



Magento® U

## Unit One Contents

About This Guide .....	vii
1. Fundamentals Introduction .....	1
1.1 Magento 2 Fundamentals: Unit One Home Page .....	1
1.2 Home Page .....	2
1.3 Introduction .....	3
1.4 Appropriate Audience .....	4
1.5 Course Content .....	5
1.6 How to Access Course Modules .....	6
2. Unit One Home Page .....	7
2.1 Magento 2 Fundamentals: Unit One Home Page .....	7
3. Preparation .....	8
3.1 Preparation .....	8
3.2 Module Topics .....	9
3.3 Environment Setup Requirements .....	10
3.4 Modes in Magento 2 .....	11
3.5 Developer Mode in Magento 2 .....	12
3.6 Production Mode in Magento 2 .....	13
3.7 Default Mode in Magento 2 .....	14
3.8 Production Mode in Magento 2 .....	15
3.9 Production Mode in Magento 2 .....	16
3.10 Specifying a Mode .....	17
3.11 Specify a Mode Using an Environment Variable .....	18
3.12 Code Demonstration   Specify Mode using Env. Variable .....	19
3.13 Specifying a Mode   Web Server Environment .....	20
3.14 Specifying a Mode   php.fpm Environment .....	21
3.15 Reinforcement Exercise (1.3.1) .....	22
4. Overview and Architecture .....	23
4.1 Magento 2 Overview and Architecture .....	23
4.2 Module Topics .....	24
4.3 Magento 2 Platform .....	25
4.4 Magento 2 Goals .....	26
4.5 Magento 2 Architecture .....	27
4.6 High-Level Client Server Structural Map .....	28
4.7 High-Level Application Map .....	29
4.8 Modular System .....	30
4.9 Areas .....	31
4.10 Libraries .....	32

4.11 Language Packs .....	33
4.12 Themes .....	34
4.13 Modules: Definition .....	35
4.14 Modules: Location .....	36
4.15 Modules: Naming a Module .....	37
4.16 Modules: Declaring a Module .....	38
4.17 Modules: Module Dependencies .....	39
4.18 Declare Dependency with Other Modules .....	40
4.19 Module Dependencies Tasks .....	41
4.20 Modules: Types of Module Dependencies .....	42
4.21 Reinforcement Exercise (1.4.1) .....	43
5. File Systems .....	44
5.1 File Systems .....	44
5.2 Module Topics .....	45
5.3 File System: Library .....	46
5.4 Code Demonstration: File Systems .....	47
5.5 File System: Core Source Code .....	48
5.6 File System: Module Structure .....	49
5.7 File System: Templates .....	50
5.8 File System: Templates .....	51
5.9 Code Demonstration: Templates .....	52
5.10 File System: Skin to Theme .....	53
6. Configuration .....	54
6.1 Configuration .....	54
6.2 Module Topics .....	55
6.3 Configuration Files .....	56
6.4 Configuration Files: Storage .....	57
6.5 Configuration Files: core_config_data .....	58
core_config_data interface (Slide Layer) .....	59
6.6 Configuration Files: Custom Config Files .....	60
6.7 Configuration Files: Scope .....	61
6.8 Configuration Files: Load Order .....	62
6.9 Configuration Files: Merging .....	63
6.10 Configuration Files: Validation .....	64
6.11 Magento\Config .....	65
6.12 Magento\Config - Loading Infrastructure .....	66
6.13 Code Demonstration: events.xml .....	67
6.14 Validating a Configuration Type .....	68
6.15 Check Your Understanding .....	69
6.16 Check Your Understanding .....	70
6.17 Error Reporting Settings .....	71





7. DI & Object Manager .....	72
7.1 DI & Object Manager .....	72
7.2 Module Topics .....	73
7.3 Dependency Injection (DI) Pattern .....	74
7.4 Dependency Injection .....	75
7.5 Using Dependency Injection .....	76
7.6 Dependency Injection Object Manager Classes .....	79
7.7 Dependency Injection   Creating Objects .....	80
7.8 Object Manager .....	81
7.9 Object Manager: Shared Instances Concept .....	82
7.10 Object Manager Configuration .....	83
7.11 Object Manager Configuration: Preferences Example .....	85
7.12 Object Manager Configuration: Argument Example .....	86
7.13 Code Demonstration: Object Manager Argument Example .....	87
7.14 Object Manager Configuration: Shared Argument Example .....	88
7.15 Object Manager Configuration .....	89
7.16 Object Manager Initiation Process .....	90
7.17 Objects: Injectable and Non-injectable .....	91
7.18 Objects: Injectable and Non-injectable Rules .....	92
7.19 Objects: Injectable and Non-injectable Rules 2 .....	93
7.20 Class Definition and Definition Compiler Tool .....	94
7.21 Class Definition andRunning the Definition Compiler Tool .....	95
7.22 Definition Compiler Tool Best Practice .....	96
7.23 Cache in Magento 2 .....	97
7.24 Cache Configuration .....	98
7.25 Cache Type .....	99
7.26 Cache Cleaning .....	100
7.27 Reinforcement Exercise (1.7.1) .....	101
8. Plugins .....	102
8.1 Plugins .....	102
8.2 Module Topics .....	103
8.3 Plugins: Definition .....	104
8.4 Plugins: Definition .....	105
8.5 Plugins: Interception .....	106
8.6 Declaring a Plugin .....	107
8.7 Before-Listener Method .....	108
8.8 After-Listener Method .....	109
8.9 Around-Listener Method .....	110
8.10 Prioritizing Plugins .....	111
8.11 Configuration Inheritance .....	112
8.12 Code Demonstration: Plugin .....	113

8.13 Reinforcement Exercise 1-8-1 .....	114
9. Events .....	115
9.1 Events.....	115
9.2 Module Topics .....	116
9.3 Events: Definition.....	117
9.4 Events: Schema .....	118
Event Firing Code (Slide Layer).....	119
9.5 Registering an Event in XML.....	120
9.6 Observer Example .....	121
9.7 Code Demonstration: Registering an Event .....	122
9.8 Reinforcement Exercise (1.9.1) .....	123

## About This Guide

---

This guide uses the following symbols in the notes that follow the slides.

Symbol	Indicates...
	A note, tip, or other information brought to your attention.
	Important information that you need to know.
	A cross-reference to another document or website.
	Best practice recommended by Magento

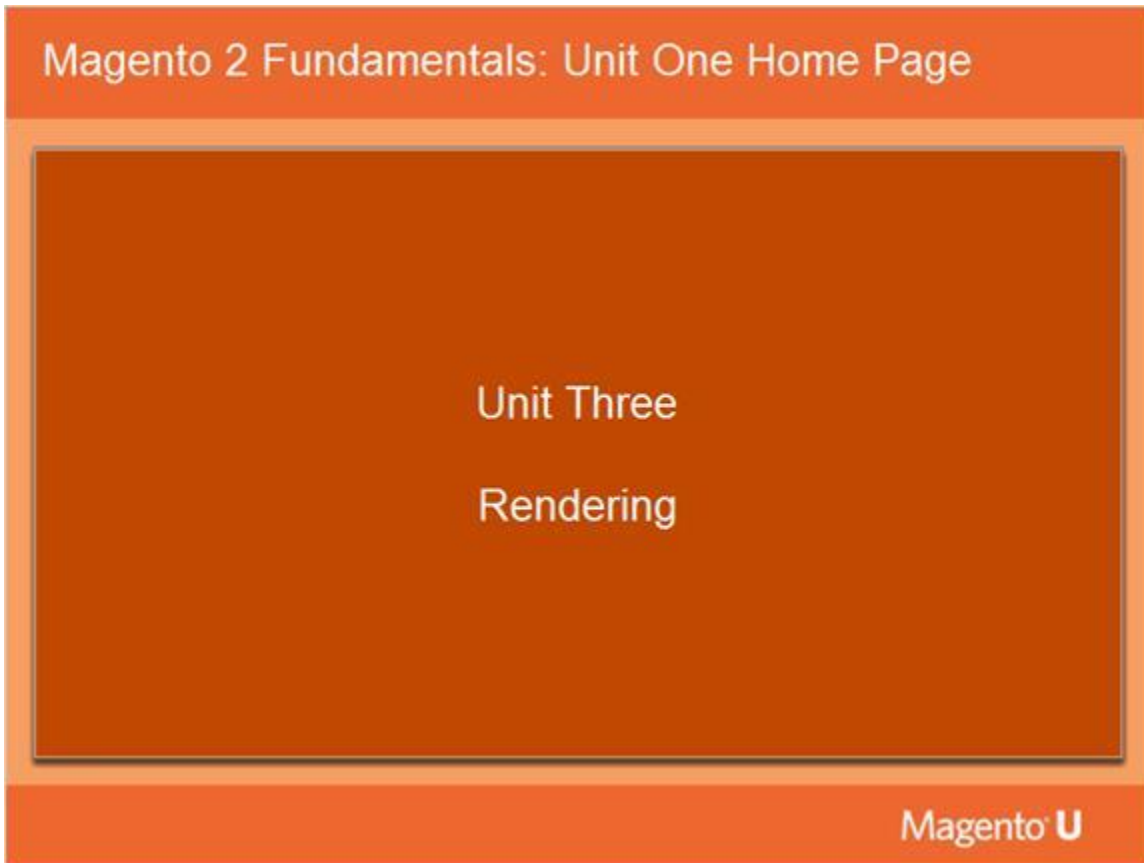




# 1. Fundamentals Introduction

---

## 1.1 Magento 2 Fundamentals: Unit One Home Page



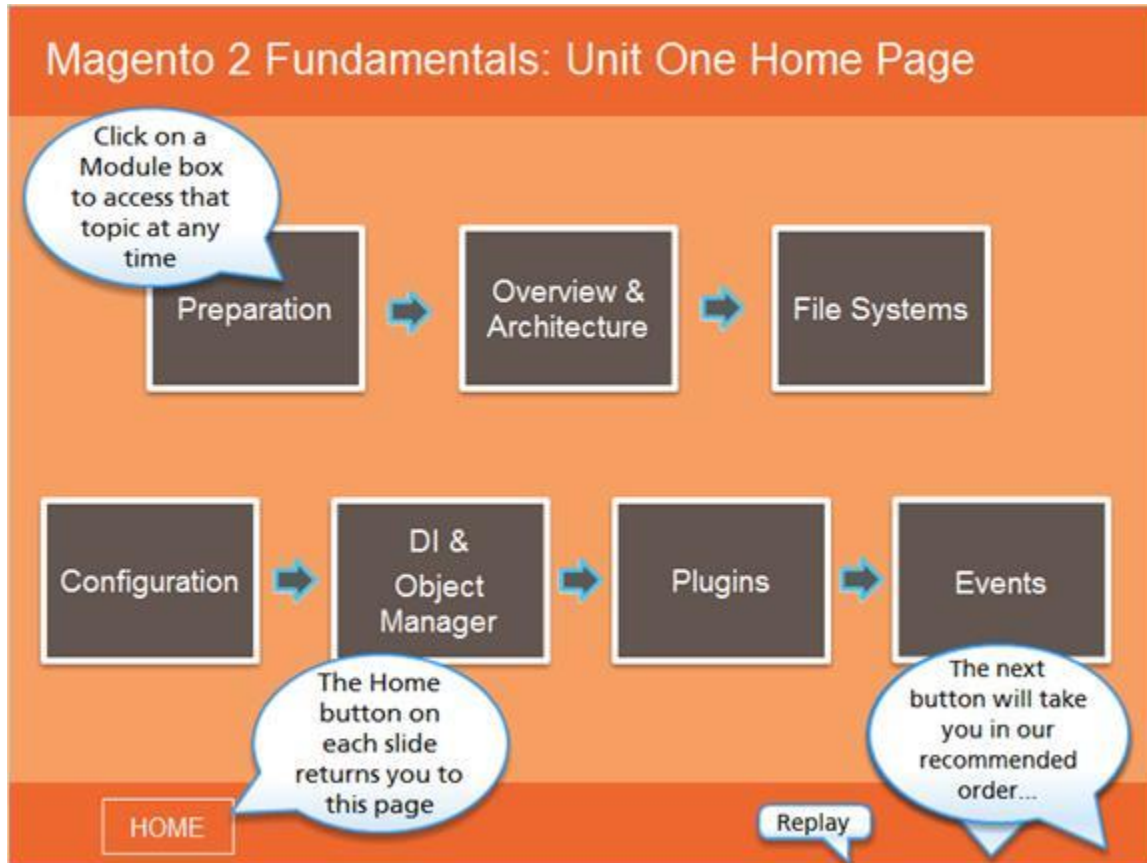
**Notes:**

Welcome to the Magento 2 Fundamentals course!

This course contains three different units -- Preparation & Configuration, Request Routing, and Rendering. You access each of the units as a separate course.

Each unit contains a number of modules. Here, in Unit One, there are seven modules.

## 1.2 Home Page



### Notes:

Each module can be accessed at any time by clicking on the appropriate box. The arrows indicate the proper sequence for taking the course the first time through.

The Home button on each of the following course slides will return you to this page so you can replay a module or select a new one.

The controls on the player allow you to replay a slide (backward curved arrow), proceed to the next slide, or return to the previous slide.

Enjoy your course!

## 1.3 Introduction

Introduction

Why should I take this course?



This is an extensive, challenging course that presents essential concepts you need to learn about Magento 2.

It assumes you have prior experience with Magento 1 and highlights changes in features and functionality between the two product versions.

**Notes:**

This is an extensive, challenging course that presents essential concepts you need to learn about Magento 2 over a series of comprehensive units. It assumes you have prior experience with Magento 1 and highlights changes in features and functionality between the two product versions throughout the course.

Be sure to give yourself enough time to go through the material at your own pace.

The built-in navigation allows you to repeat any slide or module as many times as you need, in any order you want.

## 1.4 Appropriate Audience

Appropriate Audience

This course is appropriate for developers...

... who are experienced with launching and extending the Magento platform.

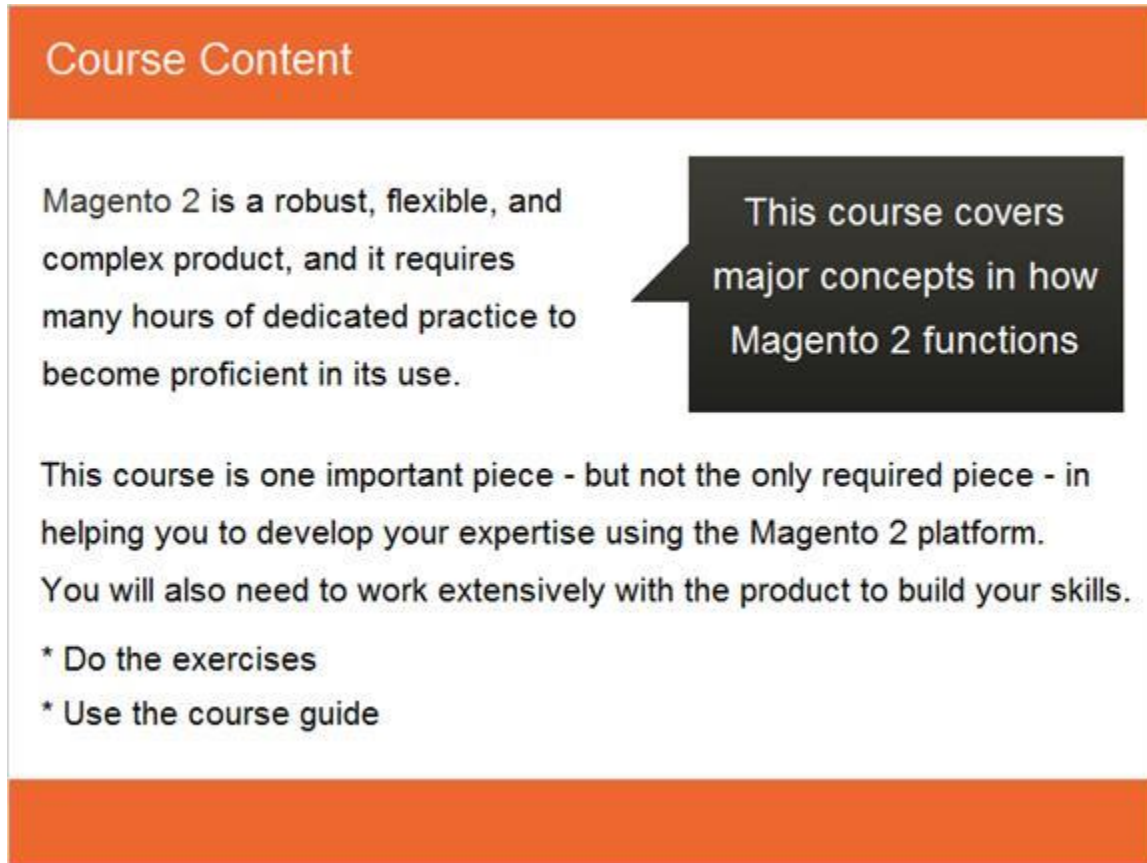
The depth of the content may prove challenging to new developers who have no familiarity with the Magento product.

**Notes:**

This course is appropriate for developers who are experienced with launching and extending the Magento platform.

The depth of the content may prove challenging to new developers who have no familiarity with the Magento product.

## 1.5 Course Content

A slide titled "Course Content" with an orange header and footer. The main content area is white. It features a paragraph about Magento 2's complexity, a callout box stating the course covers major concepts, a paragraph about the course being one piece of a larger puzzle, and a bulleted list of two tasks: "Do the exercises" and "Use the course guide".

**Course Content**

Magento 2 is a robust, flexible, and complex product, and it requires many hours of dedicated practice to become proficient in its use.

This course covers major concepts in how Magento 2 functions

This course is one important piece - but not the only required piece - in helping you to develop your expertise using the Magento 2 platform. You will also need to work extensively with the product to build your skills.

- \* Do the exercises
- \* Use the course guide

### Notes:

Magento 2 is a robust, flexible, and complex product, and it requires many hours of dedicated practice to become proficient in its use. This course is one important piece - but not the only required piece - in helping you to develop your expertise using the Magento 2 platform. You will also need to work extensively with the product to build your skills.

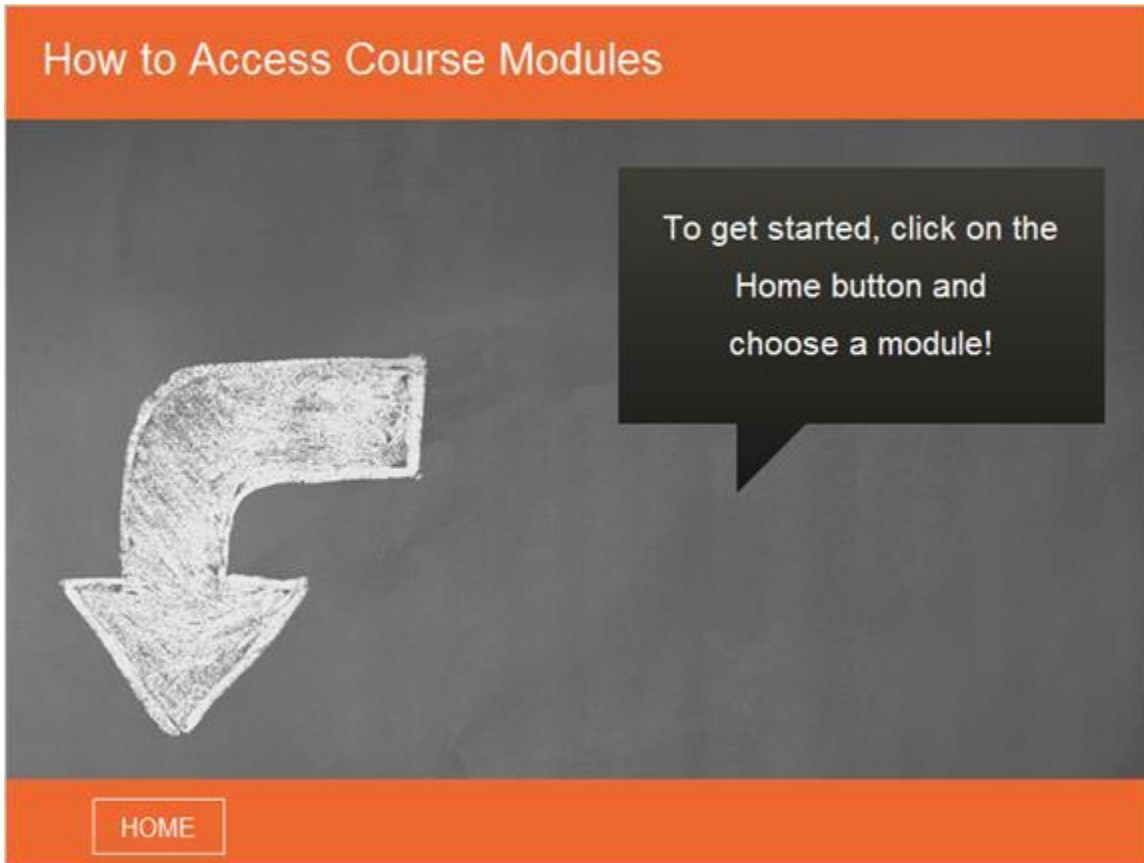
Be sure to do all the exercises presented in the course, as these are key learning opportunities and provide practical, hands-on experience with the native Magento 2 installation.

The course guide presents the narration in written form, while the slide highlights key concepts. Find the most effective way to use both to match your learning style. For example, you may want to read the slide first for a general grasp of the topic, then play the audio as you follow along with the full notes, to fill in all the details.

Remember, you can replay each slide as many times as you need to comprehend the material.

At this point in time, Magento 2 is still in Beta release, and so some functionality may change before the first official release of the product. The content in this course is based on Magento 2 Beta 0.74.

## 1.6 How to Access Course Modules

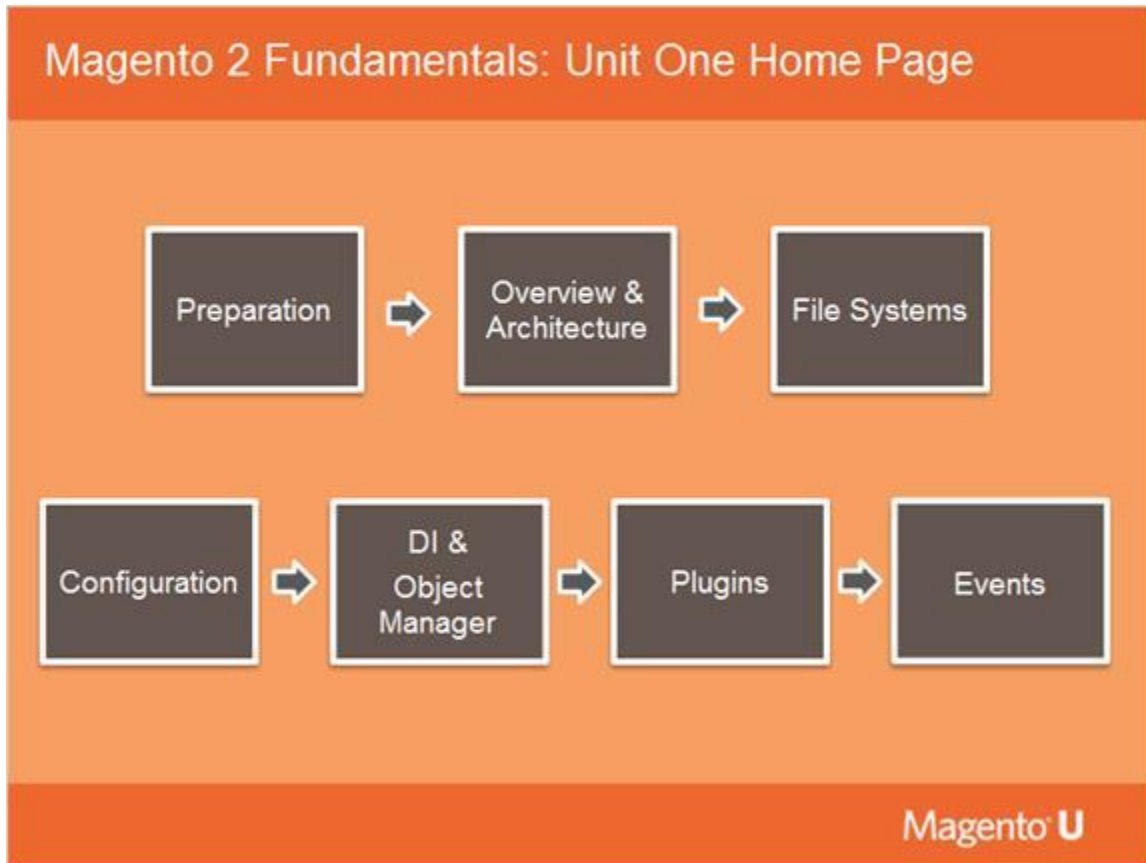


**Notes:**

To get started with the course, click on the Home button now!

## 2. Unit One Home Page

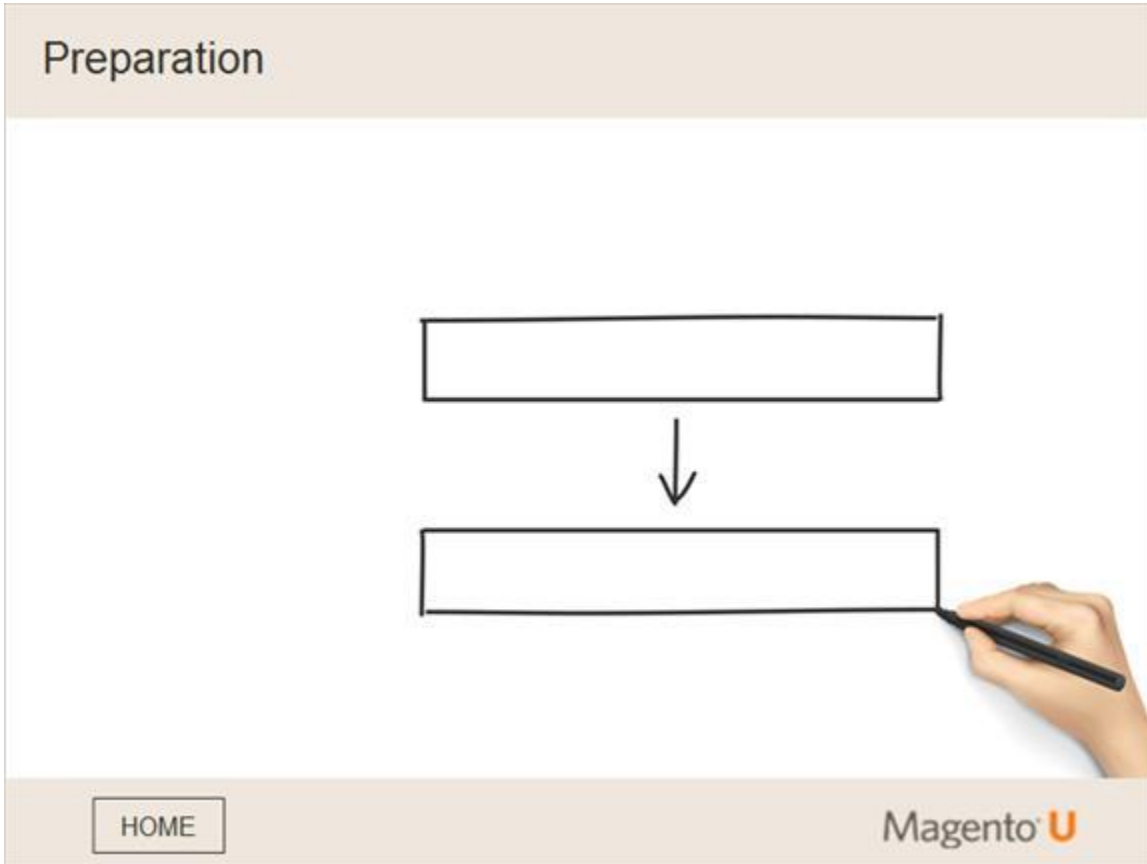
### 2.1 Magento 2 Fundamentals: Unit One Home Page



## 3. Preparation

---

### 3.1 Preparation



**Notes:**

In this module, we will discuss the preparation steps you should take in setting up your Magento 2 installation. We will cover configuration aspects in the next module.



## 3.2 Module Topics

### Module Topics



**In this module, we will discuss...**

- Environment Setup Requirements
- Modes

[HOME](#) **Magento U**

**Notes:**

In particular, this module will discuss environment setup requirements and the system modes available for different phases of the development lifecycle.

## 3.3 Environment Setup Requirements

Preparation | Environment Setup Requirements

LAMP Structure +  
Composer

Magento 2 requires the LAMP Structure + Composer

Linux \* Apache \* MySQL \* PHP \* Composer

HOME

Magento U

#### Notes:

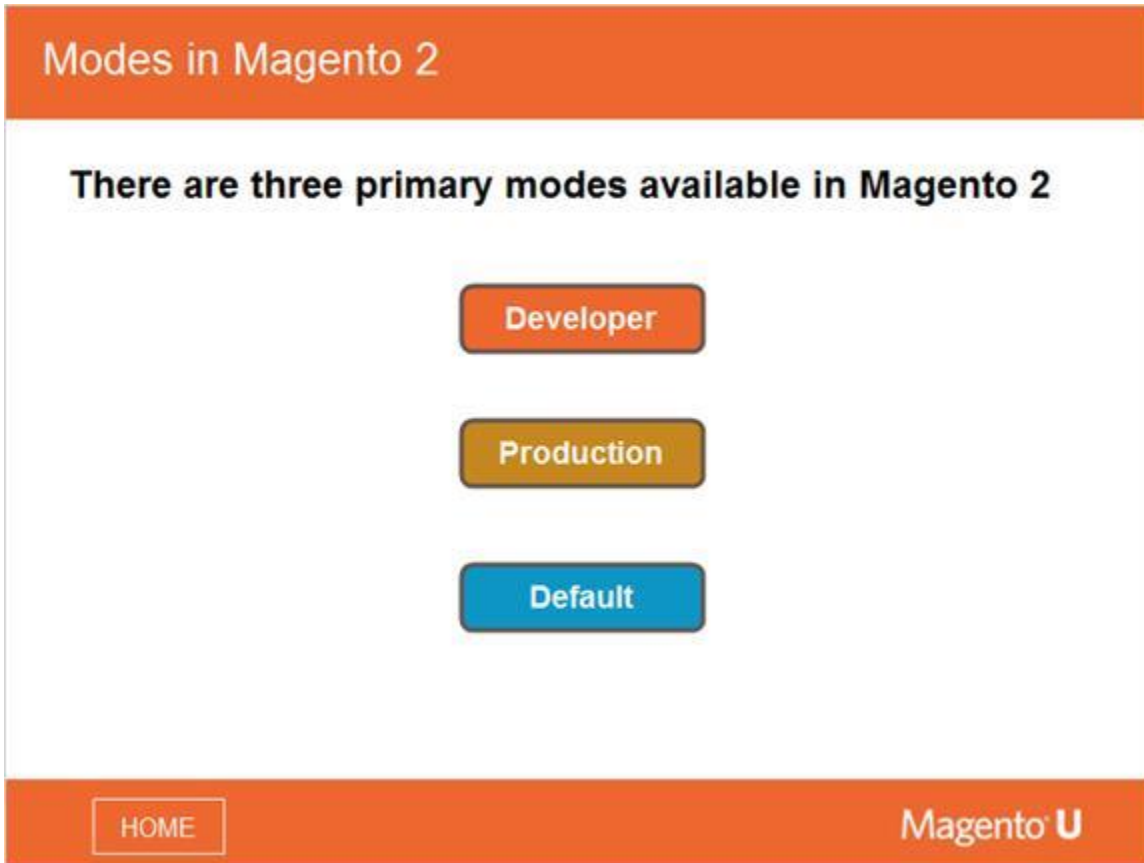
Magento 2 requires the familiar LAMP (Linux, Apache, MySQL, and PHP) structure along with PHP Composer, a tool for managing external dependencies. Magento uses Composer for downloading and installing external dependencies such as Symfony.

#### Reference:

The following are required for installing Magento 2:

- PHP Extensions
  - PDO/MySQL
  - Mbstring
  - Mcrypt
  - Mhash
  - Simplexml
- Curl
- Gd2, ImageMagick 6.3.7+, or both
- SOAP
- Application Parameters
- Website
- Environmental Parameters

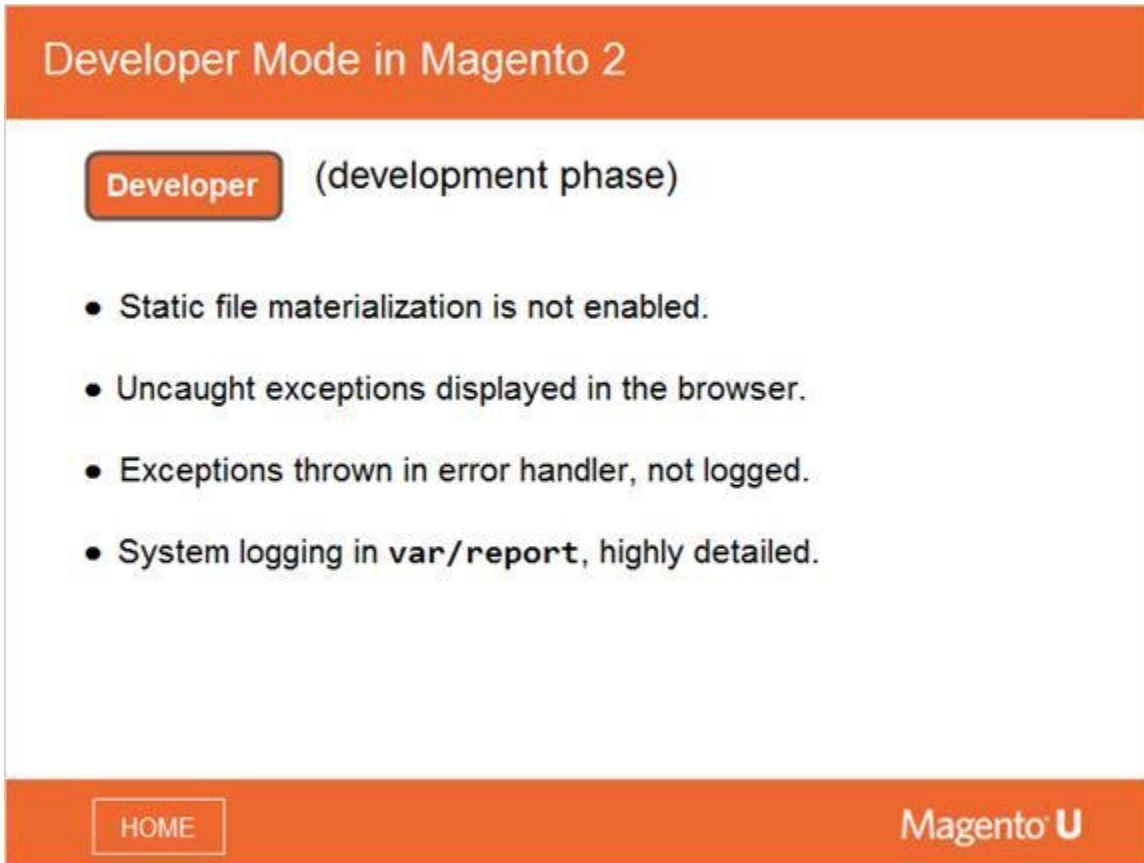
## 3.4 Modes in Magento 2

**Notes:**

Magento 2 has three primary modes – developer, production, and default.

There is also a maintenance mode, but that operates in a different way, only to prevent access to the system.

## 3.5 Developer Mode in Magento 2



### Notes:

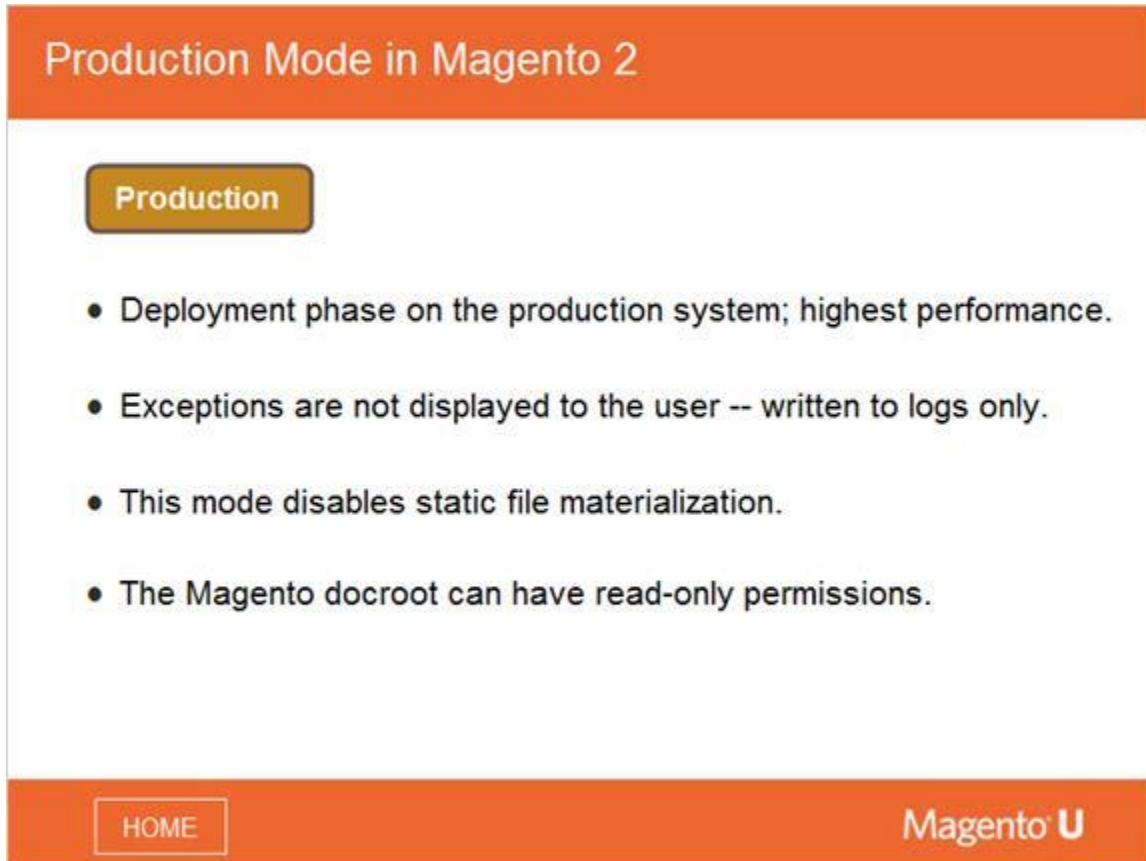
You should use the Developer mode while you are developing the code for the application. The main benefit to this mode is that error messages are visible to you. It should not be used once in production because of its impact on performance.

In Developer mode, static view files are not cached; instead, they are written to the Magento docroot every time they are called.

Uncaught exceptions are displayed in the browser, while exceptions are thrown in the error handler, rather than being logged. An exception is thrown whenever an event subscriber cannot be invoked.

System logging in var/report is highly detailed in this mode.

## 3.6 Production Mode in Magento 2

The image is a screenshot of a web page titled "Production Mode in Magento 2". At the top, there is an orange header bar with the title in white text. Below the header, on the left, is a yellow button with the word "Production" in black. To the right of the button is a list of four bullet points. At the bottom of the page, there is an orange footer bar. On the left of the footer bar is a white button with the word "HOME" in black. On the right of the footer bar is the "Magento U" logo in white.

### Production Mode in Magento 2

**Production**

- Deployment phase on the production system; highest performance.
- Exceptions are not displayed to the user -- written to logs only.
- This mode disables static file materialization.
- The Magento docroot can have read-only permissions.

**HOME** **Magento U**

### Notes:

You should run Magento in Production mode once it is deployed to a production server.

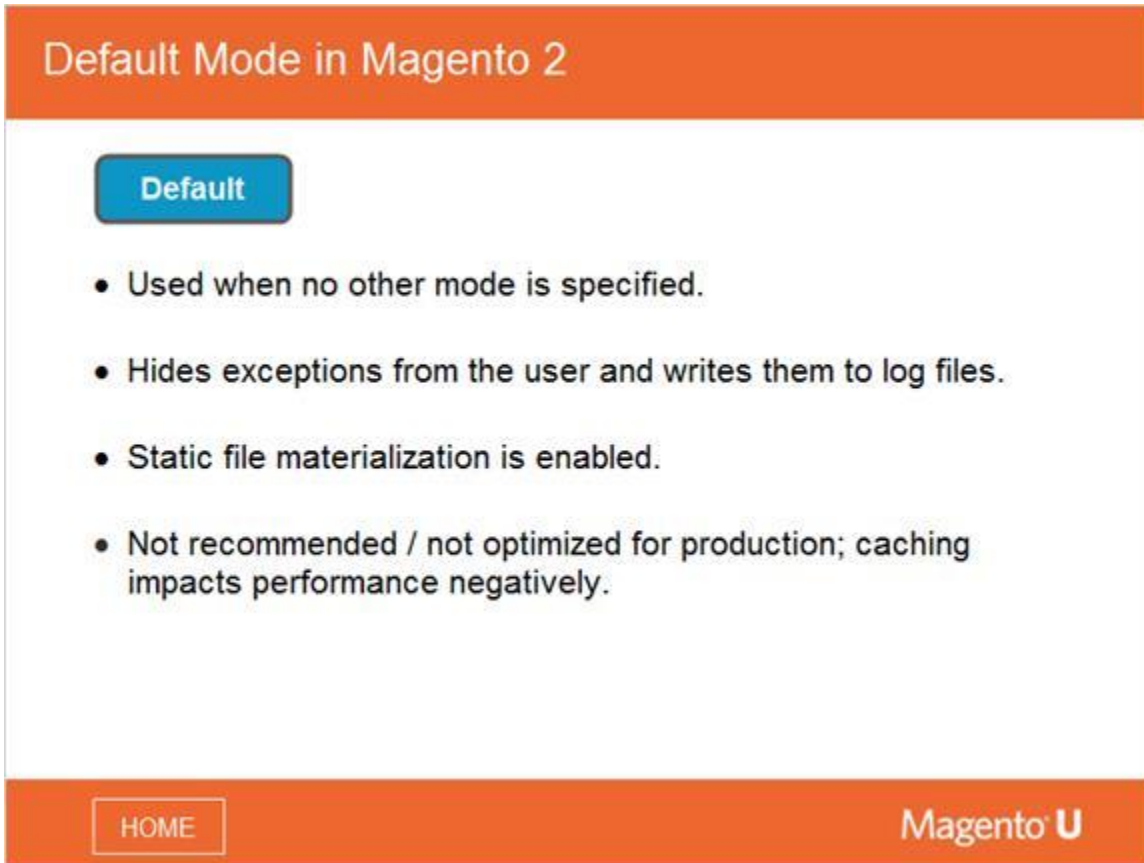
Production mode provides the highest performance in Magento 2.

The most important aspect of this mode is that errors are logged to the file system and are never displayed to the user.

In this mode, view files are not materialized; instead, URLs for the files are composed on-the-fly, without going through the fallback mechanism.

The Magento docroot has read-only permissions.

## 3.7 Default Mode in Magento 2



The screenshot shows a web interface for configuring the Magento 2 default mode. At the top, there is an orange header with the text "Default Mode in Magento 2". Below this, a blue button labeled "Default" is highlighted. Underneath the button, there is a list of four bullet points describing the default mode. At the bottom of the page, there is an orange footer containing a "HOME" button and the "Magento U" logo.

### Default Mode in Magento 2

**Default**

- Used when no other mode is specified.
- Hides exceptions from the user and writes them to log files.
- Static file materialization is enabled.
- Not recommended / not optimized for production; caching impacts performance negatively.

[HOME](#) **Magento U**

### Notes:

As its name implies, Default mode is how the Magento software operates if no other mode is specified.

In this mode, errors are logged to file reports at the server, and are never shown to a user. Static file materialization is enabled.

Default mode is not optimized for a production environment, primarily because of the adverse performance impact of static files being cached rather than materialized.

In other words, creating static files and caching them has a greater performance impact than generating them using the static file creation tool.

## 3.8 Production Mode in Magento 2

Summary of Mode Features				
	Static File Caching	Exceptions Displayed	Exceptions Logged	Performance (-) Impacted
Developer		✓		✓
Production			✓	
Default	✓		✓	✓

HOME

Magento U

**Notes:**

Here is a summary of the key features that distinguish each mode.

## 3.9 Production Mode in Magento 2

### Maintenance Mode in Magento 2

Maintenance

- Used to make a site unavailable to the public during updates or other changes.
- `Bootstrap::assertMaintenance()` controls this mode; you have to create a flag (`var/.maintenance.flag`) to enable the mode.
- Can specify a group of people that can have access during this time (`var/.maintenance.ip`).
- Maintenance mode is an out-of-the-box feature in Magento 2.

HOME

Magento U

### Notes:

Maintenance mode is used when you want to make the site unavailable to the public during updates or other changes.

The method `Bootstrap::assertMaintenance()` controls this mode, and you have to create a flag file (`var/.maintenance.flag`) to enable the mode.

You can specify a group of people to have access to the site while this mode is employed by placing the associated IPs in the file (`var/.maintenance.ip`).

Maintenance mode is an out-of-the-box feature in Magento 2.



## 3.10 Specifying a Mode

### Specifying a Mode

**Specify modes in one of two ways...**

- Use an environment variable
- Use the web server or PHP-FPM environment



[HOME](#)Magento U

### Notes:

There are basically two ways in which you can set the mode for your system – using an environment variable, or using the web server environment.

## 3.11 Specify a Mode Using an Environment Variable

### Specifying a Mode | Environment Variable

#### Specify a Mode Using an Environment Variable

Use the MAGE\_MODE system environment variable to specify a mode as follows:

```
MAGE_MODE=[developer|default|production]
```

After setting the mode, restart the web server:

- Ubuntu: `service apache2 restart`
- CentOS: `service httpd restart`

[HOME](#)Magento U

#### Notes:

Using environment variables allows you to set the mode for your Linux, Windows, Apple, or any other system. However, this approach has disadvantages.

All the environment variables differ depending upon how they are set.

The commands used to specify environment variables will depend upon the system.

## 3.12 Code Demonstration | Specify Mode using Env. Variable



### Notes:

If you are using an Apache system, you can follow along with the demo (Apache for NJX will be a little different).

- Open Apache
- Open the `.htaccess` file

*Hopefully, your Apache is configured to allow HTML files.*

- Set environment Mage mode to "developer".

Let's test what happens in this mode by making a mistake in the code.

With developer mode enabled, you will see all error messages on the screen.

The error just created is visible on the screen. If the mode were instead "default", you would see an error number but not what happened exactly. If you then went into the log file, you would see the actual errors in there. Of course, this may not be true for parse errors or fatal errors, as these will always show you a message on the screen.

Maintenance mode works in a different manner. You don't need to use mage mode variables - instead, you touch a file. I want to show you how to touch the file maintenance flag in the var folder. When maintenance mode is enabled, the screen will show the error message, "503 Internal server error". If you disable the developer mode to default, this is also the message that you will see.

This is the purpose of maintenance mode - to show people error messages while still being able to work with the files. In this mode, you can create a file, `var/.maintenance.ip.`, where you put the IP to make the site available.

Currently, maintenance mode is a file that you have to create; most likely in the future it will be a button that will allow you to put your site in maintenance mode.

## 3.13 Specifying a Mode | Web Server Environment

### Specifying a Mode | Web Server Environment

#### Specify a Mode Using the Web Server Environment

- Apache web servers with `mod_php` support this method, using `mod_env` directives.
- The Apache directive is slightly different in versions 2.2 and 2.4. Consult the Apache documentation for more information and guidance.

```
SetEnv MAGE_MODE=[developer|default|production]
```

(This should be set in the `.htaccess` file.)

[HOME](#)Magento U

#### Notes:

The other approach to setting mode is to set environment variables in the web server.

The most popular server type is probably Apache. In Apache, more environment variables are set; the `.htaccess` file is a good example.

You can set up a mode using the construction links in Magento Default, Developer, or Production mode.

### 3.14 Specifying a Mode | php-fpm Environment

#### Specifying a Mode | php-fpm Environment

##### Specify a Mode Using the php-fpm Environment

- Specify the mode in the php-fpm config, or in the system environment in which php-fpm is started.
- In the php-fpm config, the value can be set as follows:

```
env[MAGE_MODE]=[developer|default|production]
```

(The location of the php-fpm config file depends upon your system.)

[HOME](#)

Magento U

**Notes:**

For instructions on specifying the mode using the php-fpm system environment, please consult your OS documentation.

### 3.15 Reinforcement Exercise (1.3.1)

#### Reinforcement Exercise (1.3.1)

Take 15 minutes to explore your own system:

- If you are using a `php-fpm` setup, you can look at the `phpinfo()` for the location of the config file to change.
- If you are using a `mod_php` setup with Apache, try setting the mode using an `.htaccess` file.

[HOME](#)

Magento **U**

## 4. Overview and Architecture

### 4.1 Magento 2 Overview and Architecture

**Notes:**

In this overview module, we will provide an introduction to Magento 2, focusing on its architecture and the key component of its modular system, modules.

## 4.2 Module Topics

### Module Topics



**In this module, we will discuss...**

- Overview of Magento 2 Platform
- Architecture overview
- Modules

[HOME](#)Magento U

**Notes:**

The topics to be addressed in this module are: an overview of the Magento 2 platform, an overview of its architecture, and a discussion around modules and their role in Magento 2's system.



## 4.3 Magento 2 Platform

The slide features an orange header with the text 'Magento 2 Platform'. Below the header, on the right, is a dark grey speech bubble containing the text 'The Magento 2 platform is...'. On the left, there is a bulleted list of three points. At the bottom, there is an orange footer bar containing a 'HOME' button on the left and the 'Magento U' logo on the right.

### Magento 2 Platform

The Magento 2 platform is...

- A flexible, open source eCommerce platform and content management system
- Written in PHP, and leverages elements of the Zend Framework and MVC architecture
- Extremely configurable

[HOME](#) **Magento U**

### Notes:

Magento 2 continues to provide the same flexibility and extensibility of Magento 1, while introducing structural changes that address challenges around describing dependencies across the system (for easier upgrades), and around containing business logic within one system layer. All modules are developed in PHP so they are related in a consistent way.

Magento 2's instantiation mechanism and code generation provides numerous ways to customize, such as through the use of plugins.

### 4.4 Magento 2 Goals



#### Notes:

Built on a new and modern technology stack, Magento 2 integrates better with third-party solutions, and is more accessible and open to frontend developers. With an improved implementation process, partners and merchants will realize faster deployments, simplified upgrades, and faster time to value.

There were six key goals in designing and producing the Magento 2 platform:

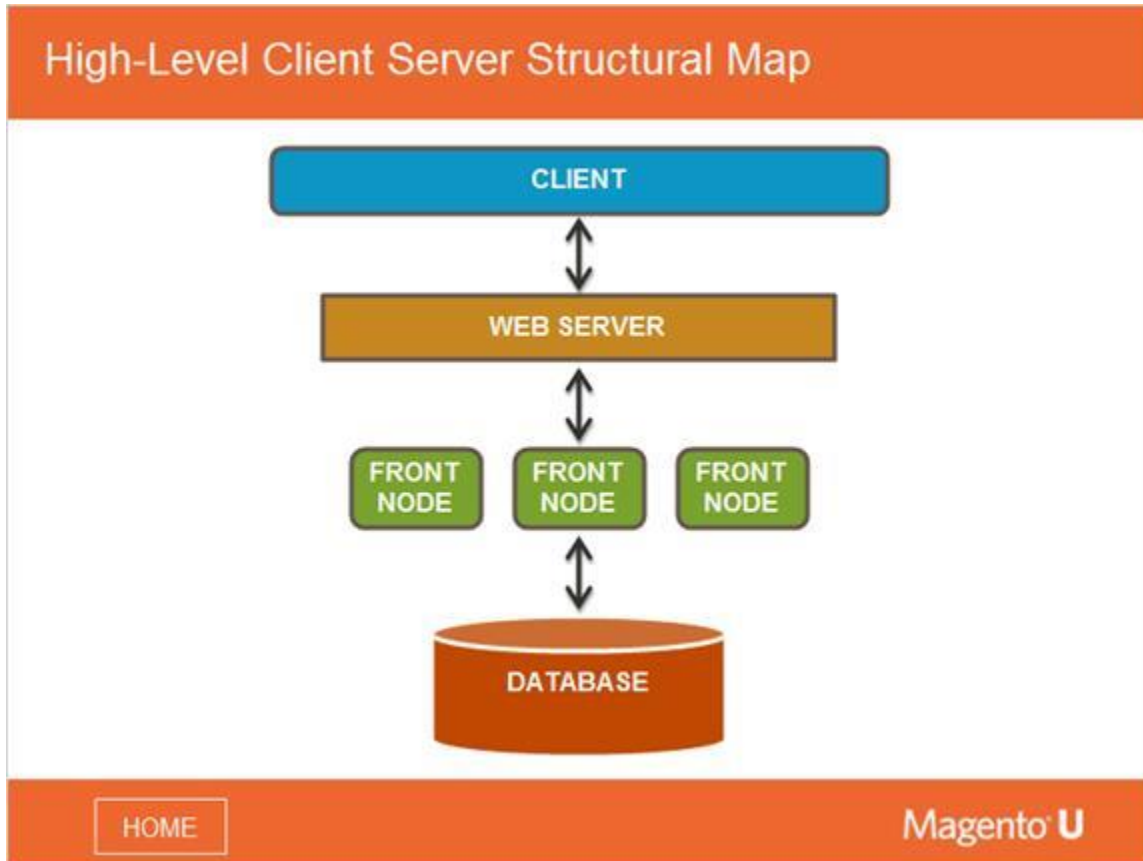
- Streamline the customization process.
- Update the technology stack.
- Improve performance and scalability.
- Reduce upgrade efforts and costs.
- Simplify integrations.
- Provide high quality tested code, testing resources, and documentation (and increase engagement with the Magento community).

## 4.5 Magento 2 Architecture

**Notes:**


Magento 2, like Magento 1, is a modular system but to a much higher degree. Magento 2 assigns greater independence to the modules – they function more as standalone units.

## 4.6 High-Level Client Server Structural Map

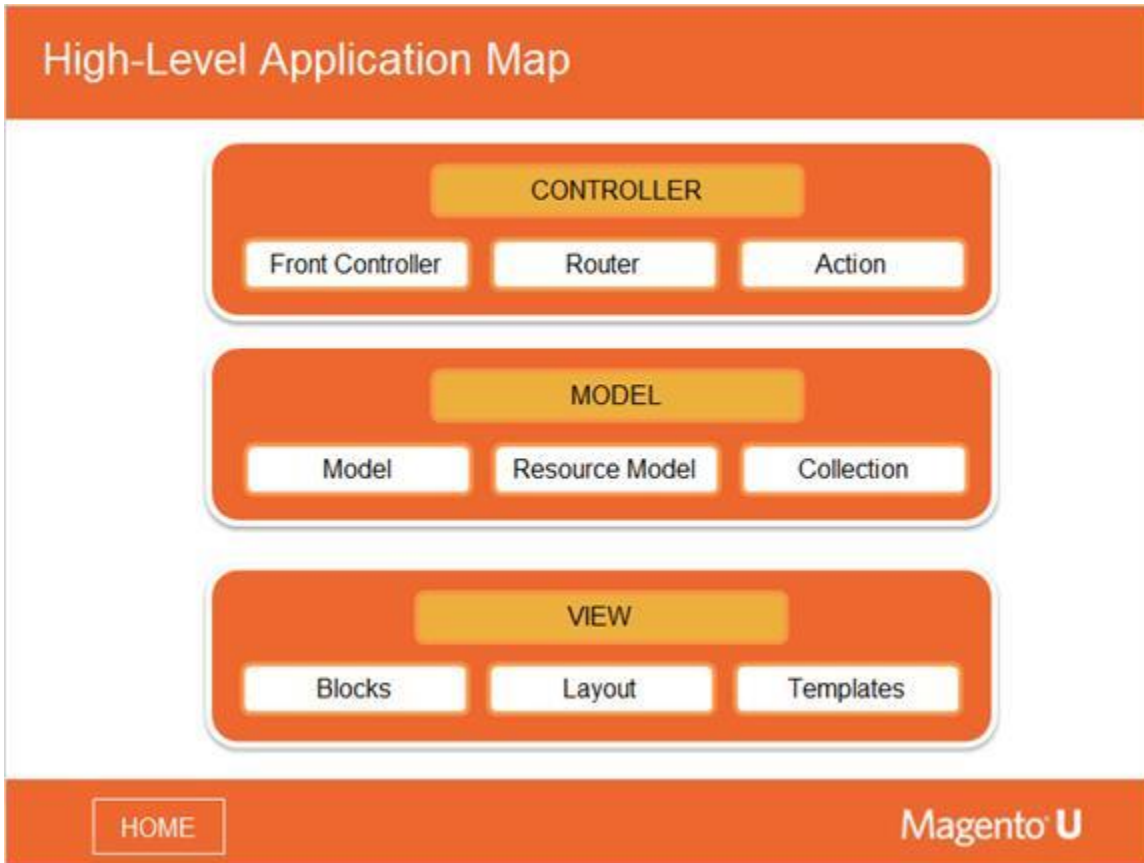


### Notes:

This structural map shows a standard client-server setup. A client sends a request to the web server or cloud, which then distributes the request among one or more front nodes. This depends on the website setup of the customer.

 Note that often the database and frontend are located in separate servers.

## 4.7 High-Level Application Map



### Notes:

This diagram presents a very high level application map. You are probably already familiar with the Model-View-Controller (MVC) concept. The model (or data layer) manipulates the data, the view layer renders the data, and the controller layer makes it all work together.

#### CONTROLLER

Magento has its own version of the MVC structure with respect to controllers. Unlike Magento 1, where one controller could process many requests, Magento 2 assigns a single controller to a single request. It is not possible to process multiple URLs in one file. The controller layer is composed of 3 systems - the front controller, the routing system, and other controllers.

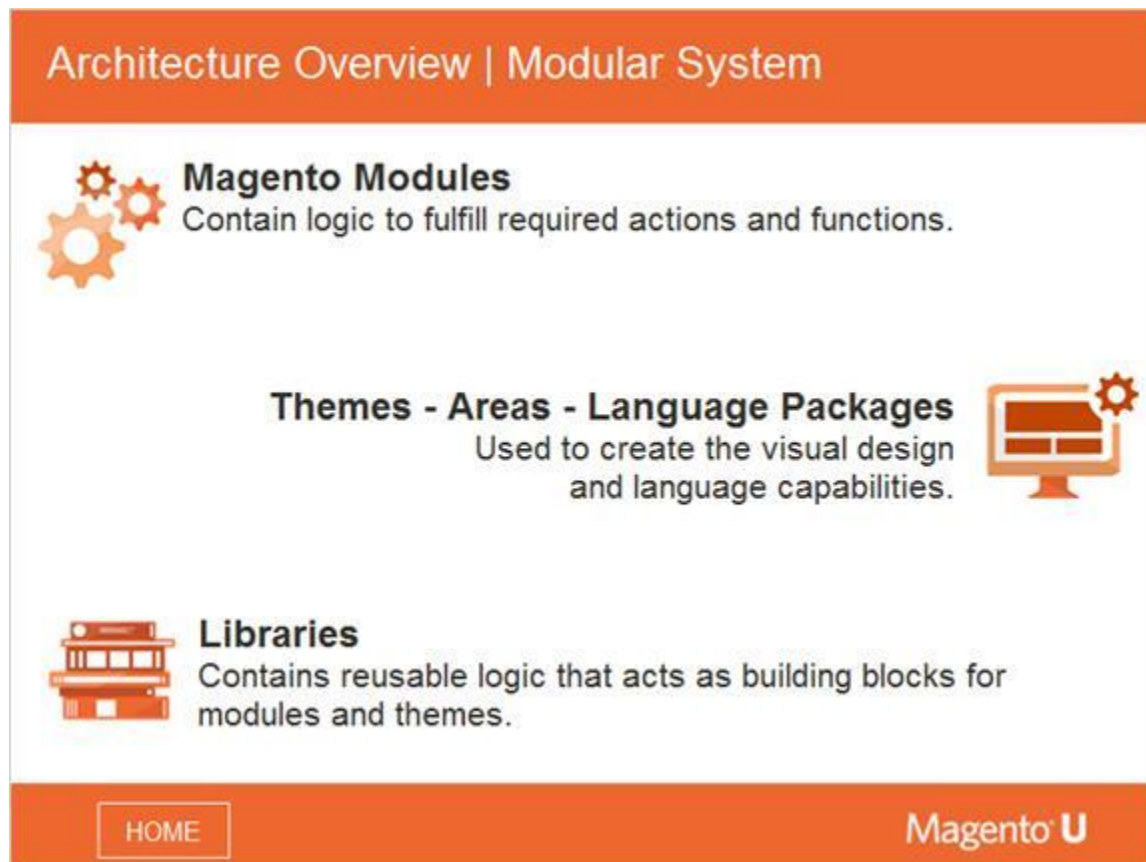
#### MODEL

The model is the part of the application that works with stored data. Magento 2 retains the resources model and collections that were in Magento 1, while adding an additional system of repositories and services.

#### VIEW

Within Magento 2, the view system and layout remains conceptually the same as with Magento 1, but the organization of the layout, templates, and blocks is different. Consequently, the approach for frontend development has changed dramatically.

### 4.8 Modular System



#### Notes:


The Magento framework provides core business logic and functionality, base classes, resource models, and data access capabilities. The fundamental concepts and rules for how the components of the website behave are defined within the Magento framework. The framework provides core components with base functionality that can then be inherited by custom components for a specific website or application.

The final behavior, look-and-feel, and capabilities of the website are determined by how the components are extended and customized. Modules and themes are the units of customization in Magento -- modules for business features, and themes for the user experience and look-and-feel. Both have a lifecycle allowing them to be installed, deleted, disabled, and so on.

The libraries contain reusable logic that acts as building blocks for modules and themes.

## 4.9 Areas

# Areas



## Areas

- Scope of configuration allows Magento to load only required config files
- Only the dependent config options for that area are loaded
- Typically have both behavior and view components

<b>Admin Panel</b> (index.php)	<b>Storefront</b> (index.php or pub/index.php)	<b>Six Areas Within Magento</b>  (entry points)
<b>Crontab</b> (pub/index.php)	<b>Install</b> (setup/index.php)	
<b>REST Web API</b> (index.php or pub/index.php)	<b>Web API SOAP</b> (index.php or pub/index.php)	

HOME
Magento U

### Notes:

Area is a scope in configuration that allows you to load configuration files and options. Within Magento 2, there are areas for frontend, admin and install, just as in Magento 1.

The purpose of areas within Magento is to increase efficiency by not requiring the loading of the entire configuration for every request.

For example, if you are invoking a REST web service, a corresponding area (such as /rest) loads code that answers only the REST call and not the code that generates HTML pages using layouts.

Each area can have completely different code on how to process URLs and requests. Magento 2 processes a URL request by first stripping off the base URL.

Typically, an area contains behavior and view components that operate separately. However, an area can have only one component -- for instance, the cron area, which has no view component. If your extension works in several areas, you should make sure it has separate behavior and view components for each area.



## 4.10 Libraries

### Libraries

- Consist of reusable logic that is commonly used across multiple applications, including application frameworks
- Do not provide independent business features; instead, they offer building blocks for modules and themes
- Are placed in the `/lib` folder and are PSR-0 compliant



[HOME](#)Magento U

### Notes:

Libraries play a big role in configuration and file management. Magento 2 places a special focus on libraries - the library component is much larger and better than in Magento 1. You can have the native implementation or substitute with your own implementation, which is a different approach from Magento 1. Magento 2 also uses interfaces much more than Magento 1.

Magento is a unique application where owners can sell products, process orders, and receive payments. To provide this functionality, Magento combines framework features like those in Symfony and Zend Framework with application features. This mix can sometimes make the system difficult to work with.

Magento 2 separates the framework from the application, which helps to alleviate this issue. The framework code and remaining code are stored in the code folder.

The most important library is `Magento/Framework`, located in `lib/internal/Magento/Framework`.

We will provide more detailed information about libraries later in the course, in Unit Three: Rendering.



## 4.11 Language Packs

### Language Packs

Language Packs

Magento can present the UI in different languages without modifying the actual application source code. Magento translates system messages, error messages, and labels for display in the UI.

By convention, the labels and system messages for the UI are expressed in English (en\_US) in the source code. To replace these phrases with a different language when the source code is interpreted, Magento has a layer of indirection.



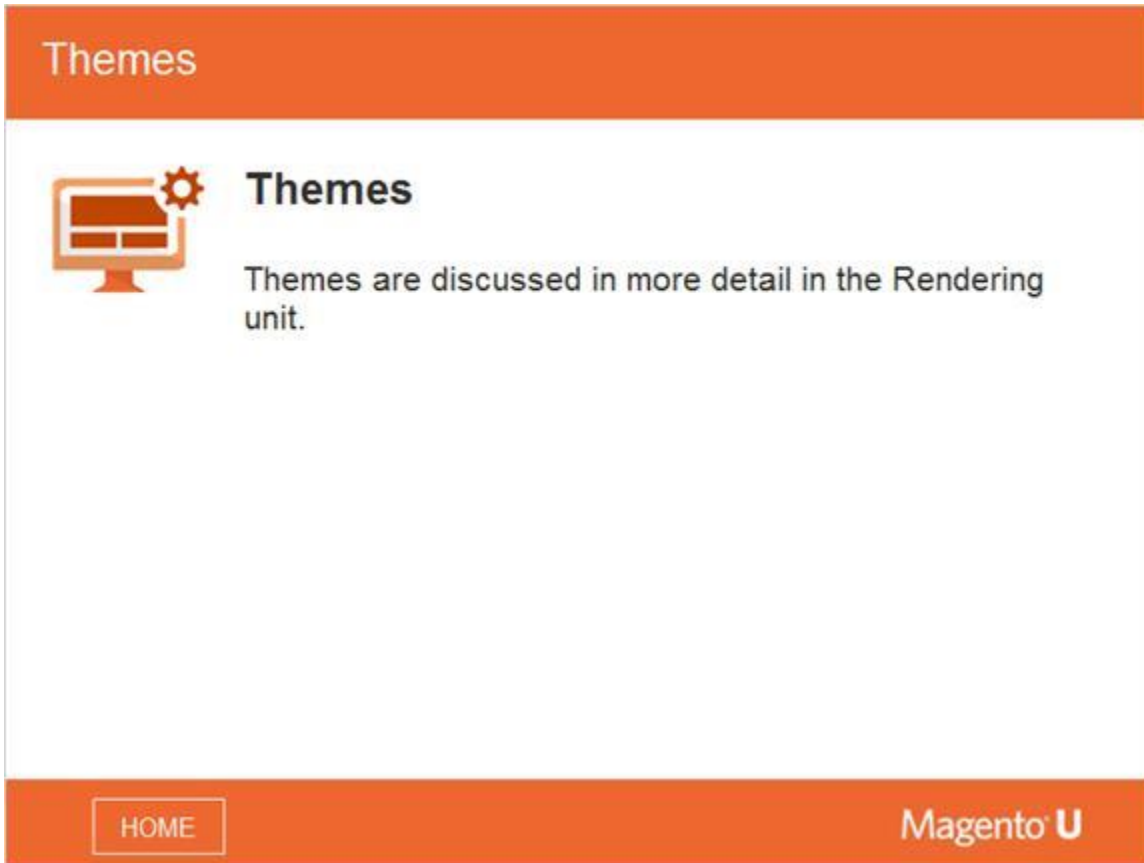
[HOME](#)Magento U

### Notes:

Magento 2 allows you to present a user interface in multiple languages without having to modify the application source code.

By default, all labels and system messages are expressed in English in the source code, but they can be replaced with another language when the code is by providing dictionary files that contain phrases from en\_US translated into a different language. The dictionary packages in other languages either ship with Magento out-of-the-box, or are provided by the community.

## 4.12 Themes



**Notes:**

Magento 2 introduces significant changes to how Magento handles themes.

Theming now primarily focuses on creating CSS files and incorporating responsive design. Also, backend tasks and frontend tasks have been separated. In this course, we will discuss the backend part of theming, which involves blocks, templates, and more.

## 4.13 Modules: Definition

Modules | Definition

**A module** is basically a package of code that encapsulates a particular business feature or set of features.

- A module is a logical group -- a directory containing blocks, controllers, helpers, models, and so on, related to the specific feature or a widget.
- Using a modular approach implies that every module encapsulates a feature and has minimum dependencies on other modules.

HOME Magento U

### Notes:

In Magento 2, a module is still a folder as in Magento 1.

Modules have been made more granular and better organized, in terms of content and interactions among modules, in facilitate scalability and performance. They are also smaller and more logical (for example, if you are looking for a checkout code, you only have to look in checkout and nowhere else).

A module provides specific product features by implementing new functionality or extending the functionality of other modules. Each module is designed to function independently, so the inclusion or exclusion of a particular module does not impact the functionality of other modules. This maximizes flexibility when customizing a site.

While modules primarily define new business features, or customizations to existing ones, they can also define a default user interface for those features, which are customizable by themes.

## 4.14 Modules: Location

### Modules | Location

**Modules:**

- Contain .php and .xml files (blocks, controllers, helpers, models, ...) related to a specific functionality
- Are located in the /app/code/ directory of a Magento installation, in a directory with the following PSR-0 compliant format:

```
/app/code/<vendor>/<module_name>
```

The folder contains all code and configuration related to a module, including the `etc/module.xml` file, which contains the name and version of the module, and any dependencies.

[HOME](#)Magento U

**Notes:**

Modules contain PHP and XML files that are related to a specific functionality.

Location example: the Customer module of Magento can be found at `/app/code/Magento/Customer`.


The location of these files will be covered in more detail in Unit Four: File Systems.

## 4.15 Modules: Naming a Module

### Modules | Naming a Module

A module should be named according to the `Vendor_Module` schema, where the:

- Namespace corresponds to a module's vendor
- Module corresponds to the name assigned to the module by the vendor

[HOME](#)


### Notes:

To name a module, follow the Magento standards on naming convention and module location within a file system. Magento 2 uses the same folder structure as with Magento 1, but the syntax is a little bit different.

Example: Naming Convention for Modules with Composite Names

To solve the difficulty of converting a module name with multiple capitalized first letters to a lowercase alias, Magento provides a class containing a corresponding array of modules with composite names. This is then used to generate a class name.

Example: Module Location Conventions

Code Base of Custom Module ..... `<root>/app/code/<Vendor>/<Module>`

Custom Theme Files ..... `<root>/app/design/<Module>/<theme>`

Libraries ..... `<root>/lib/<Vendor_Library>`

## 4.16 Modules: Declaring a Module

### Modules | Declaring a Module

A module declares itself in the file `module.xml`, located in  
`<root>/app/code/<Vendor>/<ModuleName>/etc/`

It must also be declared manually in `/app/etc/config.php`

To declare a module, the following information should be specified:

- The fully qualified name of a module (according to the naming rules)
- Dependency of a module on other modules, if any

[HOME](#)Magento U

### Notes:

The module declaration - configuration system has changed in Magento 2. Now if you want to declare a new module, you have to do it in the `module.xml` (not the `app/etc` folder, as with Magento 1).

Minimal declaration sample:

```
<config>
    <module name="Vendor_Module" schema_version="2.0.0">
    </module>
</config>
```

## 4.17 Modules: Module Dependencies

### Modules | Module Dependencies

In the Magento system, modules are partitioned into **logical** groups, each responsible for a separate feature. This implies:

- Multiple modules cannot be responsible for one feature
- One module cannot be responsible for multiple features
- A module declares explicit dependency (if any) on another module
- Any dependency on other components (ex: theme) must also be declared
- Removing or disabling a module does not result in disabling other modules

[HOME](#)Magento U

### Notes:

The concept of dependencies, or using and being dependent upon another module's features, is important in Magento. Dependency usually means that a module is loaded after another module. You define your module in the module node, and then define which modules depend on this module in the sequence node.

Modules can be dependent upon the following components:

- Other modules
  - PHP extensions
  - Libraries (either Magento Framework Library or third-party libraries)
- 🔗 **Best Practice:** When using Magento's modularity, you can lose historical information contained in a module if it is removed or disabled. We recommend storing such information before you remove or disable a module.



## 4.18 Declare Dependency with Other Modules

### Declare Dependency with Other Modules

```
<?xml version="1.0"?>
<!--
-->
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../../../../lib/internal/Magento/Framework/Module/etc/module.xsd">
    <module name="Magento_CatalogInventory"
schema_version="2.0.0">
        <sequence>
            <module name="Magento_Catalog"/>
        </sequence>
    </module>
</config>
```

[HOME](#)

Magento U

#### Notes:

This code example from the core shows that `Magento_CatalogInventory` depends on `Magento_Catalog`.



## 4.19 Module Dependencies Tasks

### Modules | Module Dependencies Tasks

There are three main steps for managing module dependencies:

- Name and declare the module (in the `module.xml` file)
- Declare any dependencies that the module has on other modules or components (in the `composer.json` file)
- (Optional) Define the desired load order of files (in the `module.xml` file)

[HOME](#)Magento U

### Notes:

If your app has external dependencies - for example, you need a 3rd party library for a module - you will need to use Composer.

A module's dependencies (that is, all of the components upon which the module is dependent) are listed in the module's `composer.json` file.

The load order of any dependencies on a module is declared using the `<sequence>` element in the `module.xml` file.

You use the `<sequence>` node to define the load order for modules. If your module depends upon other Magento modules, you can include that module in the `<sequence>`, so it will be loaded afterwards.

The `<sequence>` element is optional, and is used:

- Only when the order in which components are loaded /installed is important.
- Only for modules – no other component type is entered in the `<sequence>` section.

## 4.20 Modules: Types of Module Dependencies

### Modules | Types of Module Dependencies

There are two types of module dependencies in Magento 2:

- **Hard Dependency:** Implies that a module *cannot* function without the other modules on which it depends
- **Soft Dependency:** Implies that a module *can* function without the other modules on which it depends

If a module uses code from another module, it should declare the dependency explicitly.

[HOME](#)Magento U

### Notes:

Knowing the difference between hard and soft dependencies is important when you have to make configuration changes.

**Hard Dependencies:** The module cannot work if the module on which it is dependent is not installed. Examples of hard dependencies include:

- The module contains code that directly uses logic from another module (instances, class constants, static methods, public class properties, interfaces, and traits).
- The module contains strings that include class names, method names, class constants, class properties, interfaces, and traits from another module.
- The module de-serializes an object declared in another module.
- The module uses or modifies the database tables used by another module.

**Soft Dependencies:** The module is able to function without the other module(s) on which it depends. Examples of soft dependencies include:

- The module directly checks another module's availability.
- The module extends another module's configuration.
- The module extends another module's layout.

## 4.21 Reinforcement Exercise (1.4.1)

### Reinforcement Exercise (1.4.1)

*The solution to this exercise can be found in your Magento 2 Fundamentals Exercise Solutions guide.*

- Create a new module. What is the minimal code you need to add to accomplish this?
- Purposely make a mistake in the XML and test it to make sure it fails.
- Create a second module and make it dependent on the first. Then disable the first module and describe what happens.

[HOME](#)Magento U

### Notes:

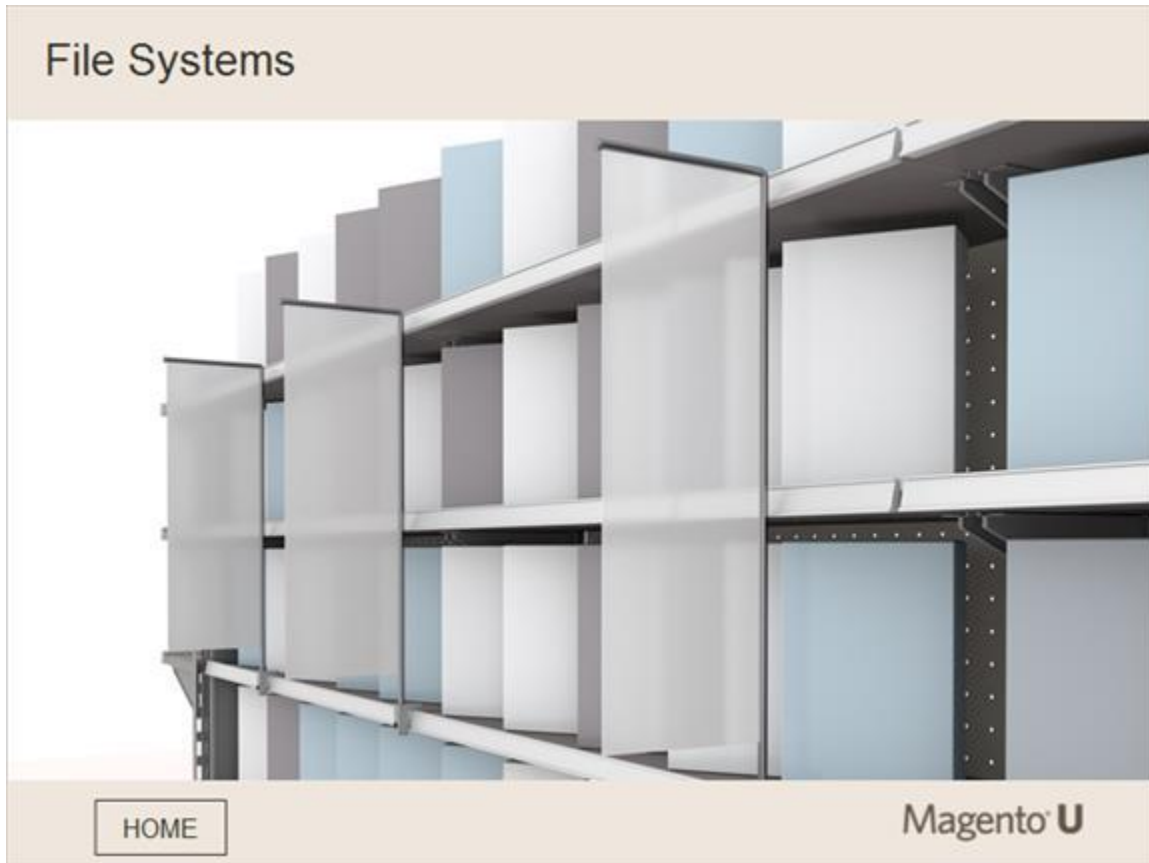
The purpose of the exercise is to learn how modules work. You will first create a module, and then break the code to see what happens when it fails.

Next, you will create a second module and make it dependent on the first. You will test the dependency by disabling the first module and seeing the effect this has on the second module.

## 5. File Systems

---

### 5.1 File Systems



**Notes:**

Let's look at the Magento 2 file system.

## 5.2 Module Topics

### Module Topics



**In this module, we will discuss...**

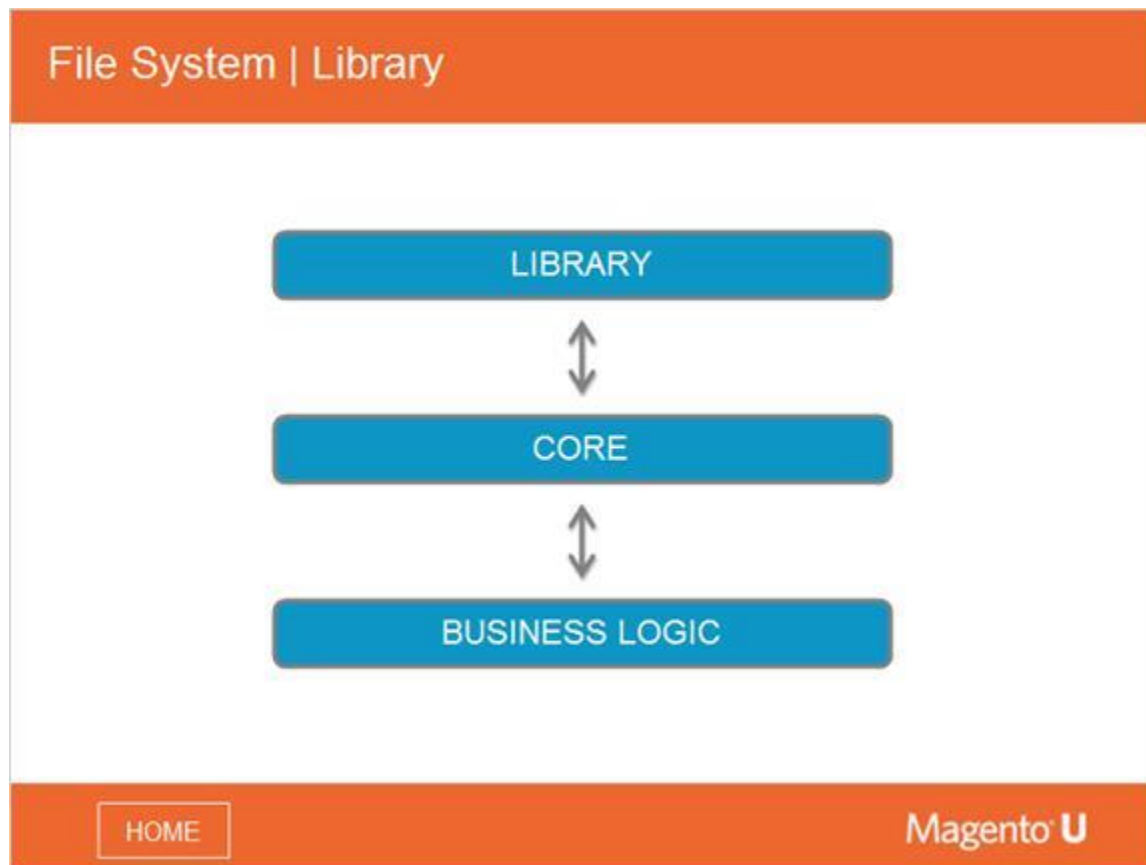
- Critical Files

[HOME](#)Magento U

**Notes:**

This module will discuss the file system structure within Magento 2 and where critical files are located.

## 5.3 File System: Library

**Notes:**

There are 3 different levels of code in Magento 2: library; core; and business logic. Each of these levels is part of the same file system.

The Magento 2 file system requirements are not as strict as in Magento 1. All classes follow the same rules, the same customization, and same principles. This means that in your module, you can create any folder you want and you will be able to access this class.

For example, in Magento 1, the `core/app` class and `config.php` were both under `model` - this happened because you had to declare one type of class.

In Magento 2, you don't need to declare anymore, it is all based on agreement. So, you can just decide you want a class and place it in any folder you like. You will be able to access the module or class without any declaration.

## 5.4 Code Demonstration: File Systems



### Notes:

Let's look at the root folder, `app/`. The `app` folder is where you have code and design folders. As there are no longer any code pools, the vendor name goes right after the `app/code/<vendor>` folder inside the list of modules.

Inside is the `index.php` file - the entry point for every HTTP request. The `setup` folder contains code for installing Magento.

`/pub` is a new folder introduced in Magento 2. It stands for "public access and production. The `/pub` folder becomes available in Production mode.

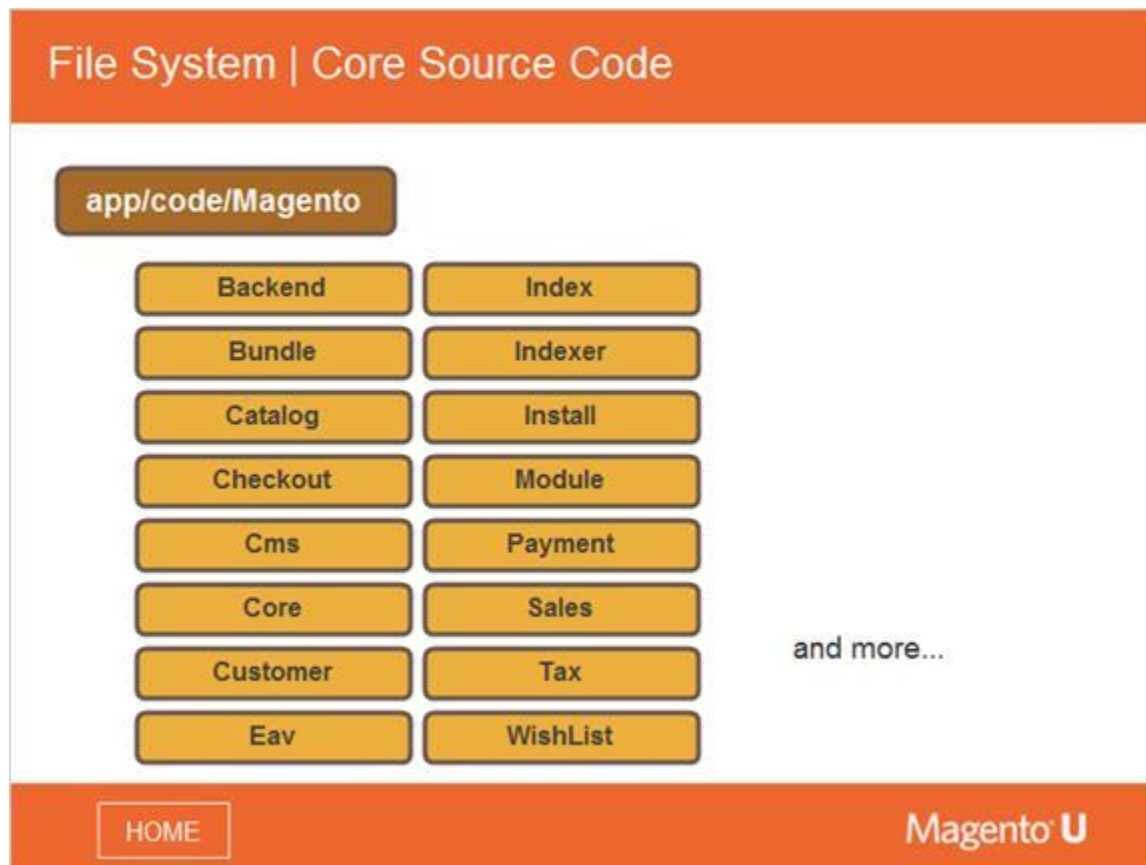
The `/lib` folder is where the internal Magento library is stored, replacing the `/lib Varian`.

The `/dev` folder contains the tools.

Every module has an `/etc` folder, and the first letter of the controller is now capitalized "C". All files follow the same concept.

In Magento 2, you don't have to declare the types of classes. You can create any folder you want and put any class in the folder. It is much easier to work without declarations and it decreases the probability of making mistakes. For example, when using a controller or creating a class, you may make small mistakes in the resource note or somewhere else and then nothing works. It can be a real headache to find out where the mistake is.

## 5.5 File System: Core Source Code

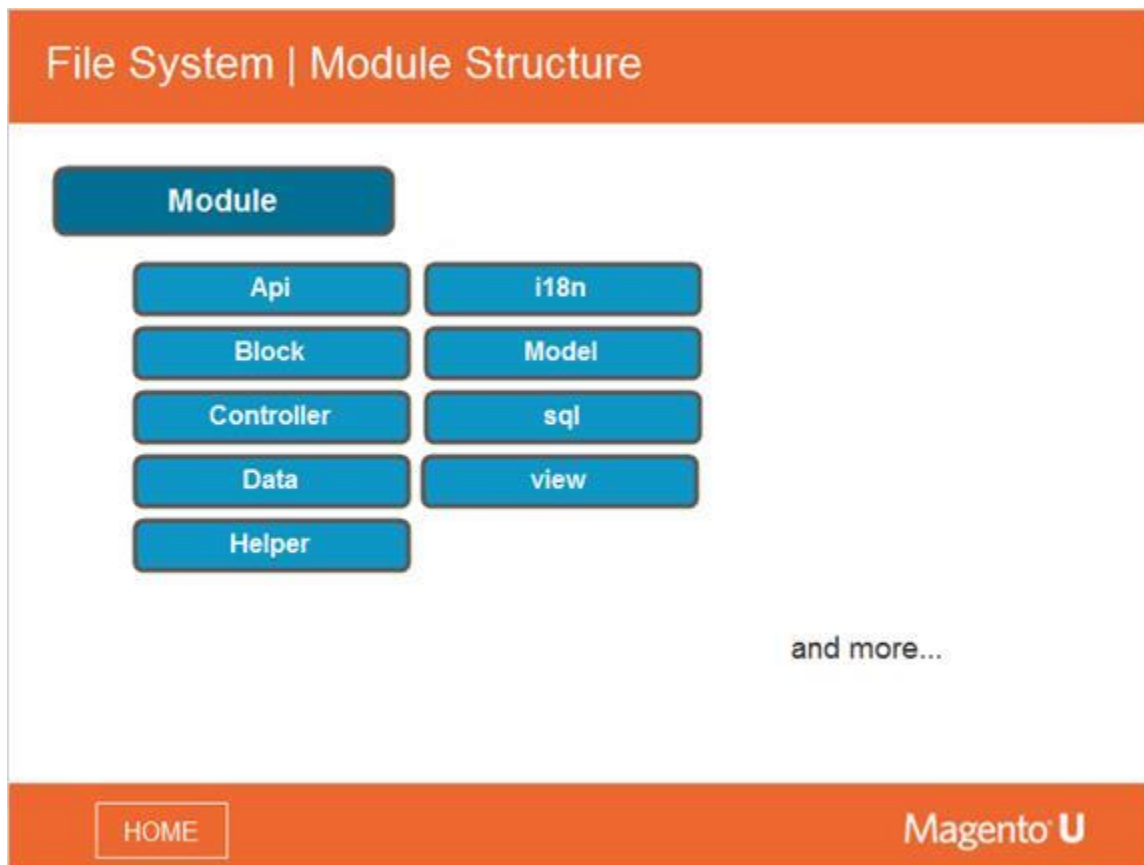


### Notes:

Magento code is organized in the following way. There is a folder, `app/code`, which is where all the php code is located. The code is packaged inside modules and modules belong to a vendor. In the native Magento installation, there is only one vendor - Magento. When you create custom code, you can create additional vendors. So, the folder structure is `app/code/<vendor>`, and inside this folder is the list of modules.



## 5.6 File System: Module Structure

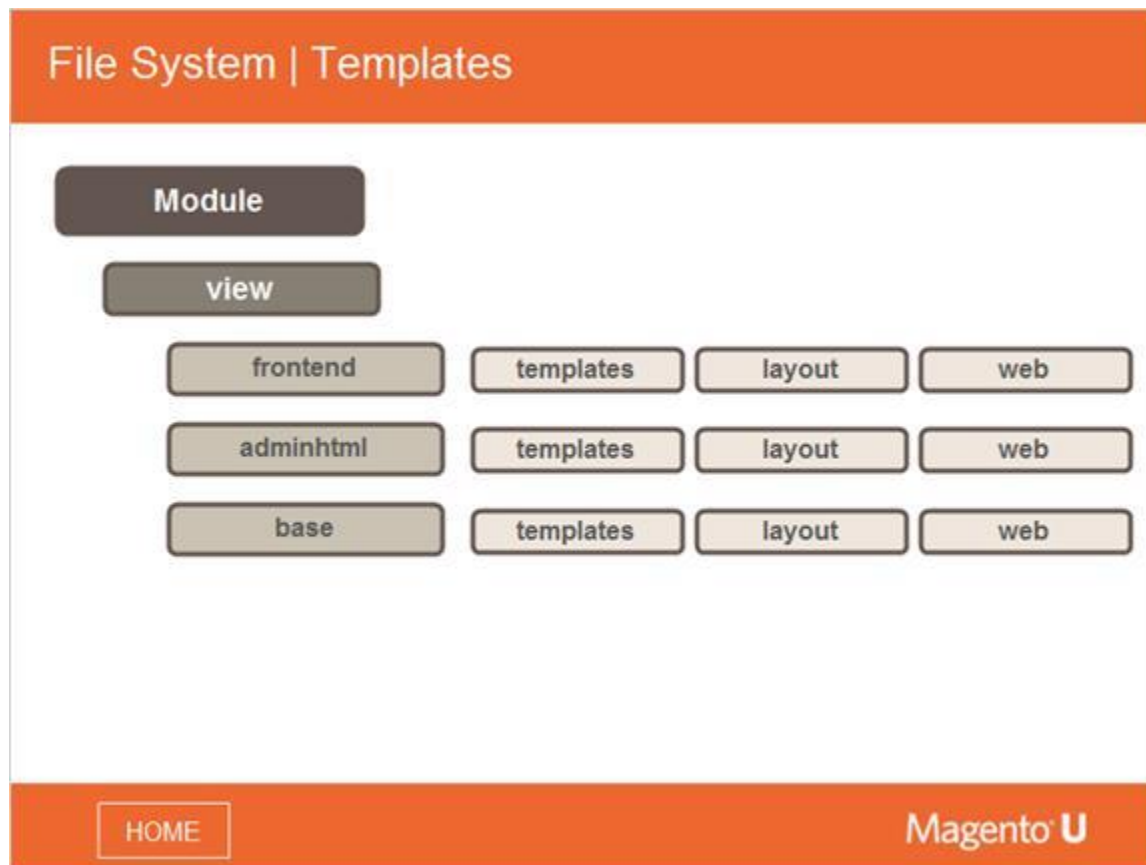


### Notes:

The diagram shows possible subfolders of a module folder. It is not as strict as in Magento 1, so a module might have any subfolder. The most popular are:

- Model: Where models and resource models are located
- Controller: For controllers
- Block: For blocks
- Helper: For helpers
- View: For layout xml files and templates.
- Sql: For upgrade scripts.

## 5.7 File System: Templates



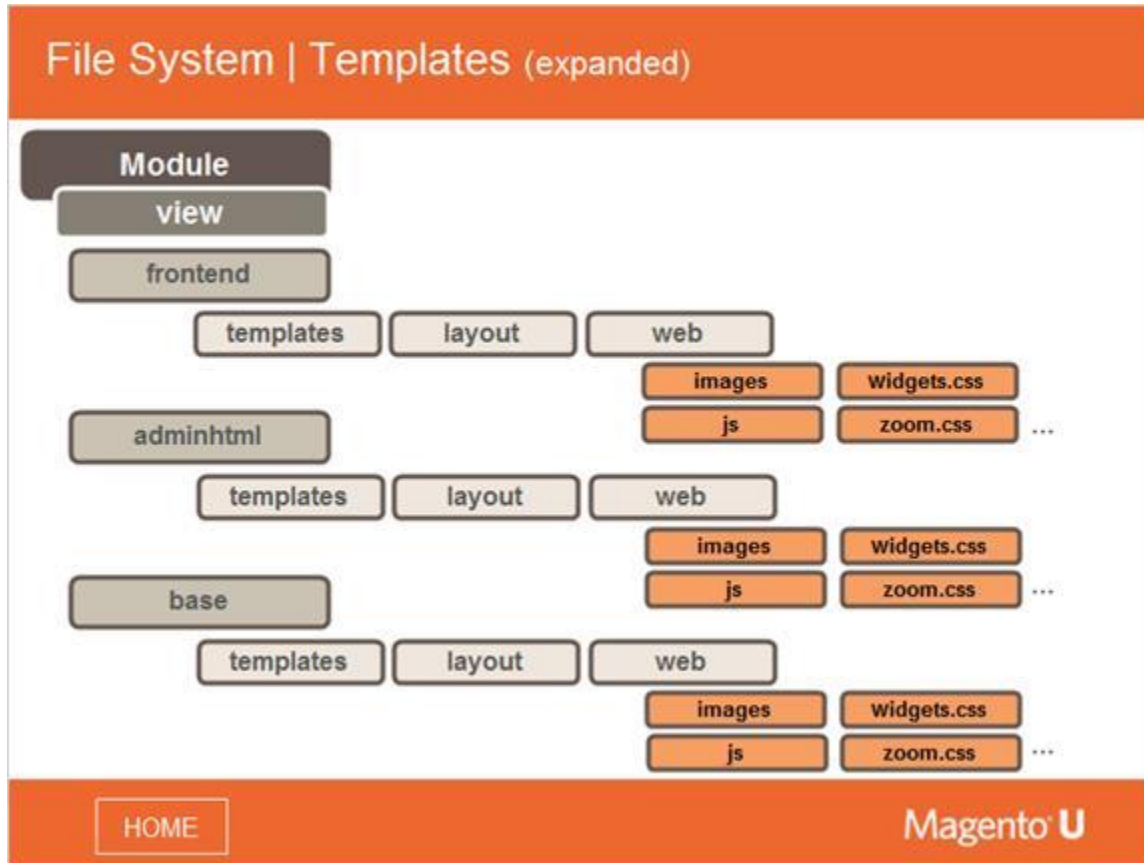
### Notes:

As we mentioned, modules are located in the app/code folder. All the native modules are located in the app/code/Magento folder.

In Magento 1, all templates were located in the design folder, separated from the php code, and grouped by themes and packages. In Magento 2, all templates and layout \*.xml files are located in the view folder. This makes modules more granular, as all associated files are packaged within one folder.

In Magento 1, templates were separated from the php code, so it was difficult to package. Now, in Magento 2, the two can be packaged together.

## 5.8 File System: Templates



### Notes:

Within the view folder are some folders that are frontend and adminhtml based. They represent areas with folders such as templates, layouts, and web. Template folders contain the phtml files, layout folders contain layout files, and web folders contain files that should be available in the web-like CSS and JavaScript.

From one point of view, this is a good thing because everything is in one module. We can package it all in one folder for distribution, making it very convenient.

However, we are no longer able to provide access from the server to our module to search for files, which can be an issue. The only folder that is *visible* from the web is “pub”, but modules are located in another folder “app”. So, if there is a file that has to be accessible through the web (like a css or image file), this can’t be done easily because the file’s location is restricted.

Magento provides a special mechanism for accessing them in dev mode, and special mechanisms to copy static files into the /pub folder for production mode.

## 5.9 Code Demonstration: Templates

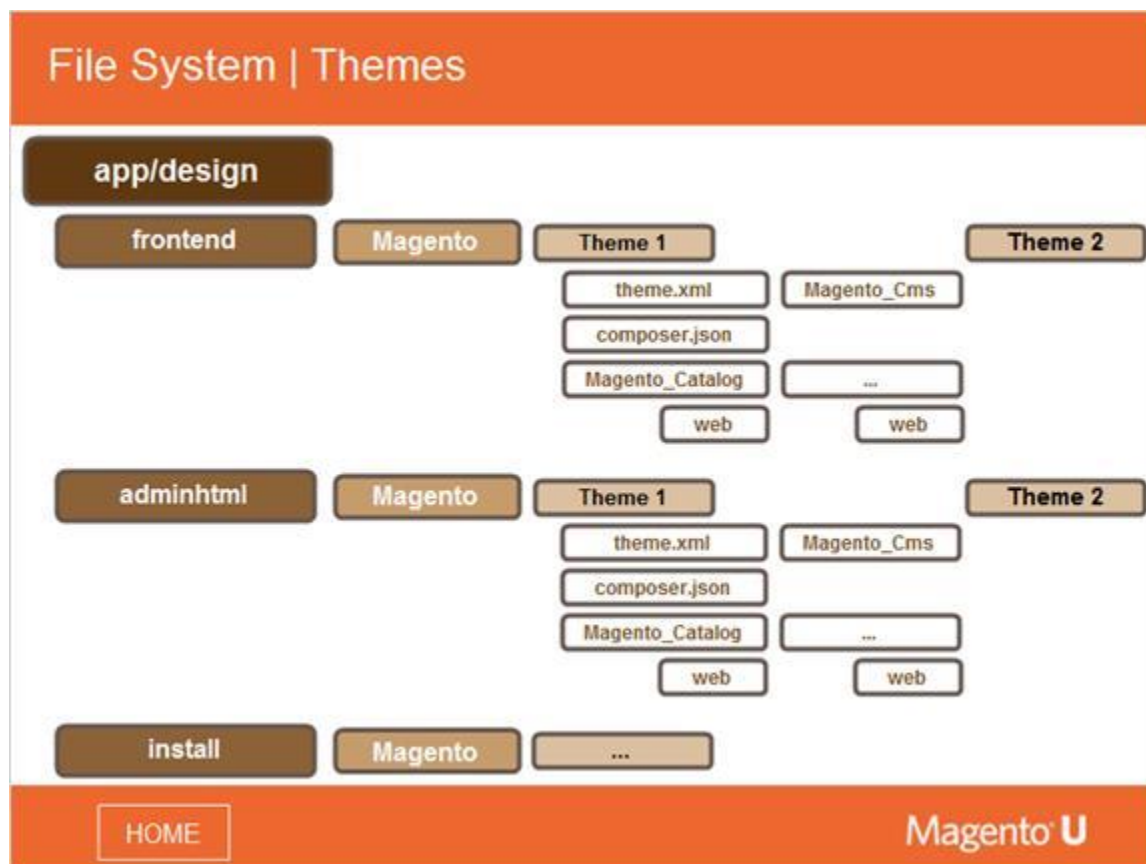


**Notes:**

Catalog View Example:

To customize the catalog view template, you need to create a new view folder in your module, and then physically set this template to the corresponding block (for example, using layout). This will be covered in more detail in the Rendering unit (unit three).

## 5.10 File System: Skin to Theme



### Notes:

Magento 2 introduces a new concept for the platform, themes, which contain not only the "skins" that were a part of Magento 1, but also templates and layout files. Themes can be thought of as a *superset* of Magento 1's Skins concept.

Themes in Magento 2 are specific to the look of the site, and no longer include templates. Themes inherit each other infinitely. You don't have to register a theme to use it. You create a new theme using xml (`theme.xml`). Theme folder might contain `<Vendor>_<Module>` folders (ex: `Magento_Cms`), which in turn can contain module-specific files.

For example, in `luma.xml`, you can see a parent and multiple levels of inheritance.

Magento 2 houses all the design elements in one folder located in the folder `app/design`.

## 6. Configuration

### 6.1 Configuration



#### Notes:

We now move on to discuss Magento 2 configuration.

Magento 2 has made substantial changes to how configuration was handled in Magento 1. Now, files are split by function and meaning, so you are left with a number of XML files composed of different file types.

For example, there is a `module.xml` for modules, a `config.xml` for generic settings like system level settings, an `event.xml` for events, and so on. These are smaller XML files, and each file is functional and follows a very strict syntax. This allows validation by XSD, which is helpful to developers when debugging XML files.

## 6.2 Module Topics

### Module Topics



**In this module, we will discuss...**

- Configuration Files
- Error Reporting Settings

[HOME](#)Magento U

**Notes:**

Within this module, we will address a number of key topics:

- Configuration files
- Error reporting settings

## 6.3 Configuration Files

### Configuration Files

In the Magento system, configuration files are simple to use, easy to troubleshoot, and validated automatically.

Predefined configuration files include: `config.php`, `config.xml`, `di.xml`, `events.xml`, `routes.xml`.

Files are loaded only when an application requests a specific configuration type.

Multiple extensions can declare configuration files that affect the same configuration type. These files are merged.

When multiple extensions have `events.xml` files, configuration is derived by merging all `events.xml` files from all the extensions.

[HOME](#)Magento U

### Notes:

In the Magento system, configuration files are simple to use, easy to troubleshoot, and validated automatically. Predefined configuration files include:

`config.php`:

- Contains database connection information
- Contains declaration of all modules

`config.xml`:

- Contains the configurations specified in the Stores > Configuration menu in the Admin panel for the default, website, and store scopes of your Magento instance.
- This menu is itself configured by the `system.xml` file, which declares the configuration keys for application configuration and defines how they are displayed in Stores > Configuration.

`di.xml`: Contains the configurations for dependency injection.

`events.xml`: Lists observers and the events to which they are subscribed.

`routes.xml`: Lists the routes and routers.

Magento loads different files and different areas separately. It will not load HTML on the frontend. It has different loaders for each file type, and does not use on demand loading.

 A complete list of configuration files can be found in the "Configuration Changes from Magento 1.x to 2.x" documentation.



## 6.4 Configuration Files: Storage

### Configuration Files | Storage

There are two main ways to store configuration files:  
in a database or in XML files.

System Configuration:

`core_config_data`

XML Configuration:

`di.xml`  
`config.xml`  
`acl.xml`  
`module.xml`  
`widget.xml`  
...

[HOME](#)

Magento U

### Notes:

In Magento 2, there are two main ways to store configuration: in a database (for merchants) and in xml files (for developers).

Storing the configuration in a database (`core_config_data`) allows a merchant to be able to change it through a generic interface. Therefore, database-level configuration options are usually ones that a merchant can understand and change.

xml file-level configuration options are usually more technical and should be changed by a developer.

The names of xml files generally make their function obvious, such as `widget.xml`, `events.xml`, `routes.xml`. Magento 2 has added some important new xml files as well - `module.xml` and `di.xml`.

## 6.5 Configuration Files: core\_config\_data

Configuration Files | core\_config\_data

core\_config\_data table structure:

Field	Type	Null	Key	Default	Extra
config_id	int(10) unsigned	NO	PRI	NULL	auto_increment
scope	varchar(8)	NO	MUL	default	
scope_id	int(11)	NO		0	
path	varchar(255)	NO		general	
value	text	YES		NULL	

5 rows in set (0.01 sec)

backend system  
configuration settings

HOME
Magento U

### Notes:

This diagram displays a core\_config\_data table structure, which feeds data into the interface a merchant uses to make changes to the site.

core\_config\_data is the same as in Magento 1. It includes scope and scope\_id fields.

Scope can be global, website or store. For a website, Scope\_id is treated as website\_id; for a store, as store\_id; for global, it is not applicable.

**core\_config\_data interface (Slide Layer)**

Configuration Files | core\_config\_data

core\_

Configuration

Scope: default config

Save Config

+ Field

+ config

+ scope

+ scope

+ path

+ value

+ 5 rows

Configuration

General

Web

Design

Currency Setup

Store Email Addresses

Contacts

Reports

Content Management

Advanced

Catalog

Inventory

MM, Shipping

WSS, Feeds

Email to a Friend

CUSTOMERS

Newsletter

Customer Configuration

Integrator

+ Country Options

+ State Options

+ Locale Options

+ Store Information

+ Single-Store Mode

RETURN TO THE DIAGRAM

## 6.6 Configuration Files: Custom Config Files

### Configuration Files | Custom Config Files

If you need custom options for a module, create a new .xml and .xsd file.

Example: Magento/Catalog/etc/  
product\_types.xml and product\_types.xsd

[HOME](#)
Magento U

### Notes:

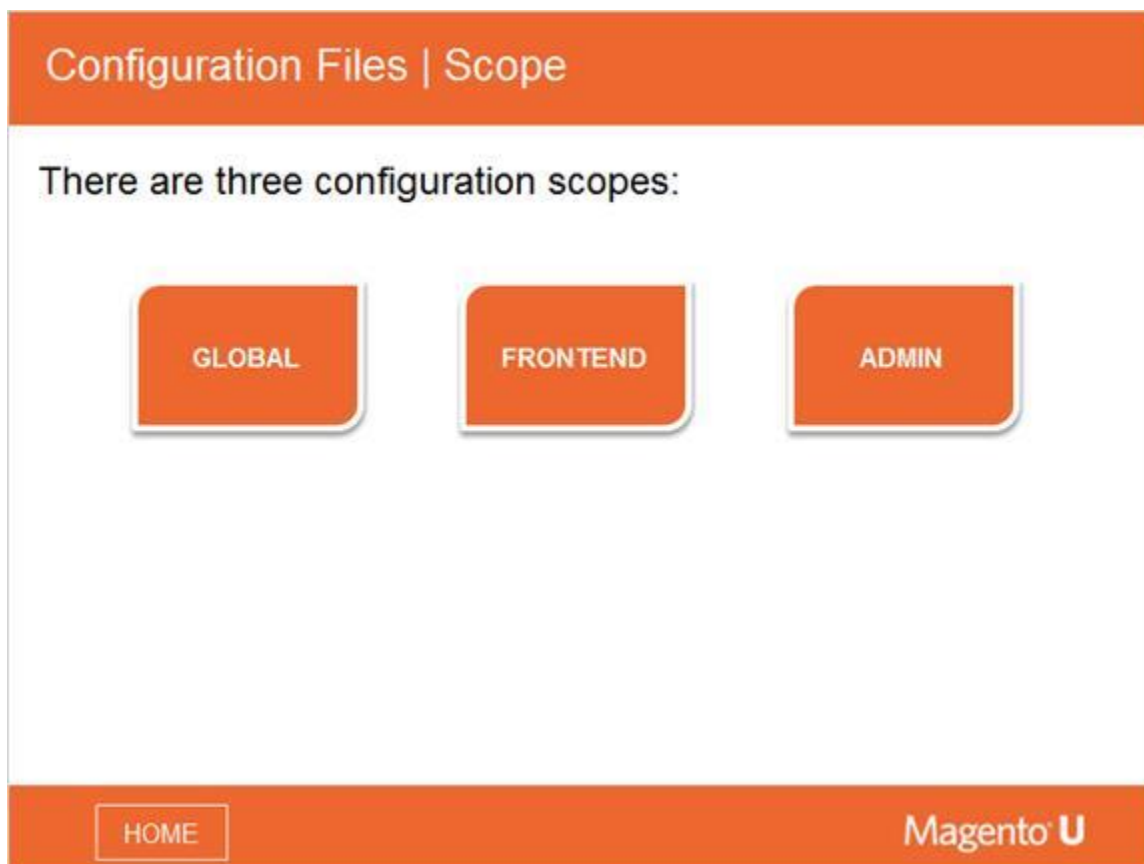
If you need custom options for a module, you have to create a new .xml and .xsd file.

Example: Magento/Catalog/etc/  
product\_types.xml and product\_types.xsd

### Module-specific configuration

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="../../../Catalog/etc/product_types.xsd">
    <type name="simple" label="Simple Product" modelInstance=
        "Magento\Catalog\Model\Product\Type\Simple" indexPriority="10" sortOrder="10">
        <customAttributes>
            <attribute name="refundable" value="true"/>
        </customAttributes>
    </type>
    <type name="virtual" label="Virtual Product" modelInstance=
        "Magento\Catalog\Model\Product\Type\Virtual" indexPriority="20" sortOrder="40">
        <customAttributes>
            <attribute name="is_real_product" value="false"/>
            <attribute name="refundable" value="false"/>
        </customAttributes>
    </type>
    <composableTypes>
        <type name="simple" />
        <type name="virtual" />
    </composableTypes>
</config>
```

## 6.7 Configuration Files: Scope



### Notes:

Within the Magento 2 xml configuration files, there are three scopes: global, frontend, and admin. Some configuration options are "scopable", others not. For example, an event can be either global or frontend or admin.

Everything defined in the <default> node overrides data from core\_config\_data.

Magento 2 provides corresponding folders for scope - a folder for frontend, and a folder for adminhtml. This means that a file like events.xml might appear in more than one folder. This separation into folders with Magento 2 is an advance over Magento 1, as now the system will load only the event needed. For frontend files, it will not go to the adminhtml folder, and vice versa. If you need files from both, they will be loaded and merged together.

- 📌 Note that some configuration options work only on the backend (admin), some on the frontend (frontend), and some on both (global).

## 6.8 Configuration Files: Load Order

### Configuration Files | Load Order

#### Load Order of Config Files

Once they are called, files are loaded in stages, according to the following groupings:

**Primary:** Loaded on bootstrap; include only config files needed for app start and installation-specific configuration

↓

**Global:** Include config files common across all app areas from all modules

↓

**Area-specific:** Files that apply to specific areas, such as adminhtml and the frontend

HOME Magento U

### Notes:

There are three steps in loading configuration files:

1. Primary, system-level files: These are files required for the application to start and installation-specific configuration (ex: `config.php`)
2. Secondary, global files: The global `di.xml` file (`app/etc/di.xml`) is loaded first, followed by the `di.xml` file from every module, along with other module configuration files (like `event.xml`).
3. Tertiary, area-specific files: These are configuration files for other specific areas, like `routes.xml`.

Some config files can be loaded at more than one stage -- for example, `di.xml` can be loaded at any of the stages; `config.xml` can be loaded as a primary or global file.

## 6.9 Configuration Files: Merging

### Configuration Files | Merging

#### Merging of Config Files

- Nodes in configuration files are merged based on their fully qualified XPaths.
- A special attribute is defined in the `$idAttributes` array and declared as an identifier.
- After two XML documents are merged, the resulting document contains all nodes from the original files.
- The second XML file either supplements or overwrites nodes in the first XML file.

[HOME](#)Magento U

#### Notes:

Let's talk about the merging of configuration files. In Magento, nodes are merged based on their XPaths and then identifier attributes. The assigned identifier must be unique for all nodes nested under the same parent node; otherwise, an error occurs.

## 6.10 Configuration Files: Validation

Configuration Files | Validation

Each config file is validated against a schema specific to its configuration type.

Example:

Events are configured in an `events.xml` file, and are validated during loading against the `events.xsd` schema file.

HOME

Magento U

### Notes:

Magento 2 uses schemas to make validation of configuration files faster and easier. The validation process differs significantly between Magento 2 and Magento 1. In Magento 2, in addition to every `*.xml` file, there is an `*.xsd` file that helps validate against a schema. The `*.xsd` files are located in the subfolders of `lib/internal/Magento/Framework` and other directory branches.

Example: Events are declared within the `events.xml` file, and then the configuration type is validated in the `events.xsd` file. The `events.xsd` file is located at: `lib/internal/Magento/Framework/Event/etc/events.xsd`



## 6.11 Magento\Config

### Magento\Config

All Magento 2 config files are processed by the library component `Magento\Config`.

The `Magento\Config` component loads, merges, and validates XML configuration files, and then converts them to proper array format.

During configuration loading, `Magento\Config` validates configuration files against schemas in XSD format.

Each XML configuration type has its own XSD schema.

[HOME](#)Magento U

### Notes:

`Magento\Config` processes the loading, merging, validation, and processing of the configurations. You can change the standard loading procedure by providing your own implementation of its interfaces. `Magento\Config` should be used to introduce a new configuration type.

`Magento\Config` provides the following interfaces for extension developers to manage configuration files:

- `\Magento\Config\DataInterface` retrieves the configuration data within a scope.
- `\Magento\Config\ScopeInterface` identifies current application scope and provides information about the scope's data.
- `\Magento\Config\FileResolverInterface` identifies the set of files to be read by `\Magento\Config\ReaderInterface`.
- `\Magento\Config\ReaderInterface` reads the configuration data from storage and selects the storage from which it reads.

The file system, database, and other storage merge configuration files according to the merging rules, and then validate the configuration files with the validation schema.

## 6.12 Magento\Config - Loading Infrastructure

### Magento\Config | Loading Infrastructure

**There are a group of files used to load an xml config:**

**Config...** class that is used to get access to the config values.

**Reader...** class that is used to read a file; usually only encapsulates the file name.

**SchemaLocator...** class that encapsulates path to the schema.

**Converter...** class that converts xml to array.

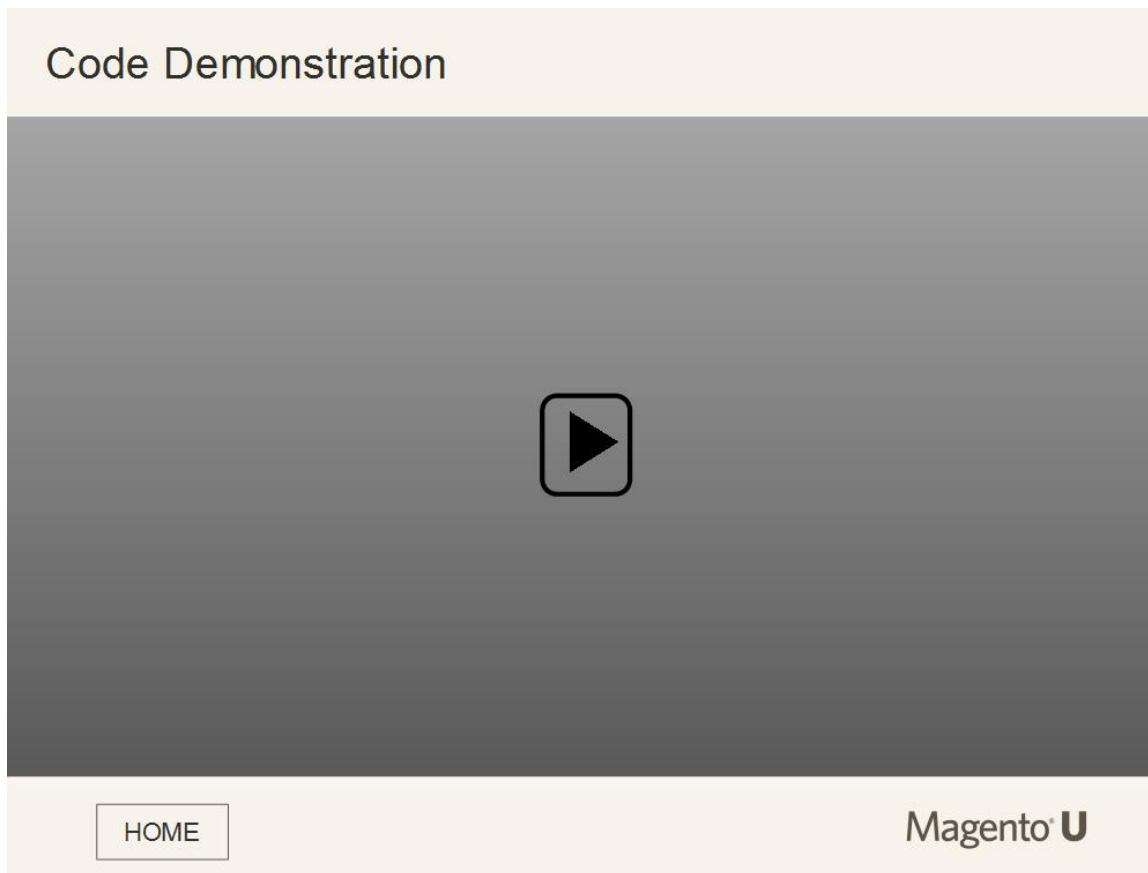
**xsd file...** schema file.

[HOME](#)Magento U

**Notes:**

A list of the files used to load an xml configuration.

## 6.13 Code Demonstration: events.xml



## 6.14 Validating a Configuration Type

### Validating Configuration XML


**Config files can be validated before\* and after the merging of files affecting the same configuration type.**

Provide two schemas for validating the configuration files (unless the validation rules for individual and merged files are identical):

- Schema for an individual file validation
- Schema for a merged file validation

New configuration files must be accompanied by XSD validation schemas. An XML configuration file and its XSD validation file must have the same name.

*\* optional*

[HOME](#)


### Notes:

In Magento, there are two possible schemas for validating configuration xml before and after merging config files. It could be the same schema, or two different schemas.

If you must use two xsd files for a single xml file, the names of the schemas should be recognizable and associated with the xml file.

To ensure validation of an xml file by an appropriate xsd file, you must specify the relative path to the xsd file in the xml file.

IDEs can validate your configuration files during development.

For example:

```
<config
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../../../lib/Magento/ObjectManager/etc/
config.xsd">
```

## 6.15 Check Your Understanding

### Check Your Understanding

Based on what you just learned about config files, if you wanted to create a new configuration type, which of these three things would you need to create?

- ☒ the .xml config file
- ☒ a loader
- ☒ the .xsd schema

[HOME](#)

Magento U

## 6.16 Check Your Understanding

### Check Your Understanding

If instead you wanted to extend an existing configuration type, which of these three things would you need to create?

- ☒ the .xml config file
- ☐ a loader
- ☐ the .xsd schema

[HOME](#)Magento U

The loader and schema already exist. The new .xml file will be merged with the other existing ones. The new .xml file you create will be automatically merged with the other .xml files.

## 6.17 Error Reporting Settings

### Error Reporting Settings

Magento uses the **strongest level of error reporting**, so that even a PHP notice will cause an exception.

It is very important, as a developer, not to suppress notices and to set a maximal level of error reporting.

Note that the way errors look is determined by the mode.

[HOME](#)Magento U

### Notes:

Error reporting settings in Magento 2 are similar to Magento 1. Magento 2 uses the strongest level of error reporting so that even a PHP notice will cause an exception.

- ☑ Note that the way errors look is determined by the mode. In Developer mode, errors will be displayed, allowing you to see the exceptions and refine your code. In other modes, there will only be a message that an error has occurred but the error itself will be recorded into the log file.



## 7. DI & Object Manager

---

### 7.1 DI & Object Manager



#### Notes:

We will now discuss the important topic of dependency injection, and its relation to Magento's object manager.

Dependency injection is a complex topic, and may be new to you. Do not worry if you do not immediately absorb all the concepts presented in this module.

We will use dependency injection throughout the course, in many of your exercises, and each time you work with the concept, you will build your knowledge and confidence.



## 7.2 Module Topics

### Module Topics



**In this module, we will discuss...**

- Dependency Injection
- Objects & Object Manager
- Cache Settings

[HOME](#)

Magento U

**Notes:**

Within this module, we will address a number of key topics:

- Dependency injection
- Objects & object manager
- Cache settings

## 7.3 Dependency Injection (DI) Pattern

### Dependency Injection (DI) Pattern

Dependency injection is a way to manage objects dependencies by setting objects to be used inside of a current object in the constructor.

This approach is called "Dependency injection".

[HOME](#)Magento U

### Notes:

Dependency injection means how you will get the objects. It creates parameters for the objects that you define on your constructor. How does it work?

You define your class and your class defines your constructor with the object that you need.

The dependency injection system is basically an object manager and the factory will create it for you, and everything is based on the configuration that we have seen before.

## 7.4 Dependency Injection

### Dependency Injection (DI) Overview

- All object dependencies are passed (injected) into an object instead of being pulled by the object from the environment.
- A dependency (coupling) implies that one component relies on another component to perform a function.
- A large amount of dependency limits code reuse and makes moving components to new projects difficult.

[HOME](#)Magento U

### Notes:

Dependency injection is configuration-based, validated by config.xsd.

Magento 2 uses dependency injection as an alternative to the Magento 1.x Mage class.

Assume you need a storeManager class in your Product class. You can declare StoreManagerInterface in the constructor of a product.

Now, using di.xml you have to define which class will be substituted for that interface. For that, you have two options:

- Define a preference and it will work globally (which means every other class that declares StoreManagerInterface will receive what you defined in preference).
- In your di.xml, assign a class for this particular object (Product), and it won't be global, just for Product.

## 7.5 Using Dependency Injection

### Using Dependency Injection

#### Constructor Injection

Constructor injection *must* be used for all optional and required service dependencies of an object. Service dependencies fulfill business functions of your object. Proxies must be used for expensive optional dependencies.

#### Method Injection

Method injection must be used for API parameters (the API objects that your object acts on).

[HOME](#)Magento U

#### Notes:

##### Constructor Injection

Constructor injection *must* be used for all optional and required service dependencies of an object. Service dependencies fulfill business functions of your object. Proxies must be used for expensive optional dependencies.

##### Method Injection

Method injection must be used for API parameters (the API objects that your object acts on).

**Constructor Injection Example:**

```
<?php
class Foo
{
    protected $_bar;

    public function __construct(Bar $bar)
    {
        $this->_bar = $bar;
    }

    public function execute()
    {
        //some code
        $this->_bar->execute();
        //some code
    }
}

$bar = new Bar();
$foo = new Foo($bar);
$foo->execute();
```

### Method Injection Example:

```
<?php
namespace Magento\Backend\Model\Menu;
class Builder
{
    /**
     * @param \Magento\Backend\Model\Menu\Item\Factory $menuItemFactory
     * @param \Magento\Backend\Model\Menu $menu
     */
    public function __construct(
        \Magento\Backend\Model\Menu\Item\Factory $menuItemFactory,
        // Service dependency
        \Magento\Backend\Model\Menu $menu // Service dependency
    ) {
        $this->_itemFactory = $menuItemFactory;
        $this->_menu = $menu;
    }
    public function processCommand(\Magento\Backend\Model\Menu\Builder\CommandAbstract $command)
    // API param
    {
        if (!isset($this->_commands[$command->getId()])) {
            $this->_commands[$command->getId()] = $command;
        } else {
            $this->_commands[$command->getId()]->chain($command);
        }
        return $this;
    }
}
```

## 7.6 Dependency Injection Object Manager Classes

### Dependency Injection Object Manager Classes

All classes related to an object manager are under `Magento\Framework`:

- `ObjectManagerInterface`
- `ObjectManager\ObjectManager`
- `App\ObjectManager`
- `App\ObjectManagerFactory`
- `App\ObjectManager\Factory\Dynamic\Developer`
- `App\ObjectManager\Factory\Dynamic\Production`

[HOME](#)Magento U

### Notes:

The following object manager classes are located under `Magento\Framework`:

- `ObjectManagerInterface`, which provides an interface.
- `ObjectManager\ObjectManager` and `App\ObjectManager`, which both provide an implementation.
- `ObjectManagerFactory`, a class that creates object manager.
- `App\ObjectManager\Factory\Dynamic\Developer` and `App\ObjectManager\Factory\Dynamic\Production`, two factories used by object manager to create an instance of a class.

## 7.7 Dependency Injection | Creating Objects

### Dependency Injection | Creating Objects

The class and method that are really creating the objects are:

Class: `Magento\Framework\ObjectManager\Factory\Dynamic\Developer` (in the developer mode)

Method: `create()`

[HOME](#)Magento U

**Notes:**

The class, `Magento\Framework\ObjectManager\Factory`, and the method, `create()`, are used in creating objects.



## 7.8 Object Manager


### Object Manager

**Object manager** is responsible for several functions:

- Creating objects.
- Implementing singleton pattern.
- Managing dependencies.
- Automatically instantiating parameters.

It defines:

- **Parameters:** Variables declared in the constructor signature.
- **Arguments:** Values passed to the constructor when the class instance is created.

[HOME](#)


### Notes:

Now let's talk about how Magento 2 instantiates objects. This involves a discussion of object management, dependence injection, and plugins, as they are all part of the same system. This is the biggest change in Magento 2, and resolves some of the issues encountered with instantiation of parameters. The new approach makes it much easier to make customizations.

In Magento 2, the object manager has replaced the `Mage` class. Object manager is the class that creates all objects and is responsible for instantiation. Generally, the creation of an object is one of the biggest problems in software development.

In Magento 1, instantiation was more or less centralized, and most of the classes were created through the config file. There are four generic patterns within Magento for working with objects: Abstract Factory, Factory method, Singleton and Builder. Each one applies in a different situation and all four of them are implemented in Magento 1. For all singletons, you can create another instance of that singleton. In Magento 1, it was registry - so Magento 1 created a class and put it in the registry. If you requested singletons, then it would go to the registry and retrieve them.

In Magento 2, the object manager class has two methods - `GET` and `CREATE`. `GET` will return a singleton object (shared instance) from the protected registry while `CREATE` will create a new instance. This is an important difference. In Magento 1, the class `Mage` was a static class and it was included at the beginning of the request flow, so you always had access. Now, it is no longer a `Mage` Class nor static. Generally, you won't call object manager. Instead, you will create the objects in another way. We will look at this a little later in the module.

Object Manager is concerned with a class's parameters and arguments.

- Arguments are injected into a class instance during its creation.
- Parameter names must correspond to constructor parameters of the configured class.

## 7.9 Object Manager: Shared Instances Concept

Object Manager | Shared Instances Concept

### Object Manager

- `get()`
- `create()`

The difference between `get()` and `create()` is that `get` returns a singleton, while `create` returns a new instance.

In other words, if you call `get()` in two different places, you will still receive the same object, but if you call `create()`, you will return different instances of the same class.

HOME
Magento U

### Notes:

If you declare your instances to be shared then it means they will act as singletons. Shared instances can still be modified but it becomes more difficult with Magento 2.

`get()` method:

```
public function get($type)
{
    $type = $this->_config->getPreference($type);
    if (!isset($this->_sharedInstances[$type])) {
        $this->_sharedInstances[$type] =
            $this->_factory->create($type);
    }
    return $this->_sharedInstances[$type];
}
```

`create()` method:

```
public function create($type, array $arguments = array())
{
    return $this->_factory->create($this->
        _config->getPreference($type), $arguments);
}
```

## 7.10 Object Manager Configuration

Object Manager Configuration

**Object manager** requires the following configuration in the `di.xml` file:

- Class definitions for retrieving class dependencies types and numbers.
- Instance configurations for retrieving how objects are instantiated and for defining their lifestyle.
- Abstraction-implementation mappings (interface preferences) for defining which implementation is to be used upon request to an interface.

HOME
Magento U

### Notes:

Magento 2 provides a new approach on how to work with objects. It not only encourages you to call object manager, but it also creates an object for you. You no longer register classes as with Magento 1 - there are no factory names as, for example, `catalog/product`. Now, you need to use the full class name when you call a class.

The creation of objects is such an important process that Magento 2 handles it automatically, via the object manager and class dependencies. Object manager creates the objects you need injected into your classes. It will also create arguments for your objects.

With Magento 2, a developer declares an interface in the constructor, and object manager automatically creates objects for it, so you don't have to call it directly. So, you create a class and you might require an interface in this class as a parameter in a constructor. You may not know what object you are getting because it is programmed according to the interface, without class definition. You use preferences to define what class will implement the interface.

This is a very modern approach, thinking in terms of interfaces and not implementations. You can create a customization that will make Magento return something as an implementation using *your* classes and not the *native* classes.

To view the interface preferences for the object manager, use the following files (depending on the level):

- `app/etc/di.xml`
- `<core module dir>/etc/<areaname>/di.xml`
- `<core module dir>/etc/di.xml`

To define your own preferences, you need to define your own `di.xml`:

- `<your module dir>/etc/<areaname>/di.xml`
- `<your module dir>/etc/di.xml`

**Constructor Method Signature Example:**

```

public function __construct(
    \Magento\Framework\Model\Context $context,
    \Magento\Framework\Registry $registry,
    \Magento\Framework\Api\ExtensionAttributesFactory $extensionFactory,
    AttributeValueFactory $customAttributeFactory,
    \Magento\Store\Model\StoreManagerInterface $storeManager,
    \Magento\Catalog\Api\ProductAttributeRepositoryInterface $metadataService,
    Product\Url $url,
    Product\Link $productLink,
    \Magento\Catalog\Model\Product\Configuration\Item\OptionFactory $itemOptionFactory,
    \Magento\CatalogInventory\Api\Data\StockItemInterfaceFactory $stockItemFactory,
    \Magento\Catalog\Model\Product\Option $catalogProductOption,
    \Magento\Catalog\Model\Product\Visibility $catalogProductVisibility,
    \Magento\Catalog\Model\Product\Attribute\Source\Status $catalogProductStatus,
    \Magento\Catalog\Model\Product\Media\Config $catalogProductMediaConfig,
    Product\Type $catalogProductType,
    \Magento\Framework\Module\Manager $moduleManager,
    \Magento\Catalog\Helper\Product $catalogProduct,
    Resource\Product $resource,
    Resource\Product\Collection $resourceCollection,
    \Magento\Framework\Data\CollectionFactory $collectionFactory,
    \Magento\Framework\Filesystem $filesystem,
    \Magento\Indexer\Model\IndexerRegistry $indexerRegistry,
    \Magento\Catalog\Model\Indexer\Product\Flat\Processor $productFlatIndexerProcessor,
    \Magento\Catalog\Model\Indexer\Product\Price\Processor $productPriceIndexerProcessor,
    \Magento\Catalog\Model\Indexer\Product\Eav\Processor $productEavIndexerProcessor,
    CategoryRepositoryInterface $categoryRepository,
    Product\Image\CacheFactory $imageCacheFactory,
    \Magento\Framework\Api\DataObjectHelper $dataObjectHelper,
    array $data = []
) {
    $this->metadataService = $metadataService;
    $this->_itemOptionFactory = $itemOptionFactory;
    $this->_stockItemFactory = $stockItemFactory;
    $this->_optionInstance = $catalogProductOption;
    $this->_catalogProductVisibility = $catalogProductVisibility;
    $this->_catalogProductStatus = $catalogProductStatus;
    $this->_catalogProductMediaConfig = $catalogProductMediaConfig;
    $this->_catalogProductType = $catalogProductType;
    $this->moduleManager = $moduleManager;
    $this->_catalogProduct = $catalogProduct;
    $this->_collectionFactory = $collectionFactory;
    $this->_urlModel = $url;
    $this->_linkInstance = $productLink;
    $this->_filesystem = $filesystem;
    $this->indexerRegistry = $indexerRegistry;
    $this->_productFlatIndexerProcessor = $productFlatIndexerProcessor;
    $this->_productPriceIndexerProcessor = $productPriceIndexerProcessor;
    $this->_productEavIndexerProcessor = $productEavIndexerProcessor;
    $this->categoryRepository = $categoryRepository;
    $this->imageCacheFactory = $imageCacheFactory;
    $this->dataObjectHelper = $dataObjectHelper;
    parent::__construct(
        $context,
        $registry,
        $extensionFactory,
        $customAttributeFactory,
        $storeManager,
        $resource,
        $resourceCollection,
        $data
    );
}

```

## 7.11 Object Manager Configuration: Preferences Example

### Object Manager Configuration | Preferences Example

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../../lib/internal/
    Magento/Framework/ObjectManager/etc/config.xsd">
    <preference for="Magento\Catalog\Api\Data\
        ProductInterface" type="Magento\Catalog\Model\Product" />
</config>
```

[HOME](#)Magento U

### Notes:

You can define preferences for interfaces, classes, and so on. Preferences define which classes will be instantiated for an interface in a constructor method. They are configured by the preference node.

Example: `Magento\Catalog\Api\Data\ProductInterface`

If you require this interface in your code, you will get an instance of this class because of this configuration. This configuration is located in the `di.xml`. The global `di.xml` (`app/etc/di.xml`) defines preferences for key Magento classes.

In your code, you can request the `http\response` interface. You will not request objects but rather interfaces. The global file `di.xml` contains preferences for key interfaces. You will look into this file periodically to check for preferences.



## 7.12 Object Manager Configuration: Argument Example

Object Manager Configuration | Argument Example

```
<type name="Magento\Catalog\Helper\Product">
  <arguments>
    <argument name="typeSwitcherLabel" xsi:type="string">
      Virtual</argument>
    <argument name="catalogSession" xsi:type="object">
      Magento\Catalog\Model\Session\Proxy</argument>
    <argument name="reindexPriceIndexerData" xsi:type="array">
      <item name="byDataResult" xsi:type="array">
        <item name="group_price_changed" xsi:type="string">
          group_price_changed</item>
        <item name="tier_price_changed" xsi:type="string">
          tier_price_changed</item>
      </item>
      <item name="byDataChange" xsi:type="array">
        ""
      </item>
    </argument>
    <argument name="productRepository" xsi:type="object">
      Magento\Catalog\Api\ProductRepositoryInterface\Proxy
    </argument>
  </arguments>
</type>
```

HOME
Magento U

### Notes:

There are different types of arguments. In this example, you can see several types: string, array and object. DI allows you to define which objects should come as an argument when a new instance is created.

So, we have a system where something is required (ex: an object or interface) in the parameters, without calling the object manager, and Magento delivers the object that implements that interface.

The question is what it will deliver? If a product interface is required, what implementation of that product interface will be delivered? The implementation may not seem to make sense in some cases - for example, when you need specific products with specific data, but you require a product object that may not be the exact object that you need.

How does Magento know what product I require? Some objects are not injectable. It is not a hard but rather soft separation. The system allows you to get the objects you want but for some objects, it does not make sense to have them generated by dependency injection.

You will have to get those by object manager, load and so on.

Another issue occurs when we want an object in a particular class. Object manager will create the object, but it also contains a constructor, which might have a huge system of dependencies that Magento has to handle. It is like a recursive system, with requirements, arguments, and so on.

Not only can you define preferences for interfaces, as shown earlier - you can also describe arguments you want included in your object. For example, the code on the slide shows an sample class that has an argument.


## 7.13 Code Demonstration: Object Manager Argument Example



### Notes:

Let's look at a code example, the class, `Magento\Catalog\Helper\Product`. This is what the constructor looks like. You can see that it requires many different classes and interfaces. It will get instances of this interface according to the preferences from `di.xml`.

The code states that for the argument `typeSwitcherLabel`, put `Virtual`. Therefore, when somebody creates a class of `Catalog\Helper\Product` in its constructor, it will generate classes according to the preferences and will put `Virtual` as a value for the `typeSwitcherLabel` parameter. This is a very important mechanism because it allows you to customize.

 Note that you can perform the same action in your module, which then redefines `di.xml`. You might open a class and see argument values that come from 'nowhere'. To identify where they actually come from, you will have to find the right `di.xml` and explore it. It might be that `di.xml` files from multiple modules are involved. You have to know where the values come from and you have to know the `di.xml` configuration. This is a very popular customization approach in Magento 2.

As another example, say one class has an array of routers as a parameter. Where does the array of routers come from? They come from configuration files that affect the parameters that are in that class and that add something to that array. This is the new way of customization. Arguments of some objects in your `di.xml` help you find what you want to add to that array.

In Magento 1, a router works by creating an event and then an observer. Different modules can add items to the array of parameters and objects, and you create `Mage::getModel()->setSomething()`. This is in contrast to Magento 2, where arguments can be passed based on configuration in `di.xml`.

## 7.14 Object Manager Configuration: Shared Argument Example

Object Manager Configuration | Shared Argument Example

```
<type name="Magento\Catalog\Model\Indexer\Product\Price\Processor">
  <arguments>
    <argument name="indexer" xsi:type="object"
      shared="false">Magento\Indexer\Model\IndexerInterface
    </argument>
  </arguments>
</type>
```

HOME
Magento U

### Notes:

A shared object is an analog of a singleton.

You may encounter situations where you have an indexer in a class, and you want that object in a new session every time. In some cases, you will need a new object, as it will not be shared. Share here refers to the parameter of an argument, which is false. It defines whether it is a singleton or not. Basically, such an approach raises new theoretical problems.

For example, it makes sense in such cases to inject. In Magento 1 classes instantiate each other directly whenever an instance is required. In Magento 2, this is no longer the case. Instead, all dependencies are defined in constructors as arguments. When there are 2000 lines of code, you don't know what will happen so instead of quoting directly for classes you have to do it the constructor.

This might create a few issues, however. For example, it makes sense to inject a system into a session's request, but it doesn't make sense to inject objects such as products, or to inject everything, as it raises the issue of dependency management.

Example:

A product controller and a Magento method create a list of products in object manager. There are situations where you have a class and the class has a constructor. You extend the class and need new objects. So should you extend the constructor? This is not always appropriate, and will not always work. In these cases, you will need to use object manager. If you extend classes that already have a constructor, you won't want to change the signature in a new object.



## 7.15 Object Manager Configuration

### Object Manager Configuration

**Object manager configurations can be specified at any of these levels:**

- Global across all of Magento (`app/etc/di.xml`)
- Entire module (`<your module directory>/etc/di.xml`)
- Area-specific\* configuration for a module  
(`<your module directory>/etc/<area>/di.xml`)

[HOME](#)

Magento U

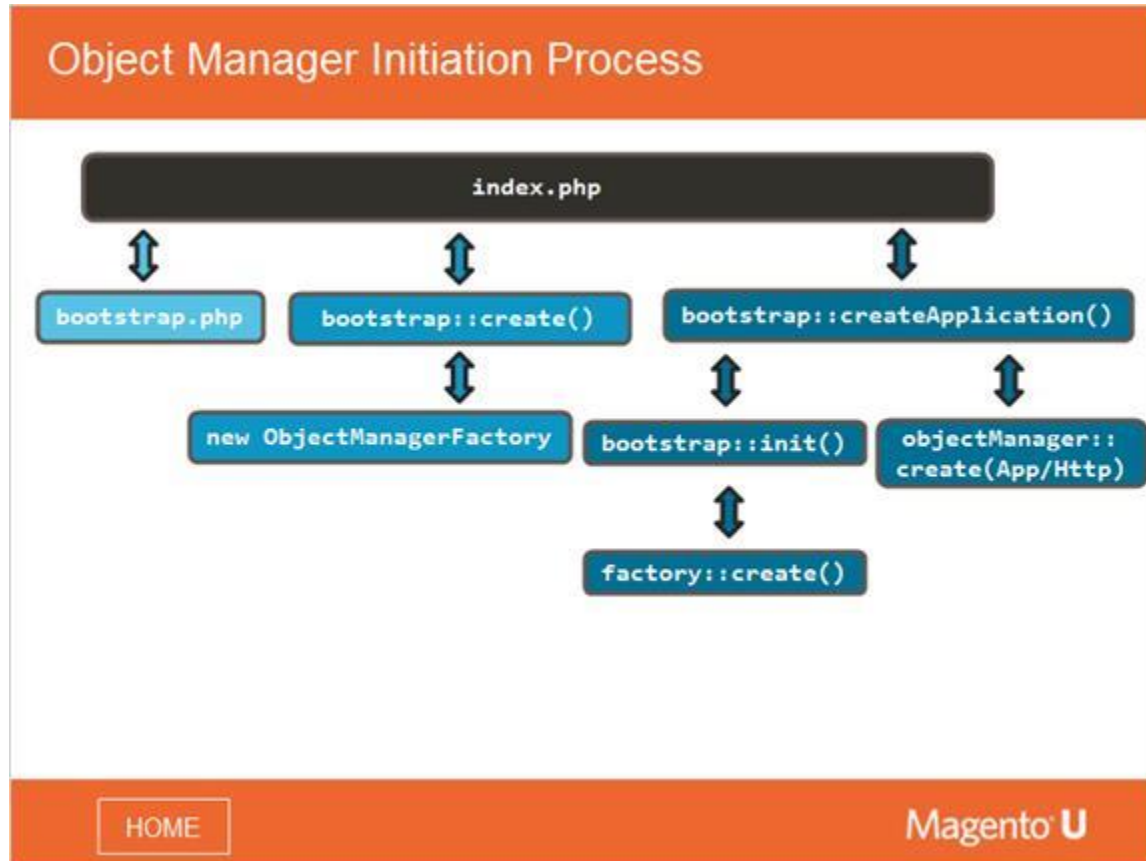
**Notes:**

\* Area-specific means specific to a module's area (example: frontend).

An area configuration overrides the global configuration.

The area-specific di.xml file is located in `modulename/etc/frontend/di.xml` or `modulename/etc/adminhtml/di.xml`.

## 7.16 Object Manager Initiation Process



### Notes:

The object manager factory creates object manager, *not* objects. If you look at the bootstrap, you will see that the factory creates an object manager. In turn, then, the object manager then creates a factory, which creates an object.

## 7.17 Objects: Injectable and Non-injectable

### Objects | Injectable and Non-injectable

**Injectable:**  
An object (typically a singleton) that can be instantiated by the object manager.

**Non-injectable:**  
An object that cannot be instantiated by the object manager. Typically, this object...

- ... has a transient lifestyle
- ... requires external input (user, database) to be properly created

Most models are *not* injectable (ex: `Magento\Catalog\Model\Product`)

[HOME](#)Magento U

### Notes:

We have briefly discussed injectables and non-injectables earlier.

Examples:

- Database objects are not injectable.
- System objects like singletons (shared instances) are injectable.

## 7.18 Objects: Injectable and Non-injectable Rules

### Objects | Injectable and Non-injectable Rules

**Injectables** can request other injectables in the constructor, but non-injectables *cannot*.

If the business function of an injectable object is to produce non-injectables, that injectable must ask for the factory in its constructor (because factories are injectables).

If the business function of an injectable object is to perform actions on non-injectables, it must receive the non-injectable as a method argument.

[HOME](#)Magento U

### Notes:

Only injectables can request other injectables in a constructor.

For an injectable object to produce a non-injectable object, it requires a factory in its constructor.

For an injectable object to perform actions on a non-injectable object, it has to receive the non-injectable object as a method argument.

## 7.19 Objects: Injectable and Non-injectable Rules 2

### Objects | Injectable and Non-Injectable Rules

You can create non-injectables in services with object factories or pass them in as method parameters.

Do not push injectables to non-injectables because it requires additional lookup during object unserialization.

[HOME](#)Magento U

**Notes:**

You can create non-injectables in services with object factories or pass them in as method parameters.

Do not push injectables to non-injectables because it requires additional lookup during object unserialization.

## 7.20 Class Definition and Definition Compiler Tool

### Class Definition and Definition Compiler Tool

**Class definitions** are read using reflection. The definition compiler tool provides a faster way to accomplish this than with PHP. The tool:

- Generates all required factories.
- Generates interceptors for all classes that have plugins declared in `di.xml`.
- Automatically compiles definitions for all modules and libraries.
- Compiles class inheritance implementation relations to increase performance of configuration inheritance operations.
- Compiles plugin definitions (the list of declared public methods).

[HOME](#)Magento U

### Notes:

Magento uses class constructor signatures to retrieve information about class dependencies, and define what dependencies to pass to an object.

The parameters specified for a class type are inherited by its descendant classes.



## 7.21 Class Definition andRunning the Definition Compiler Tool

### Running the Definition Compiler Tool

Running the compiler tool generates the following files and directories:


- **<your Magento install dir>/var/generation directory**

Contains all generated classes by Magento and modules. Code generation is used to create service classes (proxies, interceptors, factories, and builders).

- **<your Magento install dir>/var/di directory**

Contains the following:

- Definitions.php for compiled definitions.
- Plugins.php for declared public methods in plugin definitions.
- Relations.php for class inheritance implementation relations.
- Magento\Framework\ObjectManager\Definition\Compiled uses these PHP files.
- If you don't run the compiler tool and these files do not exist, the slower Magento\Framework\ObjectManager\Definition\Runtime is used instead.

[HOME](#)


### Notes:

Running the compiler tool generates the following files and directories:

- <your Magento install dir>/var/generation directory, which contains all generated classes by Magento and modules. Code generation is used to create service classes (proxies, interceptors, factories, and builders).
- <your Magento install dir>/var/di directory, which contains the following:
  - Definitions.php for compiled definitions
  - Plugins.php for declared public methods in plugin definitions
  - Relations.php for class inheritance implementation relations

These php files are used by Magento\Framework\ObjectManager\Definition\Compiled. If you don't run the compiler tool and if the preceding php files do not exist, the slower Magento\Framework\ObjectManager\Definition\Runtime is used instead.

More information about the Compiler Tool can be found at:<<https://wiki.magento.com/display/MAGE2DOC/Using+Dependency+Injection#CompilerTool>>

## 7.22 Definition Compiler Tool Best Practice

### Definition Compiler Tool | Best Practice

The definition compiler tool does not analyze auto-generated factory classes in files located in the <your Magento install dir>lib/internal/Magento directory. Do not use auto-generated factory classes at the library level - these classes must be created manually.

**Best Practice:**

Use the slower runtime object during development, and then the compiled code in production. This is much faster for production because it is all pre-compiled. It does not work well in development mode, because it is time-consuming, having to go to thousands of classes and read every constructor.

**BEST PRACTICE**

[HOME](#) **Magento U**

### Notes:

The definition compiler tool does not analyze auto-generated factory classes in files that are located in the <your Magento install dir>lib/internal/Magento directory. Do not use auto-generated factory classes at the library level - these classes must be created manually.

### 🔧 Best Practice:

Use the slower runtime object during development, and then the compiled code in production.

The definition compiler tool will go to all classes to read all the definitions and compile them together, and then the object manager will create an instance. It will not do reflection - it will just read the definitions.

This is much faster for production because it is all pre-compiled. It does not work well in development mode, because it is time consuming, having to go to thousands of classes and read every constructor.



## 7.23 Cache in Magento 2

**Notes:**

Caching in Magento 2 is similar to Magento 1. In Magento 2, you use the `Magento\Cache` library component.

## 7.24 Cache Configuration

### Cache Configuration

You should set the primary configuration settings for caching as follows:

- DI configuration files, to define the pre-configured cache settings (`app/etc/di.xml` or `app/etc/*/di.xml`).
- Deployment configuration files, to tweak the application during the deployment as necessary (`app/etc/local.xml`).

Magento also has a `Magento\Cache` library component for implementing Magento-specific caching.

[HOME](#)Magento U

**Notes:**

Configuring the cache involves setting up the cache frontend and cache backend. Therefore, you will need to specify the cache frontend configuration, attach the cache types to the cache frontend, and set up the cache backend for the cache frontend. Magento 2 uses the cache frontend for all caching operations. Use the `di.xml` file of a module to make the settings for the cache frontend. These settings should contain a unique identifier, so the cache frontend can be easily retrieved from the pool.

The custom settings will override the default settings for the parts of the application that use the corresponding cache frontend. If needed, you can avoid saving the new cache into storage using the `Magento\Cache\Core` class.

## 7.25 Cache Type

### Cache Type

Cache type groups the cached data based on its functional role.

Cache type facilitates the handling of the cache -- for example, to execute operations not on the whole cache, but on parts assigned to the same cache type.

You can manage cache types via the Cache Management page in the Admin panel, or via the programming interface in the runtime.

HOME
Magento U

### Notes:

Interception is an approach used to reduce conflicts among extensions by changing the behavior of the *same* class or method. An interceptor is technically a new generated class, which will call every plugin as well as the original method.

### Interception Example:

```
<?php
namespace Magento\Catalog\Model\Product\Type;
/**
 * Interceptor class for @see \Magento\Catalog\Model\Product\Type
 */
class Interceptor extends \Magento\Catalog\Model\Product\Type
```

Magento 2 includes code generation. Let's look at the process of creating a plugin for a class, as an example. When someone requires a class, it will not get the class instance but an interceptor of the class, a generated class. This class will include your plugin, either at the beginning or the end. If you put it in the middle it will execute the original class.

## 7.26 Cache Cleaning

### Cache Cleaning

There are two ways to clean the cache:

- From the backend
- By physically removing the cache files

In Linux, it is a command (`rm -rf var/cache/*`) running from the document root folder.

[HOME](#)Magento U

### Notes:

Cache cleaning in Magento 2 is similar to Magento 1. There are two options for cleaning: using set admin functionality, or by manually removing files.

It is preferable to clean cache via the backend because it thoroughly cleans the cache.

The second option is to manually remove the cache files individually. The second option is much easier and faster in developer mode.

## 7.27 Reinforcement Exercise (1.7.1)

### Reinforcement Exercise (1.7.1)

*The solution to this exercise can be found in your Magento 2 Fundamentals Exercise Solutions guide.*

In the empty module you created in Exercise 1.4.1, add custom configuration xml/xsd files.

[HOME](#)

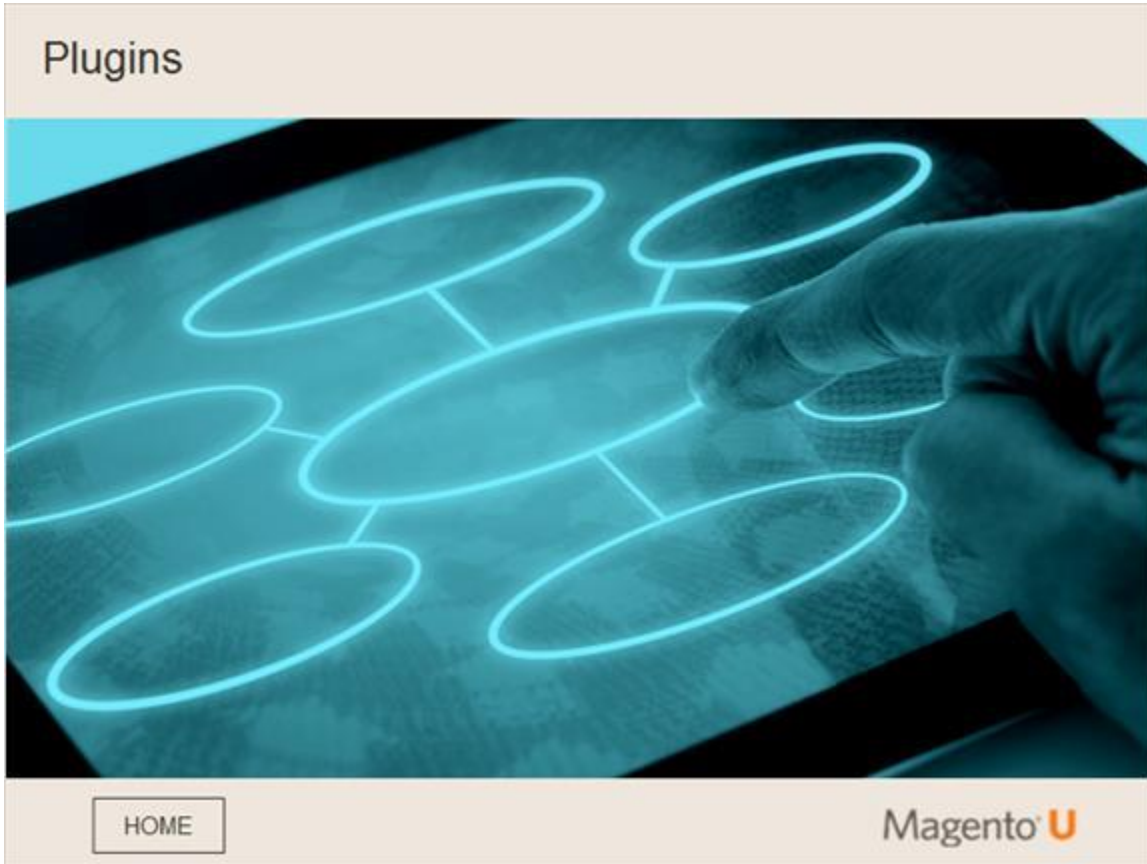
Magento U

**Notes:**

## 8. Plugins

---

### 8.1 Plugins



**Notes:**

This module discusses how best to use one of the powerful new features within Magento 2, plugins.

## 8.2 Module Topics

### Module Topics



**In this module, we will discuss...**

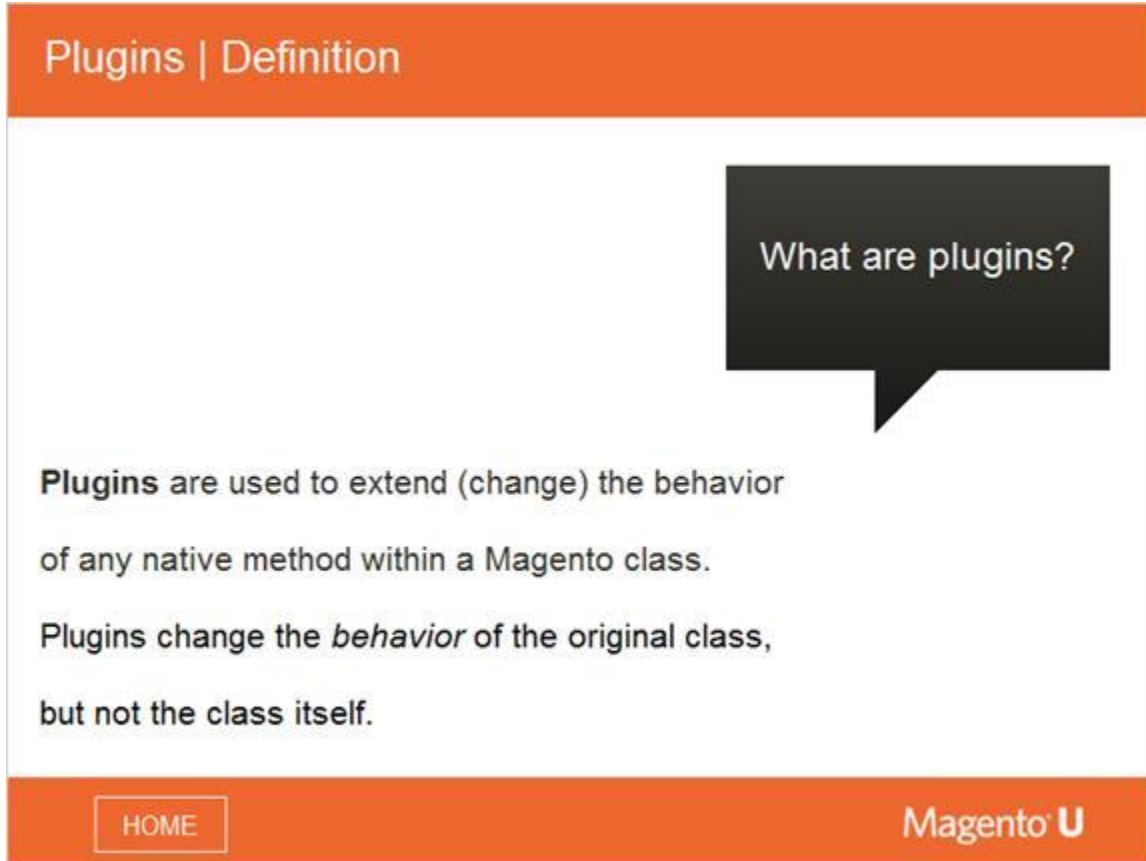
- Plugins
- Configuration Inheritance

[HOME](#)Magento U

**Notes:**

The topics covered within this module are plugins and configuration inheritance.

## 8.3 Plugins: Definition



Plugins | Definition

What are plugins?

**Plugins** are used to extend (change) the behavior of any native method within a Magento class.

Plugins change the *behavior* of the original class, but not the class itself.

HOME


Magento U

### Notes:

Plugins together with DI change the developer experience. The purpose is to make Magento customization simpler, and to decrease the probability of conflict.

Plugins are used to extend (change) the behavior of any native method within a Magento class. Native methods refer to those Magento class methods included with a native installation. The behavior of native methods can be changed by creating an extension. Extensions use the Plugin class to change the behavior of the methods. Plugins change the *behavior* of the original class, but not the class itself.

You cannot use plugins for final methods, final classes, and classes created without dependency injection. To ensure that plugins work correctly, you must follow declaration and naming rules.

 **Note:** Plugins allow you to modify a single method, while a preference allows you to change a whole class.



## 8.4 Plugins: Definition

### Plugins | Customizations

In Magento 2, customizations are accomplished through:

- Preferences - allow you to rewrite a class and work on the whole class level.
- Dependency injection - discussed in the last module.
- Plugins - allow you to customize a method. This method is basically rewrites and events at the class level.

You can begin your plugin after a method to modify its result.

With each plugin, you can put an observer before or after the method ends.

Plugins do not conflict with each other because they are executed one after another.

[HOME](#)Magento U

### Notes:

In Magento 1, there are two major ways to customize: events and rewrites. Events are good but you may not find them in the places you want every time, and if there are too many events, you may end up with one event triggering another and another. Rewrites is a powerful approach but it requires a solid understanding of what you are doing and how to fix possible issues.

In Magento 2, customizations are accomplished through: preferences, which allow you to rewrite a class and work on the whole class level; dependency injection; and plugins, which allow you to customize a method. This method is basically rewrites and events at the class level.


You can begin your plugin after a method to modify its result. With each plugin, you can put an observer before or after the method ends. Plugins do not conflict with each other because they are executed one after another.

## 8.5 Plugins: Interception

### Plugins | Interception

Interception is an approach used to reduce conflicts among extensions by changing the behavior of the *same* class or method.

An interceptor is technically a new generated class, which will call every plugin as well as the original method.

[HOME](#)


### Notes:

In Magento 1, there were different cache types, cache groups, configuration caches, and so on. In Magento 2, there are even more cache types - for example, the definition of caches.

To create a new cache type:

```
<?php

class %Vendor%_%Module%_Model_Cache_Type extends \Magento\Cache\Frontend\Decorator\TagScope
{
    public function __construct(\Magento\App\Cache\Type\FrontendPool
                                $cacheFrontendPool)
    {
        parent::__construct($cacheFrontendPool->get('%cache_type_id%'),
                            '%cache_type_tag%');
    }
}
```

You must specify the following parameters:

- Vendor\_Module defines the name of a module that uses a cache type. A module can use several cache types, and a cache type can be used in several modules.
- %cache\_type\_id% defines a unique identifier for cache type.
- %cache\_type\_tag% defines a unique tag to be used in cache-type scoping.

You must then configure the grid on the Cache Management page, or using the programming interface at runtime.

## 8.6 Declaring a Plugin

### Declaring a Plugin

You declare a plugin for an object in the `di.xml` file for a module.

```
<config>
    <type name="{ObservedType}">
        <plugin name="{pluginName}"
            type="{PluginClassName}"
            sortOrder="1"
            disabled="true"/>
    </type>
</config>
```

[HOME](#)

Magento U

#### Notes:

You must specify the following elements when declaring a plugin:

- **Type name:** A class, interface, or virtual type that the plugin observes.
- **Plugin name:** An arbitrary name that identifies the plugin; used to merge the configurations for the plugin.
- **Plugin type:** The name of a plugin class or its virtual type; uses the naming convention `<ModuleName>\Plugin`.
- **Plugin sort order:** The order in which plugins that call the same method are run.
- **Plugin disabled:** Set to true to disable a plugin.

Here is an example of a plugin. The syntax is very simple. The way it works: you define a plugin - you create a class within another class, then inside that class you can define which methods write the plugin.

## 8.7 Before-Listener Method

### Plugin Example | Before-Listener Method

To change arguments of an original method, or add some behavior before the method is called, use the before-listener method. Ex:

```
namespace My\Module\Model\Product;
class Plugin
{
    public function beforeSetName(\Magento\Catalog\Model\Product $subject, $name)
    {
        return array('(' . $name . ')');
    }
}
```

[HOME](#)

Magento U

**Notes:**

If you need to change the *arguments* of an original method, or add some behavior *before* the method is called, you should use the before-listener method.

A code example is presented on the slide.

## 8.8 After-Listener Method

### Plugin Example | After-Listener Method

To change values returned by an original method, or add behavior after an original method is called, use the after-listener method. The after prefix should be added to the name of an original method. Ex:

```
namespace My\Module\Model\Product;
class Plugin
{
    public function afterGetName(\Magento\Catalog\Model\Product $subject,
    $result)
    {
        return '|' . $result . '|';
    }
}
```

[HOME](#)

Magento U

**Notes:**

If you need to change the *values* returned by an original method, or add some behavior *after* an original method is called, you should use the after-listener method.

A code example is presented on the slide.

## 8.9 Around-Listener Method

### Plugin Example | Around-Listener Method

To change both the arguments & returned values of an original method, or add behavior before / after the method is called, use the around-listener method. The `around` prefix should be added to the name of an original method. Ex:

```
namespace My\Module\Model\Product;

class Plugin
{
    public function aroundSave(\Magento\Catalog\Model\Product $subject, \Closure $proceed)
    {
        $this->doSmthBeforeProductIsSaved();
        $returnValue = $proceed();
        if ($returnValue) {
            $this->postProductToFacebook();
        }
        return $returnValue;
    }
}
```

[HOME](#)

Magento U

#### Notes:

If you need to change *both* the arguments and returned values of an original method, or add some behavior *before or after* the method is called, you should use the around-listener method.

The around-listener method will receive two parameters (\$subject and \$proceed) followed by the arguments belonging to an original method:

- \$subject parameter will provide an access to all public methods of the original class.
- \$proceed parameter will call the next plugin or method.

Any further method arguments will be passed to the around plugin methods after the \$subject and the \$proceed arguments. They have to be passed on to the next plugin method when calling \$proceed().

A code example is presented on the slide.

## 8.10 Prioritizing Plugins

### Prioritizing Plugins

Several conditions influence how plugins apply to the same class/interface:

- Whether a listener method in a plugin should apply before, after, or around an original method
- The sort order of the plugin: a parameter defines the order in which plugins that use the same type of listener & call the same method are run

[HOME](#)
Magento U

### Notes:

Use one or more of the following methods to extend/modify an original method's behavior with the interception functionality:

- Change the arguments of an original method through the before-listener.
- Change the values returned by an original method through the after-listener.
- Change both the arguments and returned values of an original method through the around-listener.
- Override an original method (a conflicting change).

 **Note:** Overriding a class is a conflicting change. Extending a class's behavior is a non-conflicting change.

If several plugins apply to the same original method, the following sequence is observed:

1. The *before* listener in a plugin with the highest priority - that is, with the smallest value of `sortOrderargument()`.
2. The *around* listener in a plugin with the highest priority - that is, with the smallest value of `sortOrderargument()`.
3. Other *before* listeners in plugins according to sort order specified for them - that is, from the smallest to the greatest value().
4. Other *around* listeners in plugins according to the sort order specified for them - that is, from the smallest to the greatest value.
5. The *after* listener in a plugin with the lowest priority - that is, with the greatest value of `sortOrderargument()`.
6. Other *after* listeners in plugins, in the reverse sort order specified for them - that is, from the greatest to the smallest value



## 8.11 Configuration Inheritance

### Configuration Inheritance

Because of configuration inheritance, we can create a module, make it dependent from the core module (using the `<sequence>` node in the `module.xml`) and redefine preference for a certain interface.

In this case, our preference will be taken into account, which will have a similar effect as a rewrite in Magento 1.

For another option, we can redefine the argument declaration in the `di.xml` (in the same way as with a preference), which adds more flexibility into customizations.

HOME

Magento U

#### Notes:

Because of configuration inheritance, we can create a module, make it dependent from the core module (using the `<sequence>` node in the `module.xml`) and redefine preference for a certain interface.

In this case, our preference will be taken into account, which will have a similar effect as a rewrite in Magento 1.

For another option, we can redefine the argument declaration in the `di.xml` (in the same way as with a preference), which adds more flexibility into customizations.



## 8.12 Code Demonstration: Plugin

**Notes:**

Here is an example of a plugin. You see it is just a class, not extending anything, which means it doesn't have access to its protected variables.

The after-plugin will have as a parameter the original object and result. So you can modify the result, So that whatever you return here will be a new result.

The around plugin here, takes all parameter that the original plugin takes.

You can see here in the interceptor.php the call for the before plugin to be executed, then it checks Around plugin, or executes the original method, depending on what is found.

## 8.13 Reinforcement Exercise 1-8-1

### Reinforcement Exercise (1.8.1)

*The solution to this exercise can be found in your Magento 2 Fundamentals Exercise Solutions guide.*

For the class, `Magento\Catalog\Model\Product`  
method, `getPrice()`:

- Create a plugin that will modify price
- Create a preference that will substitute a core class with your own

HOME

Magento U

## 9. Events

### 9.1 Events

**Notes:**

This module focuses on Events and related topics.

## 9.2 Module Topics

### Module Topics



In this module, we will discuss...

- Events

[HOME](#)Magento U

**Notes:**

The topic to be addressed in this module is: events.

## 9.3 Events: Definition



Events | Definition

What are events?

**Events** are commonly used in applications to handle external actions or input, such as a user clicking a mouse. Each action is interpreted as an event.

HOME

Magento U

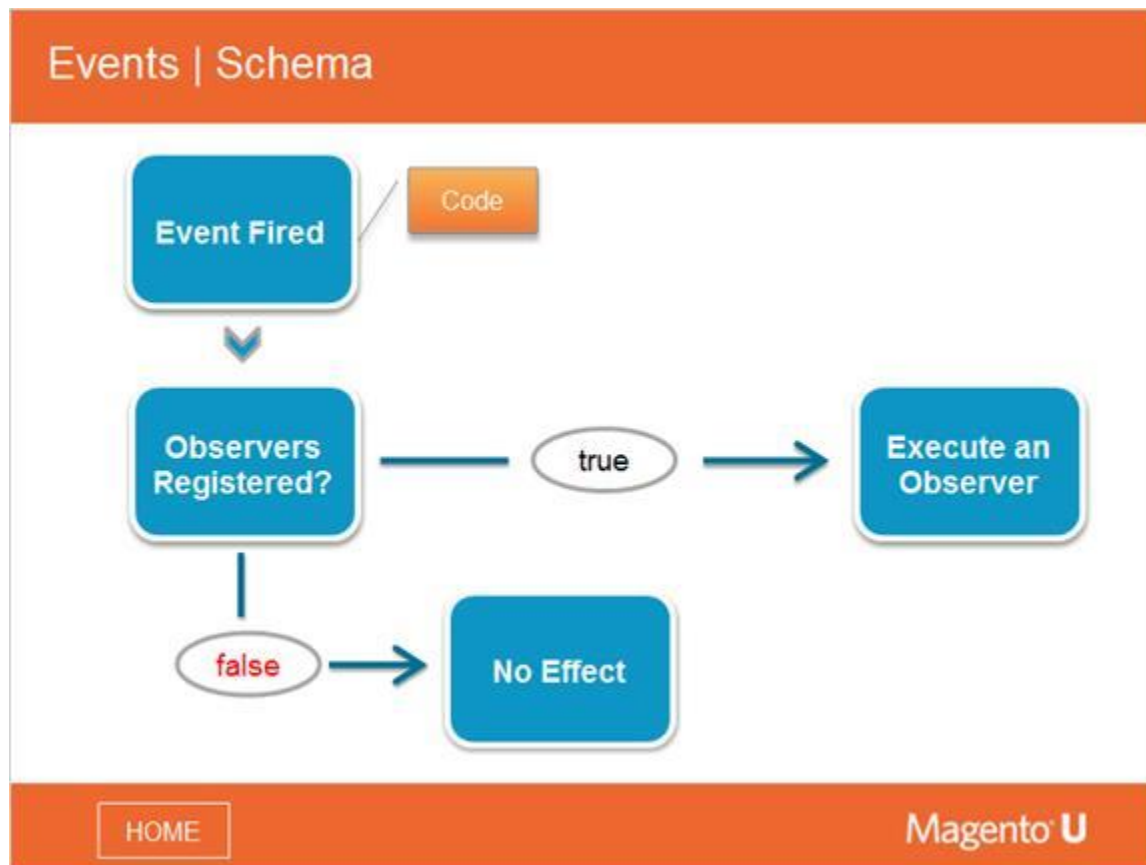
### Notes:

**Events** are commonly used in applications to handle external actions or input, such as a user clicking a mouse. Each action is interpreted as an event.

Events are part of the event-observer pattern. This design pattern is characterized by objects (subjects) and their list of dependents (observers). Events trigger objects to notify their observers of any state changes, usually by calling one of their methods.

It is the same concept as in Magento 1. Assume you need an event at certain place in the code. In Magento 1 it would be fired by `Mage::dispatchEvent()`, while in Magento 2 there is a special event manager class that fires events.

## 9.4 Events: Schema

**Notes:**

The code fires an event, and the event manager checks whether there are any observers registered. Events can be global, frontend, or admin. Events are declared in an events.xml file. If there is an observer registered, event manager will call this observer and pass the event's parameters to the observer. So, an observer has access to an event's parameters. If there are no observers registered, then the event has no effect.

## Event Firing Code (Slide Layer)

### Events | Schema

```
Class Magento\Catalog\Model\Product, method isSalable:
    public function isSalable()
    {
        $this->_eventManager->dispatch(
            'catalog_product_is_salable_before', ['product' => $this]);

        $salable = $this->isAvailable();

        $object = new \Magento\Framework\Object
            (['product' => $this, 'is_salable' => $salable]);
        $this->_eventManager->dispatch(
            'catalog_product_is_salable_after',
            ['product' => $this, 'salable' => $object]
        );
        return $object->getIsSalable();
    }
```

[RETURN TO  
DIAGRAM](#)

## 9.5 Registering an Event in XML

### Registering an Event in XML

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../lib/internal/Magento/
Framework/Event/etc/events.xsd">
  <event name="customer_login">
    <observer name="catalog" instance="Magento\Catalog\Model\
      Product\Compare\Item"
      method="bindCustomerLogin" shared="false" />
  </event>
  <event name="customer_logout">
    <observer name="catalog" instance="Magento\Catalog\Model\Product\
      Compare\Item" method="bindCustomerLogout" shared="false" />
  </event>
  <event name="page_block_html_topmenu_gethtml_before">
    <observer name="catalog_add_topmenu_items" instance="Magento\
      Catalog\Model\Observer" method="addCatalogToTopmenuItems" />
  </event>
</config>
```

[HOME](#)

Magento U

**Notes:**

In the firing code example, you can see how the product class fires two events: `catalog_product_is_salable_before` and `catalog_product_is_salable_after`, with parameters that will be available to the observer.



## 9.6 Observer Example

### Observer Example

```
public function bindCustomerLogin()
{
    $this->_getResource()->updateCustomerFromVisitor($this);
    $this->_catalogProductCompare->setCustomerId($this->
                                getCustomerId())->calculate();

    return $this;
}
```

[HOME](#)

Magento U

**Notes:**

This code example shows an observer from the class `Magento\Catalog\Model\Product\Compare\Item`, method `bindCustomerLogin()`.

## 9.7 Code Demonstration: Registering an Event



## 9.8 Reinforcement Exercise (1.9.1)

### Reinforcement Exercise (1.9.1)

*The solution to this exercise can be found in your Magento 2 Fundamentals Exercise Solutions guide.*

In your module:

- Create an observer to the event: `controller_action_predispatch`.
- Get a URL from the request object, `$request->getPathInfo()`.
- Log it into the file.

[HOME](#)

Magento U

**Notes:**

Note that if you decide to extend the xml config file, you might also need to update the xsd schema as well.