports this. It is ignored by other editor types like Checkbox, Grid, FileUploadEditor etc.

```
public class SomeForm
{
    [Placeholder("Show this inside the editor when it is empty")]
    public string FieldWithPlaceHolder { get; set; }
}
```

## Hint Attribute

**namespace:** *Serenity.ComponentModel*, **assembly:** *Serenity.Core*

Sets a hint for a form field.

- Hint is shown when field label is hovered.
- This has no effect on columns.

```
public class SomeForm
{
    [Hint("Show this when my caption is hovered")]
    public string FieldWithHint { get; set; }
}
```

## CssClass Attribute

**namespace:** *Serenity.ComponentModel*, **assembly:** *Serenity.Core*

Sets CSS class for grid columns and form fields.

- In forms, class is added to container div with `.field` class that contains both label and editor.
- For columns, it sets `cssClass` property of `SlickColumn`, which adds this class to slick cells for all rows.
- Slick column headers are not affected by this attribute, use `[HeaderCssClass]` for that.

```
public class SomeForm
{
    [CssClass("extra-class")]
    public string FieldWithExtraClass { get; set; }
}

public class SomeColumn
{
    [CssClass("extra-class")]
    public string CellWithExtraClass { get; set; }
}
```

## HeaderCssClass Attribute

**namespace:** *Serenity.ComponentModel*, **assembly:** *Serenity.Core*

Sets CSS class for grid column headers.

- This has no effect for forms.

- It sets `headerCss` property of `SlickColumn`, which adds this class to slick header for that column.

```
public class SomeColumn
{
    [HeaderCssClass("extra-class")]
    public string FieldWithExtraHeaderClass { get; set; }
}
```

## AlignCenter Attribute

**namespace:** *Serenity.ComponentModel*, **assembly:** *Serenity.Core*

Centers text horizontally.

- Used to control text alignment in grids by adding `align-center` CSS class to corresponding `SlickGrid` column.
- Column headers are not affected by this attribute. You may use `[HeaderCssClass("align-center")]` for that.
- Note that it has no effect on editors or forms.

## AlignRight Attribute

**namespace:** *Serenity.ComponentModel*, **assembly:** *Serenity.Core*

Right aligns text horizontally.

- Used to control text alignment in grids by adding `align-right` CSS class to corresponding `SlickGrid` column.
- Column headers are not affected by this attribute. You may use `[HeaderCssClass("align-right")]` for that.
- Note that it has no effect on editors or forms.

## Ignore Attribute

**namespace:** *Serenity.ComponentModel*, **assembly:** *Serenity.Core*

Skips a property while generating grid column or form field list.

- Use this to ignore a property for UI, but still use it for other purposes like JSON serialization.
- This might be useful when a type is used as a Service Request and Form Declaration at the same time.

```
public class SomeColumns
{
    [Ignore]
    public string DontGenerateAColumnForMe { get; set; }
}
```

# Grids

# Formatter Types

## URLFormatter

This formatter lets you put a link with a URL to a grid column.

It takes optional arguments below:

Option Name	Description
UrlFormat	<p>This is the format of URL. A sample would be "<i>http://www.site.com/{0}</i>" where <i>{0}</i> is the UrlProperty value.</p> <p>If no format is specified, link will be the value of UrlProperty as is.</p> <p>If your URL format starts with "~/", it will be resolved to application root. For example, if format is "~/upload/{0}" and your application runs at "localhost:3045/mysite", resulting URL will be "/mysite/upload/xyz.png".</p>
UrlProperty	<p>This is name of the property that will be used to determine link URL.</p> <p>If not specified, it is the name of the column that this formatter is placed on.</p> <p>If UrlProperty value starts with "~/" it will be resolved like UrlFormat.</p>
DisplayFormat	<p>This is the display text format of link. A sample would be "<i>click to open {0}</i>" where <i>{0}</i> is the DisplayProperty value.</p> <p>If no format is specified, link will be the value of DisplayProperty as is.</p>
DisplayProperty	<p>This is name of the property that will be used to determine link text.</p> <p>If not specified, it is the name of the column that this formatter is placed on.</p>
Target	<p>This is the target of the link. Use "_blank" to open links in a new tab.</p>



# Persisting Settings

Serenity 2.1.5 introduced ability to persist grid settings including these details:

- Visible Columns and Display Order
- Column Widths
- Sorted Columns
- Advanced Filters (ones created with Edit Filter link on bottom right)
- Quick Filters (as of writing, not yet available)
- State of Include Deleted Toggle

By default, grids doesn't automatically persist anything.

Thus, if you hide some columns and navigate away from Orders page, when you come back, you'll see that those hidden columns became visible again.

You need to turn on persistence for all grids, or for individual ones that you want them to remember their settings.

## Turning On Persistence by Default

DataGrid has a static configuration parameter with name *DefaultPersistenceStorage*. This parameter controls where grids save their settings automatically by default. It is initially null.

In `ScriptInitialization.ts`, you might turn on persistence for all grids by default like below:

```
namespace Serene.ScriptInitialization {
  Q.Config.responsiveDialogs = true;
  Q.Config.rootNamespaces.push('Serene');

  Serenity.DataGrid.defaultPersistenceStorage = window.sessionStorage;
}
```

This saves settings to browser session storage, which is a key/value dictionary that preserves data while any browser window stays open. When user closes all browser windows, all settings will be lost.

Another option is to use browser local storage. Which preserves settings between browser restarts.

```
Serenity.DataGrid.defaultPersistenceStorage = window.localStorage;
```

Using any of the two options, grids will start to remember their settings, between page reloads.

## Handling a Browser that is Shared by Multiple Users

Both sessionStorage and localStorage is browser scoped, so if a browser is shared by multiple users, they'll have same set of settings.

If one user changes some settings, and logs out, and other one logs in, second user will start with settings of the first user (unless you clear localStorage on signout)

If this is a problem for your application, you may try writing a custom provider:

```
namespace Serene {
    export class UserLocalStorage implements Serenity.SettingStorage {
        getItem(key: string): string {
            return window.localStorage.getItem(
                Authorization.userDefinition.Username + ":" + key);
        }

        setItem(key: string, value: string): void {
            window.localStorage.setItem(
                Authorization.userDefinition.Username + ":" + key, value);
        }
    }
}

//...
Serenity.DataGrid.defaultPersistenceStorage = new UserLocalStorage();
```

Please note that this doesn't provide any security. It just lets users have separate settings.

## Setting Persistence Storage Per Grid Type

To turn on persistence, or change target storage for a particular grid, override getPersistenceStorage method:

```
namespace Serene.Northwind {
    //...
    export class OrderGrid extends Serenity.EntityGrid<OrderRow, any> {
        //...

        protected getPersistenceStorage(): Serenity.SettingStorage {
            return window.localStorage;
        }
    }
}
```

You may also turn off persistence for a grid class by returning *null* from this method.

## Determining Which Setting Types Are Saved

By default, all settings noted at start are saved, like visible columns, widths, filters etc. You may choose to not persist / restore specific settings. This is controlled by *getPersistenceFlags* method:

```
namespace Serene.Northwind {
    //...
    export class OrderGrid extends Serenity.EntityGrid<OrderRow, any> {
        //...

        protected getPersistenceFlags(): GridPersistenceFlags {
            return {
                columnWidths: false // dont persist column widths;
            }
        }
    }
}
```

Here is the set of complete flags:

```
interface GridPersistenceFlags {
    columnWidths?: boolean;
    columnVisibility?: boolean;
    sortColumns?: boolean;
    filterItems?: boolean;
    quickFilters?: boolean;
    includeDeleted?: boolean;
}
```

## When Settings Are Saved / Restored

Settings are automatically saved when you change something with a grid like:

- Choosing visible columns with Column Picker dialog
- Resizing a column manually
- Editing advanced filter
- Dragging a column, changing position
- Changing sorted columns

Settings are restored on first page load, just after grid creation.

## Persisting Settings to Database (User Preferences Table)

Serene 2.1.5 comes with a *UserPreferences* table that you may use as a persistence storage. To use this storage, you just need to set it as storage similar to other storage types.

```
/// <reference path="../../Common/UserPreference/UserPreferenceStorage.ts" />

Serenity.DataGrid.defaultPersistenceStorage = new Common.UserPreferenceStorage();
```

Don't forget to add reference statement, or you'll have runtime errors, as TypeScript has problems with ordering otherwise.

OR

```
namespace Serene.Northwind {
    //...
    export class OrderGrid extends Serenity.EntityGrid<OrderRow, any> {
        //...

        protected getPersistenceStorage(): Serenity.SettingStorage {
            return new Common.UserPreferenceStorage();
        }
    }
}
```

## Manually Saving / Restoring Settings

If you need to save / restore settings manually, you may use methods below:

```
protected getCurrentSettings(flags?: GridPersistenceFlags): PersistedGridSettings;
protected restoreSettings(settings?: PersistedGridSettings, flags?: GridPersistenceFlags): void;
```

These are protected methods of DataGrid, so can only be called from subclasses.



# Code Generator (Sergen)

Sergen has some extra options that you may set through its configuration file (Serenity.CodeGenerator.config) in your solution directory.

Here is the full set of options:

```
public class GeneratorConfig
{
    public List<Connection> Connections { get; set; }
    public string KDiff3Path { get; set; }
    public string TFPath { get; set; }
    public string TSCPath { get; set; }
    public bool TFSIntegration { get; set; }
    public string WebProjectFile { get; set; }
    public string ScriptProjectFile { get; set; }
    public string RootNamespace { get; set; }
    public List<BaseRowClass> BaseRowClasses { get; set; }
    public List<string> RemoveForeignFields { get; set; }
    public bool GenerateSSImports { get; set; }
    public bool GenerateTSTypings { get; set; }
    public bool GenerateTSCode { get; set; }
}
```

Connections, RootNamespace, WebProjectFile, ScriptProjectFile, GenerateSSImports, GenerateSSTypings and GenerateTSCode options are all available in user interface, so we'll focus on other options.

## KDiff3 Path

Sergen tries to launch KDiff3 when it needs to merge changes to an existing file. This might happen when you try to generate code for an entity again. Instead of overriding target files, Sergen will execute KDiff3.

Sergen looks for KDiff3 at its default location under C:\Program Files\Kdiff3, but you may override this path with this option, if you installed Kdiff3 to another location.

## TFSIntegration and TFPath

For users that work with TFS, Sergen provides this options to make it possible to checkout existing files and add new ones to source control. Set TFSIntegration to true, if your project is versioned in TFS, and set TFPath if tf.exe is not under its default location at C:\Program Files\Visual Studio\x.y\Common7\ide\

```
{
  // ...
  "TFSIntegration": true,
  "TFPath": "C:\Program Files\...\tf.exe"
}
```

## RemoveForeignFields

By default, Sergen examines your table foreign keys, and when generating a row class, it will bring all fields from all referenced foreign tables.

Sometimes, you might have some fields in foreign tables, e.g. some logging fields like `InsertUserId`, `UpdateDate` etc. that wouldn't be useful in another row.

You'd be able to remove them manually after code generation too, but using this option it might be easier. List fields you want to remove from generated rows as an array of string:

```
{
  // ...
  "RemoveForeignFields": ["InsertUserId", "UpdateUserId",
    "InsertDate", "UpdateDate"]
}
```

Note that this doesn't remove these fields from table row itself, it only removes these view fields from foreign joins.

## BaseRowClasses

If you are using some base row class, e.g. something like `LoggingRow` in `Serene`, you might want Sergen to generate your rows deriving from these base classes.

For this to work, list your base classes, and the fields they have.

```
{
  // ...
  "BaseRowClasses": [{
    "ClassName": "Serene.Administration.LoggingRow",
    "Fields": ["InsertUserId", "UpdateUserId",
      "InsertDate", "UpdateDate"]
  }]
}
```

If Sergen determines that a table has all fields listed in "Fields" array, it will set its base class as "ClassName", and will not generate these fields explicitly in row, as they are already defined in base row class.

It is possible to define more than one base row class. Sergen will choose the base row class with most matching fields, if a row's fields matches more than one base class.



## Used Tools and Libraries

Serenity platform makes use of some valuable open source tools and libraries that are listed below (in alphabetic order)

This list might seem a bit long, but not all of them are direct dependencies for a Serenity Application.

Some of them are only used during development of Serenity platform itself, while some are dependencies for optional features.

We tried to reuse open source libraries, where there is a quality one available to avoid reinventing the wheel.

**Autonumeric** (<https://github.com/BobKnothe/autoNumeric>)

**BlockUI** (<https://github.com/malsup/blockui/>)

**Bootstrap** (<https://github.com/twbs/bootstrap>)

**Cake Build** (<https://github.com/cake-build/cake>)

**Cecil** (<https://github.com/jbevain/cecil>)

**Clean-CSS [Node]**  
(<https://github.com/jakubpawlowicz/clean-css>)

**Colorbox** (<https://github.com/jackmoore/colorbox>)

**Dapper** (<https://github.com/StackExchange/dapper-dot-net>)

**DialogExtend** (<https://github.com/ROMB/jquery-dialogextend>)

**jLayout** (<https://github.com/bramstein/jlayout>)

**Json.NET** (<https://github.com/JamesNK/Newtonsoft.Json>)

**JSON2** (<https://github.com/douglascrockford/JSON-js>)

**JSRender** (<https://github.com/BorisMoore/jsrender>)

**jQuery** (<https://github.com/jquery/jquery>)

**jQuery Cookie** (<https://github.com/carhartl/jquery-cookie>)

**jQuery Validation** (<https://github.com/jzaefferer/jquery-validation>)

**jQuery UI** (<https://github.com/jquery/jquery-ui>)

**jQuery.event.drag**  
(<http://threedubmedia.com/code/event/drag>)

**Less.JS (Node)** (<https://github.com/less/less.js>)

**Linq.js** (<http://linqjs.codeplex.com/>)

**metisMenu** (<https://github.com/onokumus/metisMenu>)

**Munq** (<https://munq.codeplex.com/>)

**NodeJS** (<https://github.com/joyent/node>)

**Pace** (<https://github.com/HubSpot/pace>)

**PhantomJS** (<https://github.com/ariya/phantomjs>)

**RazorGenerator** (<https://razorgenerator.codeplex.com/>)

**RSVP** (<https://github.com/tildeio/rsvp.js/>)

**Saltarelle Compiler** (<https://github.com/erik-kallen/SaltarelleCompiler>)

**Select2** (<https://github.com/ivaynberg/select2>)

**SlickGrid** (<https://github.com/mleibman/SlickGrid>)

**Toastr** (<https://github.com/CodeSeven/toastr>)

**UglifyJS2 (Node)** (<https://github.com/mishoo/UglifyJS2>)

**XUnit** (<https://github.com/xunit/xunit>)