



Magento® U

Contents – Unit One

About This Guide.....	ix
1. Fundamentals Introduction	1
1.1 Fundamentals of Magento 2 Development - Unit One.....	1
1.2 Introduction.....	2
1.3 Appropriate Audience	3
1.4 Course Content.....	4
1.5 Best Way to Take the Course... ..	5
1.6 Begin Your Course	6
1.7 Unit One: Preparation & Configuration Home Page	7
2. Preparation	8
2.1 Preparation.....	8
2.2 Preparation Module Topics	9
2.3 Preparation LAMP & Composer.....	10
2.4 Preparation Magento 2 Installation	11
2.5 Preparation Magento 2 Modes	12
3. Overview & Architecture	13
3.1 Magento 2 Overview & Architecture	13
3.2 Overview Module Topics	14
3.3 Overview Magento 2 Platform & Architecture.....	15
3.4 Magento 2 Platform & Architecture Platform.....	16
3.5 Magento 2 Platform & Architecture Goals.....	17
3.6 Magento 2 Platform & Architecture Architecture.....	18
3.7 Magento 2 Platform & Architecture Areas.....	19
3.8 Magento 2 Platform & Architecture Magento 2 Essentials.....	20
3.9 Magento 2 Platform & Architecture Components	22
3.10 Magento 2 Platform & Architecture Paths.....	23
3.11 Magento 2 Platform & Architecture File Types	24
3.12 Magento 2 Platform & Architecture Config Files	25

3.13 Magento 2 Platform & Architecture PHP Classes.....	26
3.14 Magento 2 Platform & Architecture PHP Classes.....	27
3.15 Magento 2 Platform & Architecture Development Process.....	28
3.16 Magento 2 Platform & Architecture Enable Custom Code	29
3.17 Magento 2 Platform & Architecture Modules.....	30
3.18 Overview Modules.....	31
3.19 Modules Location	32
3.20 Modules Naming a Module	33
3.21 Modules Registering a Module/Empty Module Structure	34
3.22 Modules module.xml.....	35
3.23 Modules module.xml Example	36
3.24 Modules registration.php.....	37
3.25 Modules Module Dependencies.....	38
3.26 Modules Types of Module Dependencies	39
3.27 Modules Module Dependencies Tasks.....	40
3.28 Reinforcement Exercise 1.3.1: Modules.....	41
4. File System	42
4.1 File System.....	42
4.2 File System Module Topics	43
4.3 File System Root Folders.....	44
4.4 File System App Folder Contents.....	45
4.5 File System Framework & Core Modules	46
4.6 File System Core Source Code	47
4.7 File System Framework Source Code	48
4.8 File System Module Structure	49
4.9 File System Module View File Types.....	50
4.10 File System Templates	51
4.11 File System Templates (expanded).....	52
4.12 File System Themes	53
4.13 Demo Static Files	54

4.14 Check Your Understanding (1.4.A)	55
5. Development Operations.....	56
5.1 Development Operations	56
5.2 Development Operations Module Topics.....	57
5.3 Development Operations Modes	58
5.4 Modes Modes in Magento 2.....	59
5.5 Modes Developer Mode in Magento 2.....	60
5.6 Modes Production Mode in Magento 2	61
5.7 Modes Default Mode in Magento 2	62
5.8 Modes Summary of Mode Features	63
5.9 Modes Maintenance Mode in Magento 2.....	64
5.10 Modes Specifying a Mode	65
5.11 Modes Specifying a Mode: Environment Variable	66
5.12 Demo Specify Mode Using Environment Variable	67
5.13 Modes Specifying a Mode: Web Server Environment.....	68
5.14 Modes Specifying a Mode: php-fpm Environment.....	69
5.15 Reinforcement Exercise 1.5.1: Mode	70
5.16 Development Operations Command-Line Interface	71
5.17 Command-Line Interface Magento 2 CLI.....	72
5.18 Development Operations Cache.....	73
5.19 Cache Cache in Magento 2	74
5.20 Cache Cache Configuration.....	75
5.21 Cache Cache Type	76
5.22 Cache Cache Cleaning.....	77
5.23 Reinforcement Exercise 1.5.2: Cache	78
6. DI & Object Manager	79
6.1 Dependency Injection & Object Manager	79
6.2 DI & Object Manager Module Topics	80
6.3 DI & Object Manager Dependency Injection.....	81
6.4 Dependency Injection DI Pattern.....	82

6.5 Dependency Injection Overview.....	83
6.6 Reinforcement Exercise 1.6.1: Dependency Injection	84
6.7 Dependency Injection Class Instantiation in Magento 2	85
6.8 Dependency Injection Different Classes Instantiation	86
6.9 DI & Object Manager Object Manager	88
6.10 Object Manager.....	89
6.11 Object Manager Shared Instances Concept	91
6.12 Object Manager Object Manager Usage	92
6.13 Object Manager Magento 2 Best Practice.....	93
6.14 Check Your Understanding (1.6.A)	94
6.15 Check Your Understanding (1.6.B)	95
6.16 Object Manager Auto-generated Classes.....	96
6.17 Object Manager Configuration	97
6.18 Object Manager Configuration Specification	100
6.19 Object Manager Configuration Preferences Example	101
6.20 Object Manager Configuration Argument Example	102
6.21 Demo Dependency Injection	104
6.22 Object Manager Configuration Shared Argument Example	106
6.23 Reinforcement Exercise 1.6.2: Object Manager.....	107
7. Plugins	108
7.1 Plugins	108
7.2 Plugins Module Topics	109
7.3 Plugins Definition.....	110
7.4 Plugins Customizations	111
7.5 Declaring a Plugin	112
7.6 Plugin Example Before-Listener Method.....	113
7.7 Plugin Example After-Listener Method	114
7.8 Plugin Example Around-Listener Method.....	115
7.9 Prioritizing Plugins	116
7.10 Configuration Inheritance & Plugins	117

7.11 Plugins Interception.....	118
7.12 Reinforcement Exercise 1.7.1: Plugins 1	119
7.13 Demo Plugins	120
7.14 Reinforcement Exercise 1.7.2: Plugins 2	121
8. Events	122
8.1 Events	122
8.2 Events Module Topics.....	123
8.3 Events Definition.....	124
8.4 Events Schema	125
8.5 Events Core Example: Saving an Order Process	126
8.6 Events Core Example: Saving an Order Process	127
8.7 Demo Registering an Event	128
8.8 Reinforcement Exercise 1.8.1: Events	129
9. Module Configuration	130
9.1 Module Configuration	130
9.2 Module Configuration Module Topics.....	131
9.3 Module Configuration Configuration Files	132
9.4 Configuration Files Overview.....	133
9.5 Configuration Files Storage.....	135
9.6 Configuration Files core_config_data.....	136
9.7 Configuration Files Backend System Config Page.....	137
9.8 Configuration Files Scope	138
9.9 Configuration Files Load Order	139
9.10 Configuration Files Merging	140
9.11 Configuration Files Validation.....	141
9.12 Configuration Files Magento\Framework\Config	142
9.13 Configuration Files Magento\Config Loading	143
9.14 Demo Schema Files.....	144
9.15 Configuration Files Validating Configuration XML.....	145
9.16 Configuration Files Creating a Custom Config File	146

9.17 Configuration Files | Custom Config Files Example..... 148

9.18 Configuration | Error Reporting Settings 150

9.19 Error Reporting Settings | Overview 151

9.20 Check Your Understanding (1.5.A) 152





9.21 Check Your Understanding (1.5.B) 153

9.22 Reinforcement Exercise 1.9.1: Module Configuration 154

9.23 End of Unit One 155

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 Fundamentals of Magento 2 Development - Unit One



Notes:

Copyright © 2016 Magento, Inc. All rights reserved. 12-01-16


Fundamentals of Magento 2 Development v2.1

Software version: Magento 2 v.2.1.0

1.2 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.

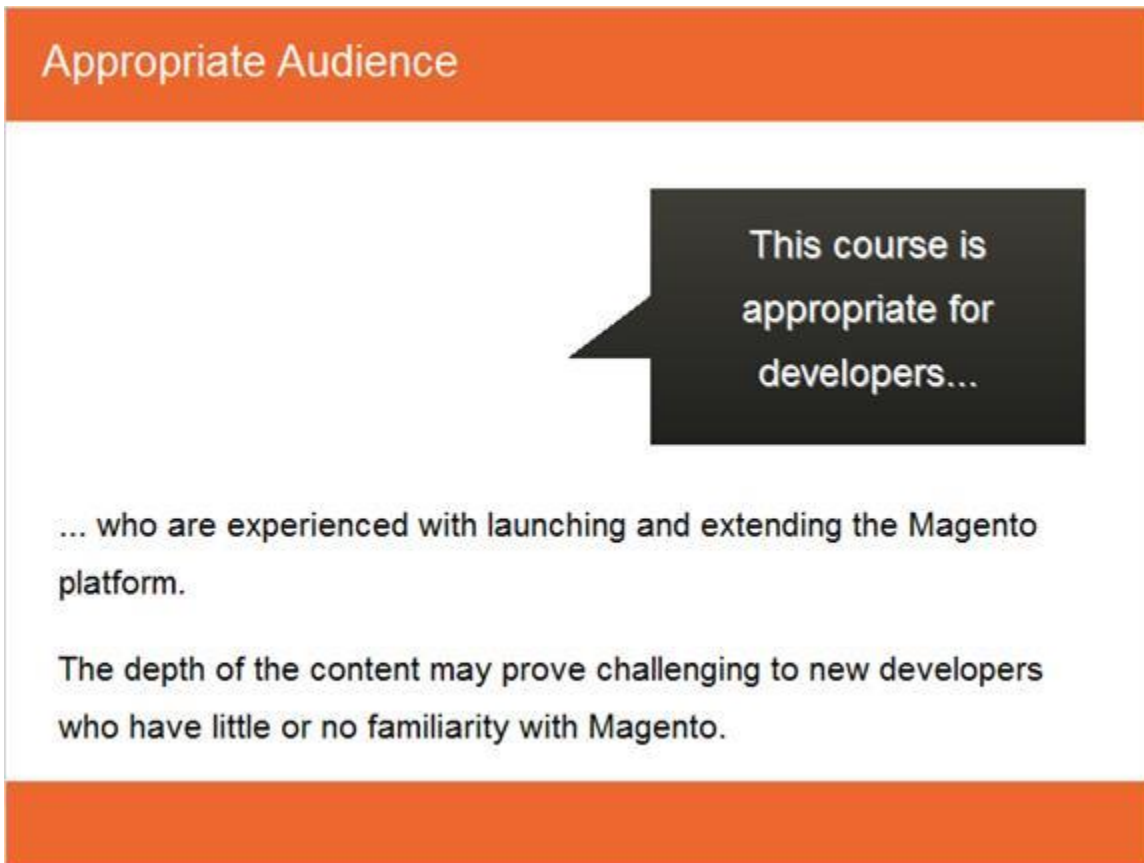
Notes:

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.

1.3 Appropriate Audience



The slide has an orange header bar with the text "Appropriate Audience". Below the header is a white rectangular area. In the upper right of this area is a dark gray speech bubble pointing left, containing the text "This course is appropriate for developers...". Below the speech bubble, there are two paragraphs of text. The first paragraph starts with "... who are experienced with launching and extending the Magento platform." The second paragraph starts with "The depth of the content may prove challenging to new developers who have little or no familiarity with Magento." The slide is framed by orange bars at the top and bottom.

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 little or no familiarity with Magento.

Notes:

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

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

1.4 Course Content

Course Content

This course covers major concepts in how Magento 2 functions

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.

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.

The content in this course is based on Magento 2.1.0.

1.5 Best Way to Take the Course...

Best Way to Take the Course...

To increase your comprehension and retention of the material, there are some easy, but important, steps you should take as you progress through the course:

- Have your Magento 2 installation open while you take the course.
- Look up the references suggested in the notes.
- Study the code examples on the slides and in the notes. Explore the code.
- Complete the exercises when presented in the course - this is an important step in reinforcing your knowledge of the topic just completed.

Notes:

Studies have shown that the more senses you use to learn a subject, the better your comprehension and retention of the material. So, do not use just the slides or just the notes to learn the material. Be sure to both listen to and read the slides as a topic is explained, and use the course guide notes.

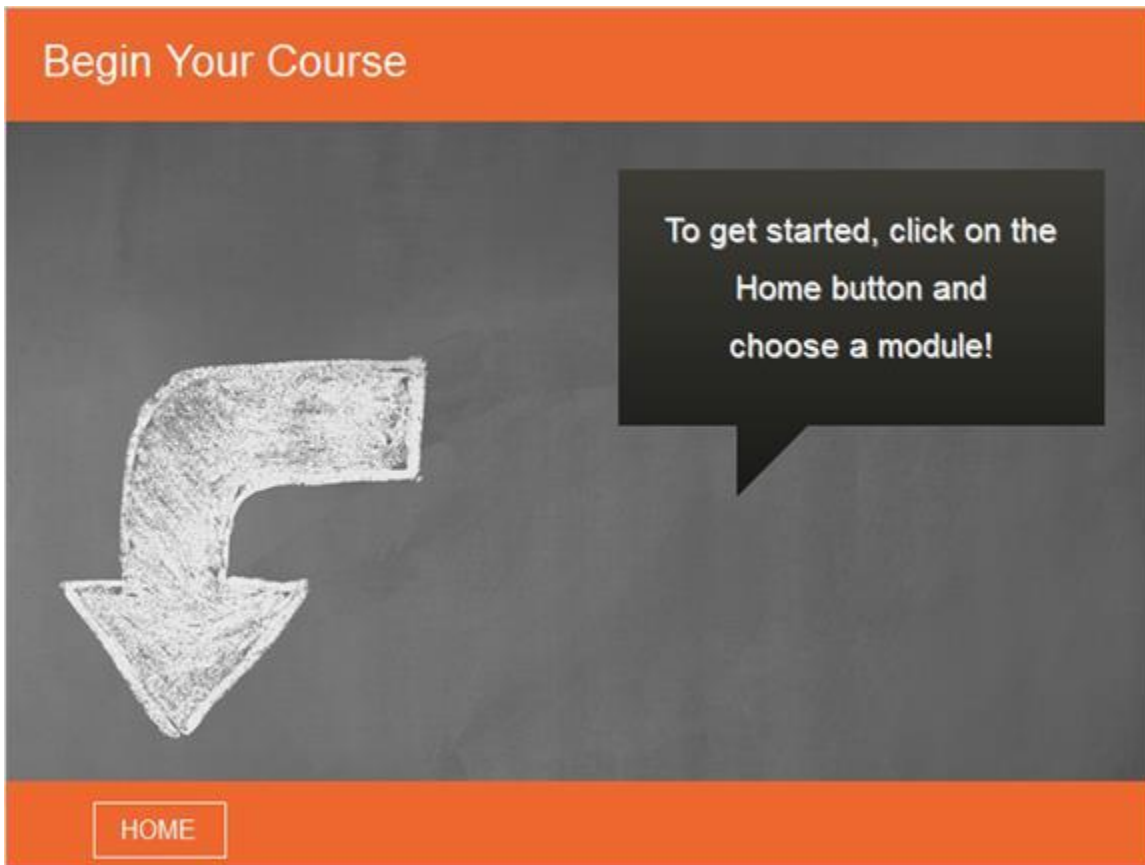
The course guide presents the narration (audio) in written form, while the slide highlights key concepts. Find the most effective way to use both to match your learning style. For example, you may want to read the slide first for a general grasp of the topic, then play the audio as you follow along with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

Be sure to do all the exercises presented in the course, as these are key learning opportunities and provide practical, hands-on experience with a native Magento 2 installation. These exercises have been specifically designed to help solidify the concepts when presented. This is also why you should complete the exercises when they appear in the course, and not skip them to be completed at another time.

Take the time to just explore the code, looking for examples of the topics presented.

Everyone learns differently. Following this advice should help you to get the most out of your course.

1.6 Begin Your Course

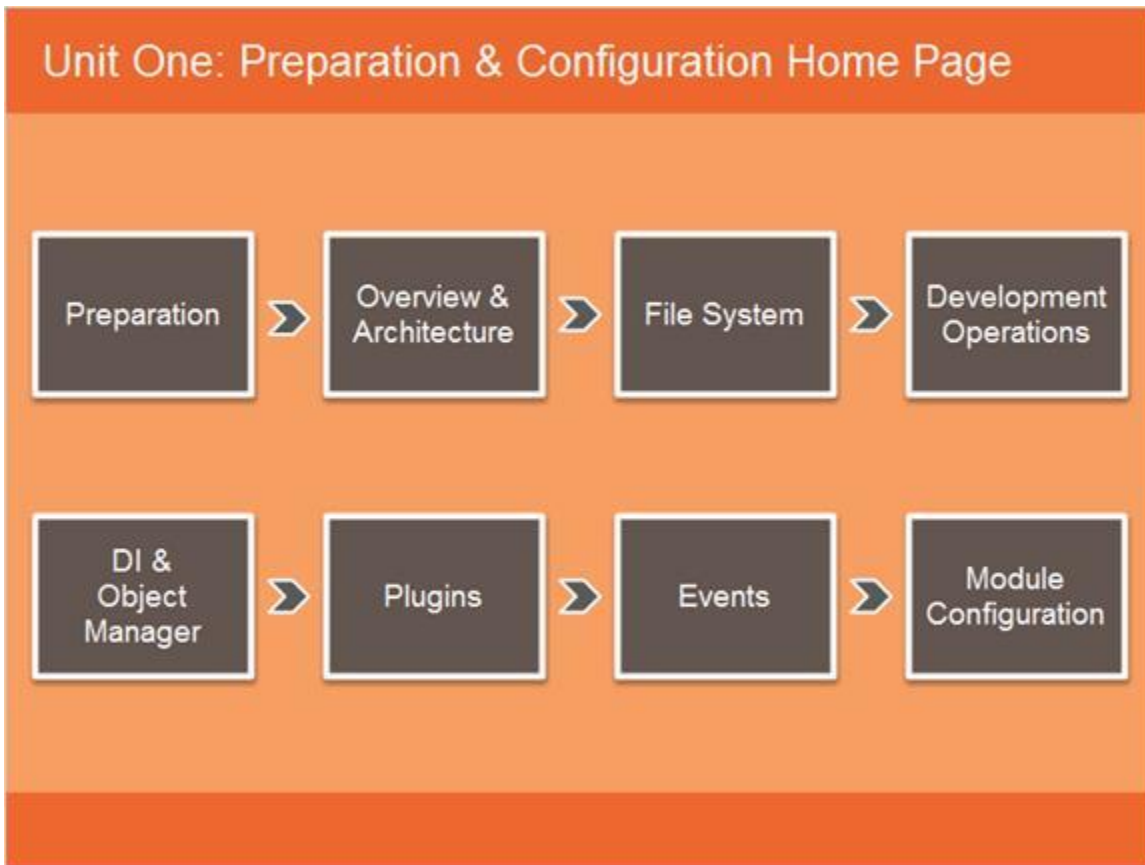


Notes:

To get started with the course, click the Home button now.

Enjoy your course!

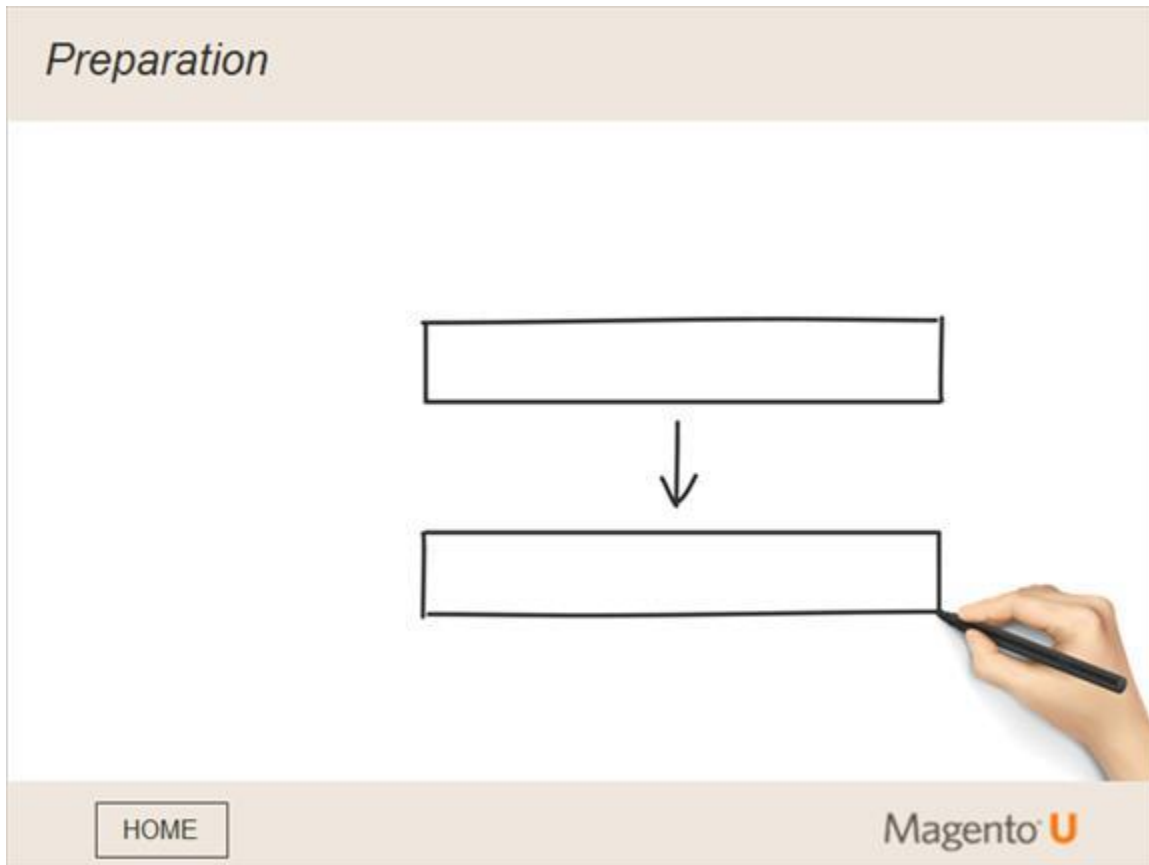
1.7 Unit One: Preparation & Configuration Home Page



Notes:

2. Preparation

2.1 Preparation



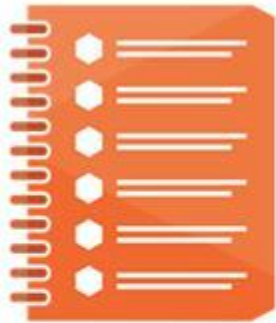
Notes:

In this module, we will discuss the preparation steps you should take to set up your Magento 2 installation for development.

We will cover configuration aspects in the next module.

2.2 Preparation | Module Topics

Preparation | Module Topics



In this module, we will discuss...

- Environment setup requirements
- Modes: Introduction

[HOME](#)

Magento U

Notes:

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

2.3 Preparation | LAMP & Composer

Preparation | LAMP & Composer

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 stack (Linux, Apache, MySQL, and PHP) along with PHP Composer, a tool for managing external dependencies. Magento makes extensive use of Composer for downloading and installing external dependencies such as Symfony.

Reference:

For more information on Composer, you can visit the website: <http://getcomposer.org>

2.4 Preparation | Magento 2 Installation

A screenshot of a web page titled "Preparation | Magento 2 Installation". The page has an orange header bar with the title. Below the header, the word "Installation" is displayed in bold. Underneath, there is a bulleted list with two items: "Cloning repository" and "Use Composer (recommended)". Below the list, a text line says "See the [Installation Guide](#) for more information...". At the bottom of the page, there is an orange footer bar containing a "HOME" button on the left and the "Magento U" logo on the right.

Preparation | Magento 2 Installation

Installation

- Cloning repository
- Use Composer (recommended)

See the [Installation Guide](#) for more information...

HOME

Magento U

Notes:

For installation you need a cloning repository. Using Composer is recommended.

See the Installation Guide for more information.

2.5 Preparation | Magento 2 Modes

Preparation | Magento 2 Modes

Modes

There are three modes in Magento 2: Default, Developer, and Production. These will be discussed in detail in Module 5.

For now, set your Magento mode to “Developer” (to display errors on the screen, not in the log file).

For Apache:

- Open the `.htaccess` file from the Magento root folder.
- Uncomment the first line and change it to: `SetEnv MAGE-MODE developer`

For other servers:

- Set the environmental variable `MAGE-MODE` to `developer`, following the instructions for your server.

[HOME](#)Magento U

Notes:

There are three modes in Magento 2: Default, Developer, and Production. We will discuss these modes in detail later in Module 5.

For now, set your Magento mode to Developer.

Use the settings shown on the slide for your server.

3. Overview & Architecture

3.1 Magento 2 Overview & 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.

3.2 Overview | Module Topics

Overview | Module Topics



In this module, we will discuss...

- Overview of the 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.

3.3 Overview | Magento 2 Platform & Architecture



3.4 Magento 2 Platform & Architecture | Platform

Magento 2 Platform & Architecture | Platform

The Magento 2 platform is...

- A flexible, open source commerce 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.

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

3.5 Magento 2 Platform & Architecture | 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 return on investment.

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)

3.6 Magento 2 Platform & Architecture | 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.

3.7 Magento 2 Platform & Architecture | Areas

Magento 2 Platform & Architecture | 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.
- Technically six areas, but you really only work with two: Adminhtml, frontend.

Six Areas Within Magento

Admin Panel (Adminhtml)	REST Web API
Storefront (frontend)	SOAP Web API
Crontab	Install

[HOME](#)


Notes:

An area is a scope in the configuration that allows you to load only the configuration files and options that are required for a specific request. Within Magento 2, there are areas for frontend, admin, and install, just as in Magento 1. Developers will primarily interact with only the Adminhtml and frontend areas. The entry point for those areas is the `index.php` file (and the `pub/index.php` file for the frontend).

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 it does not load, for example, the code that generates HTML pages using layouts.

Each area can have completely different code on how to process URLs and requests.

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.

 **Note:** API areas are discussed later in the course, in Unit Six.

3.8 Magento 2 Platform & Architecture | Magento 2 Essentials

Magento 2 Platform & Architecture | Magento 2 Essentials

Magento 2 Essential Concepts

1. Modular code structure
2. Themes
3. Layout files
4. Merged config files
5. Object instantiation “magic”
6. Naming conventions for controllers and layouts
7. Events and plugins

[HOME](#)

Notes:

This slide lists key Magento 2 concepts. We'll review them one by one.

1. Magento 2 is a modular system with its own specific structure. There are some exceptions to this rule -- for example, framework files are not technically modular, and some generic static files belong to a theme, not a module. Most of the Magento 2 code is wrapped in modules. We will cover module folder structure later.
2. Themes are a set of files that define how a website will look. Themes include all types of static assets, and are deeply connected to the Magento 2 rendering system (which is covered in Unit 3). Some static files are located in the module folder, while others are in a theme location. Themes and modules are connected, a concept covered in more detail later. For now, it is important to understand that while PHP code is mostly located in modules, generic static assets are organized in themes.
3. Layout, an essential concept in the Magento 2 rendering system, is a set of XML files that define which elements should be on a page (topology). Layout configuration defines which blocks should be present on a page and the block hierarchy. Block is a special PHP class in Magento 2 that is usually (but not always) connected to a template. Each block's generated HTML together comprises the whole page. So a block class generates a piece of HTML (using its template), and layout defines which blocks should be included on the page. Layouts are highly configurable and extendable. You can create layout updates in your module to modify any existing page or create a new layout for a new page. We will discuss layouts in depth in Unit 3.
4. In Magento 2, there are several types of configuration files -- for example, `events.xml`, `routes.xml`, `acl.xml`, and more. Those files are distributed across different modules. In other words, each module can have its own set of standard config files. When a certain type of information is requested by an application (like a list of available routes), Magento merges all config files of a certain type (example: `routes.xml`) and derives information from there.
5. Object instantiation “magic,” or dependency injection (DI), is a Magento 2 feature. When you need a new object, you declare it (or its interface) in the constructor, and Magento will deliver an instance. We will discuss this in greater depth later in this unit (in Module Five).

6. Naming conventions are another important feature of Magento. Routes are constructed in a special way in Magento 2. To process a route, you need to create a corresponding controller (and register your route in `routes.xml`). We will talk more about this in the next unit (Unit 2). Layouts are also connected to route names. So in order to add a new element to a page, you need to create a special layout update file, whose name is related to the route. We will talk about layouts in Unit 3.
7. Events and plugins are two major customization techniques in Magento 2. Events are fired in the core code, and developers can add observers to that event. Each event has parameters, so a developer can use or even modify them. Plugin technology allows developers to add specific behavior to every public method of each class. This is very powerful tool, described more fully later in this unit (in Module 7).

3.9 Magento 2 Platform & Architecture | Components

Magento 2 Platform & Architecture Components		
Component	Location	Comment
Configuration	app/etc/	env.php, modules.php
Framework	lib/internal/Magento/Framework	Framework classes
Modules	app/code/Magento	Business logic
Command-line Tool	bin/magento	Important utility; Run "php bin/magento list" to see all available commands.
Themes	app/design/	Contains static files that belong to a theme.
Dev Tools	dev	Various dev tools, like testing framework, sample data installer, ...

[HOME](#)Magento U

Notes:

This slide lists major Magento 2 components. These are high-level elements of Magento with which every developer will interact.

Configuration (Magento) refers to the general configuration for an instance. It is not the same as module configuration. General configuration defines which modules are available, the database credentials, and more. Module-specific configuration defines the behavior of a module.

Framework is low-level code, not related to features and business logic. Framework defines how the whole system works -- for example, interaction with databases, URL processing, logging, and remote requests to tools.

Modules are where Magento business logic and features are implemented. When developing with Magento, you create your own modules.

Command-line tool is a small, but critical, part of Magento 2. It is a bin/magento script that allows you to do a host of different things, like cleaning the cache, generating static content, and telling Magento to execute database upgrades.

Theme is a set of files (typically static assets) that define how pages look. Parts of a theme can be located in modules (for example, functional JavaScript), or in the folder app/design. Magento U offers an entire course on how to use theming in Magento 2.

Dev tools include scripts and tools that assist a Magento developer (for example, a testing framework would be located here).

3.10 Magento 2 Platform & Architecture | Paths

Magento 2 Platform & Architecture Paths		
	Composer Installation	Cloning Repo
Modules	vendor/magento/module-*	app/code/Magento/*
Framework	lib/internal/Magento/Framework/	vendor/magento/framework
Themes	vendor/magento/theme-frontend-*	app/design/frontend adminhtml/Magento/

Note: This course uses naming conventions from the Cloning Repo installation.

HOME

Magento U

Notes:

Magento 2 can be installed in two ways -- by cloning the repository, or by using a Composer installation. The themes and framework will vary based on the module paths. This slide shows the different paths when using either the Composer or repository methods.

You can find more information about installation in the official documentation:

<http://devdocs.magento.com/guides/v2.0/install-gde/bk-install-guide.html>

3.11 Magento 2 Platform & Architecture | File Types

Magento 2 Platform & Architecture | File Types

- Configuration files (XML files, and a few PHP files)
- PHP classes
- Layout instructions (*.xml files)
- Templates (*.phtml files)
- JavaScript modules (*.js files)
- JavaScript templates (*.html files)
- Static assets (CSS, images, ...)

[HOME](#)Magento U

Notes:

There are many different file types used in Magento 2.


This course focuses on the first four types presented on this slide.

The last three apply to frontend development, which is covered in a separate Magento U course.

3.12 Magento 2 Platform & Architecture | Config Files

Magento 2 Platform & Architecture | Config Files

- Global config files:
 - In the app/etc folder
 - di.xml, env.php, config.php, vendor_path.php
- Core and custom modules' config files located in the module's etc folder
- Theme configuration files

[HOME](#)

Notes:

Magento 2 configuration can be confusing to a developer who has never worked with Magento before, so all the major aspects will be covered in this course.

We can divide all configuration into three separate groups:

Global configuration: Magento instance configuration, which defines which modules are enabled, database credentials, session management, and so on.

Module configuration: Defines a module's behavior -- which routes it supports, which events it handles, etc.

Theme configuration: Defines a theme's name, parent, and some other details; usually a very small amount of code.

3.13 Magento 2 Platform & Architecture | PHP Classes

The slide features an orange header with the title 'Magento 2 Platform & Architecture | PHP Classes'. Below the header, the text 'Magento 2 PHP Classes' is displayed. A bulleted list follows, detailing various class types: Model/resource model/collection classes, API interfaces, Controllers, Blocks, and Observers. 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 PHP Classes

- Model/resource model/collection classes
- API interfaces
- Controllers
- Blocks
- Observers

HOME Magento U

Notes:

This slide presents types of PHP classes in Magento 2:

- **Model/resource/collection.** These classes are used to interact with databases, and to implement specific entity behavior (example: product or customer). Note that Magento 2 is moving away from using these classes to using API interfaces, but currently these classes are still an important part of Magento.
- **API Interface.** This is an internal module API that defines which operations are available. It includes CRUD operations for a module's entities as well as other module-specific operations (example: customer login). Magento 2 has a special mechanism that assigns certain classes to implement an API Interface. We will discuss this in more detail in Unit 5.
- **Controllers.** Controllers are classes that handle specific pages in accordance with the controller role in the MVC (Model-View-Controller) pattern.
- **Blocks.** Blocks are special classes that represent a part of a page. Blocks are usually (but not always) connected to PHTML template files. Blocks can be thought of as a data container for a template, which represents a piece of the HTML on the page. We will talk more about blocks in Unit 3.
- **Observers.** Magento 2 fires events in different places along its execution flow. Developers can create Observer classes that will react to events.

3.14 Magento 2 Platform & Architecture | PHP Classes

Magento 2 Platform & Architecture | PHP Classes

Magento 2 PHP Classes

(continued)

- Plugins
- Helpers
- Setup / upgrade scripts
- Ui components
- Other...

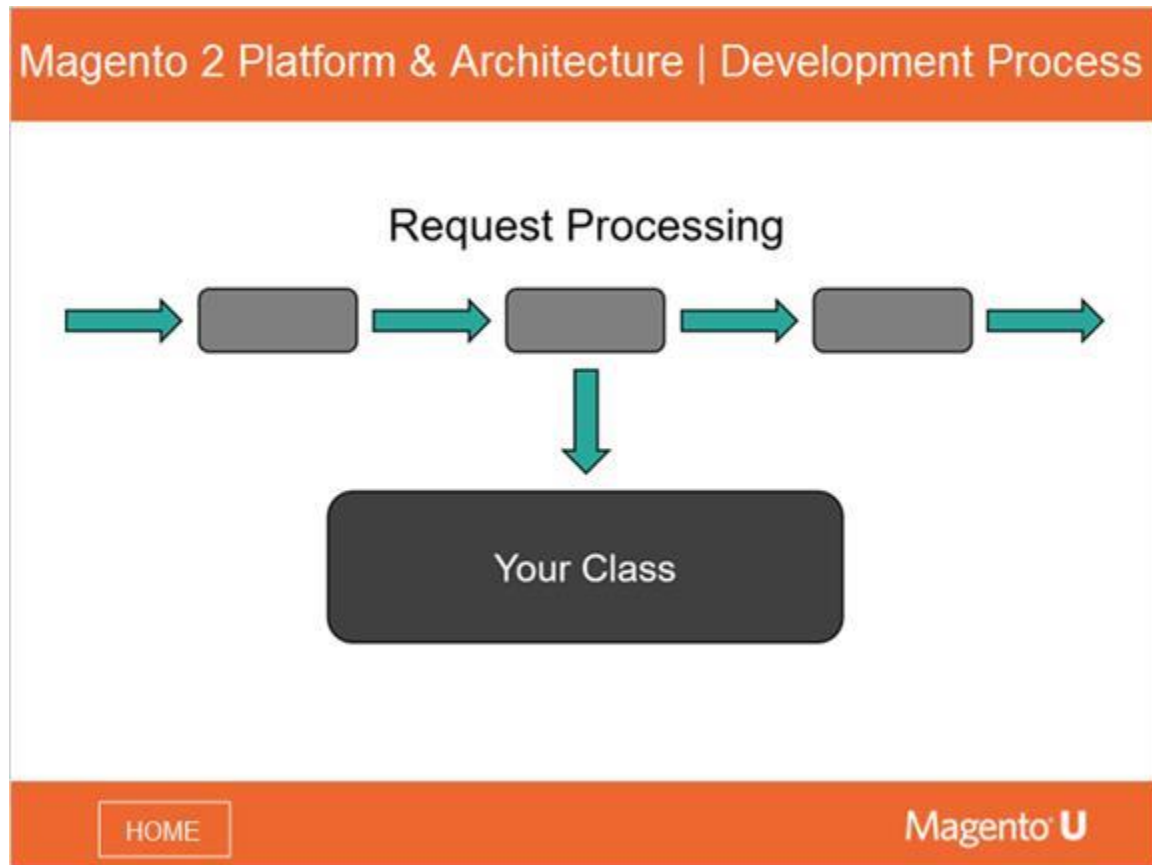
[HOME](#)

Notes:

(continued)

- **Plugins.** Plugins are a very powerful customization technique. Plugins can be thought of as a wrapper around any public method of any class. A developer can modify a method's behavior by using plugins.
- **Helpers.** Helpers are auxiliary classes that encapsulate some useful functions (ex: performing tax calculations).
- **Setup/upgrade scripts.** These scripts are special files that upgrade a database schema, or add data to a database. We will talk more about databases and scripts in Unit 4.
- **UiComponents.** The UiComponent technology in Magento 2 allows a developer to create a component -- an independent element on a page with its own backend part. We will talk more about UiComponents in Unit 3.
- **Other...** Note that while we listed the most important PHP class types here, it is not a complete list.

3.15 Magento 2 Platform & Architecture | Development Process



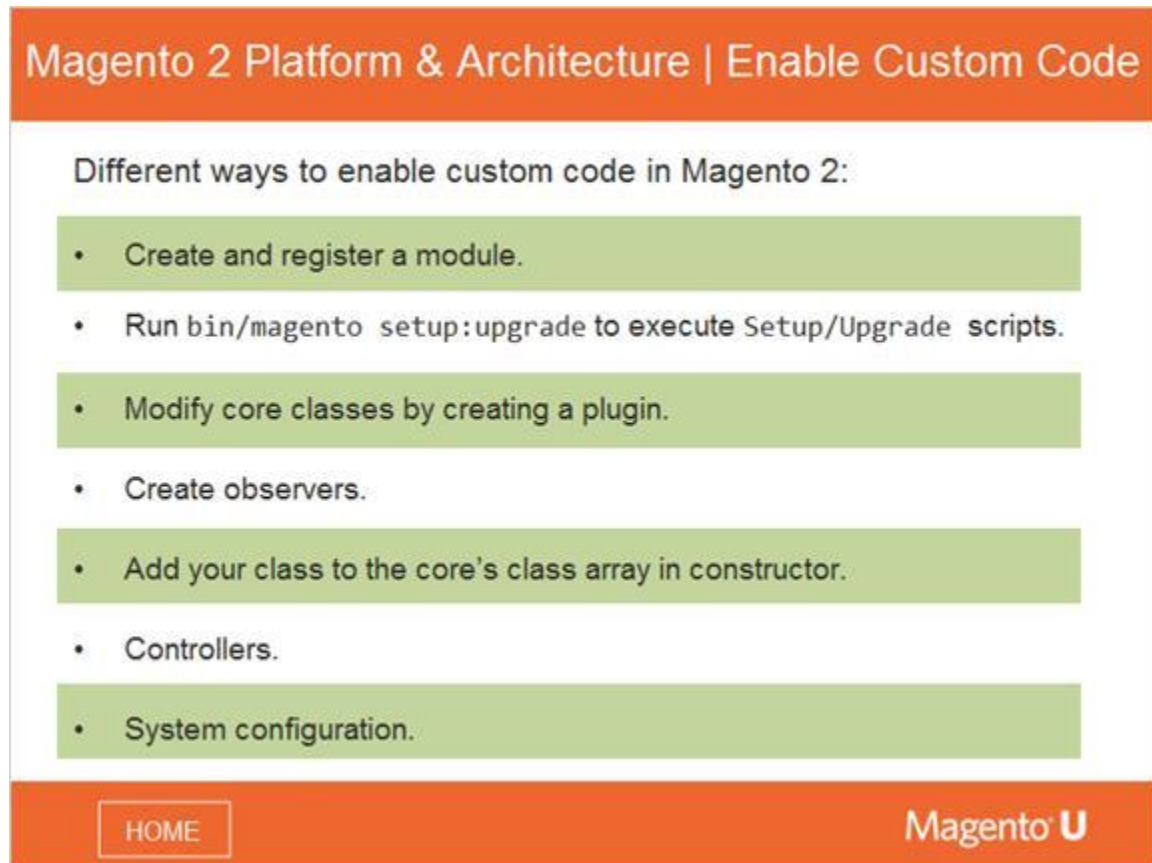
Notes:

This slide shows the typical development/customization process. Your class is placed somewhere in the execution flow, where some of its methods get executed.

There are many ways that this can be accomplished, including:

- Adding the class into the array of another class's constructor
- Creating a plugin
- Creating an observer

3.16 Magento 2 Platform & Architecture | Enable Custom Code



The slide features an orange header with the title 'Magento 2 Platform & Architecture | Enable Custom Code'. Below the header, the text 'Different ways to enable custom code in Magento 2:' is followed by a bulleted list of seven items, each highlighted in a light green box. At the bottom, there is an orange footer bar containing a 'HOME' button and the 'Magento U' logo.

Magento 2 Platform & Architecture | Enable Custom Code

Different ways to enable custom code in Magento 2:

- Create and register a module.
- Run `bin/magento setup:upgrade` to execute Setup/Upgrade scripts.
- Modify core classes by creating a plugin.
- Create observers.
- Add your class to the core's class array in constructor.
- Controllers.
- System configuration.

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

Notes:

This slide provides more detail on “enabling” custom code. We will cover all of these later in this course.

Note that this list is not complete -- it provides more commonly used generic steps, rather than feature-specific ways (for example, create a backend attribute for a product, or a total model for a quote).

Magento has a variety of ways to execute your code in different situations.

3.17 Magento 2 Platform & Architecture | Modules

Magento 2 Platform & Architecture | Modules

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 .php and .xml files (blocks, controllers, helpers, models, ...) related to a specific functionality.
- Using a modular approach implies that every module encapsulates a feature and has minimum dependencies on other modules.

[HOME](#)Magento U

Notes:

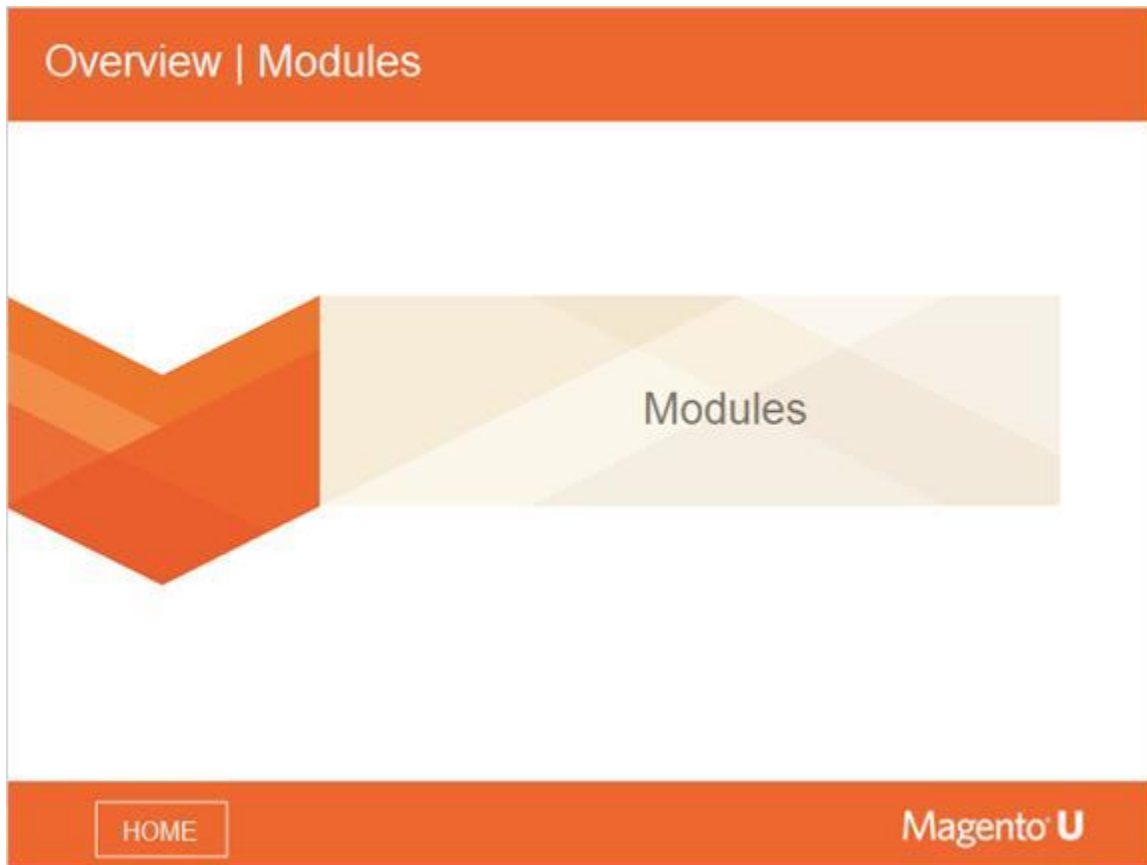
In Magento 1, files belonging to one module were not kept within a single directory; In Magento 2, a module is now contained within a single folder.

In Magento 2, modules have been made more granular, are better organized, and their dependencies are more explicit. This facilitates building scalable and maintainable shops. Magento 2 modules are also smaller and the functionality is grouped in a more logical manner (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.

3.18 Overview | Modules



3.19 Modules | Location

Modules | Location

Modules:

- Are located under the `/app/code` or `vendor` directory of a Magento installation, in a directory following PSR-4 compliant format:

`app/code/<vendor>/<module_name>/`
`vendor/<vendor-composer-namespace>/<module_name>/`
- Modules residing within `vendor/` register themselves using a Composer autoloader file to call:

`\Magento\Framework\Module\Registrar::registerModule`
`('<module name', __DIR__')`

[HOME](#)Magento U

Notes:

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

Third-party modules can also be installed directly into the `vendor/` folder from packagist or private repositories by composer.

Modules under `vendor/` make themselves known to Magento using a Composer autoload file callback as in:


```
\Magento\Framework\Module\Registrar::registerModule(<module name>, __DIR__);
```

3.20 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.

A module name consists of the vendor name and the module name separated with an underscore (_). Both the vendor and the module name have to start with an uppercase character, for example: `Magento_Customer`.

Code Base of Custom Module	<code><root>/app/code/<Vendor>/<Module></code>
Custom Theme Files	<code><root>/app/design/<area>/<Vendor>/<theme>/<Vendor_Module></code>
Code Base of Custom Module	<code><root>/app/code/<Vendor>/<Module></code>
Custom Theme Files	<code><root>/app/design/<Module>/<theme></code>

3.21 Modules | Registering a Module/Empty Module Structure

Modules | Registering a Module/Empty Module Structure

- Every module is located in its folder.
- Custom-written modules are located in the `app/code` folder, regardless of installation type.
- Modules are grouped by vendors.
- The usual path to a module: `app/code/<Vendor>/<ModuleName>` name (for Composer installation core modules located at `vendor/magento/module-*`).
- Modules are usually referred to by `<Vendor>_<ModuleName>` name (ex: `Magento_Catalog`).
- Every module must have an `etc/module.xml` and a `registration.php` file.

[HOME](#)Magento U

Notes:

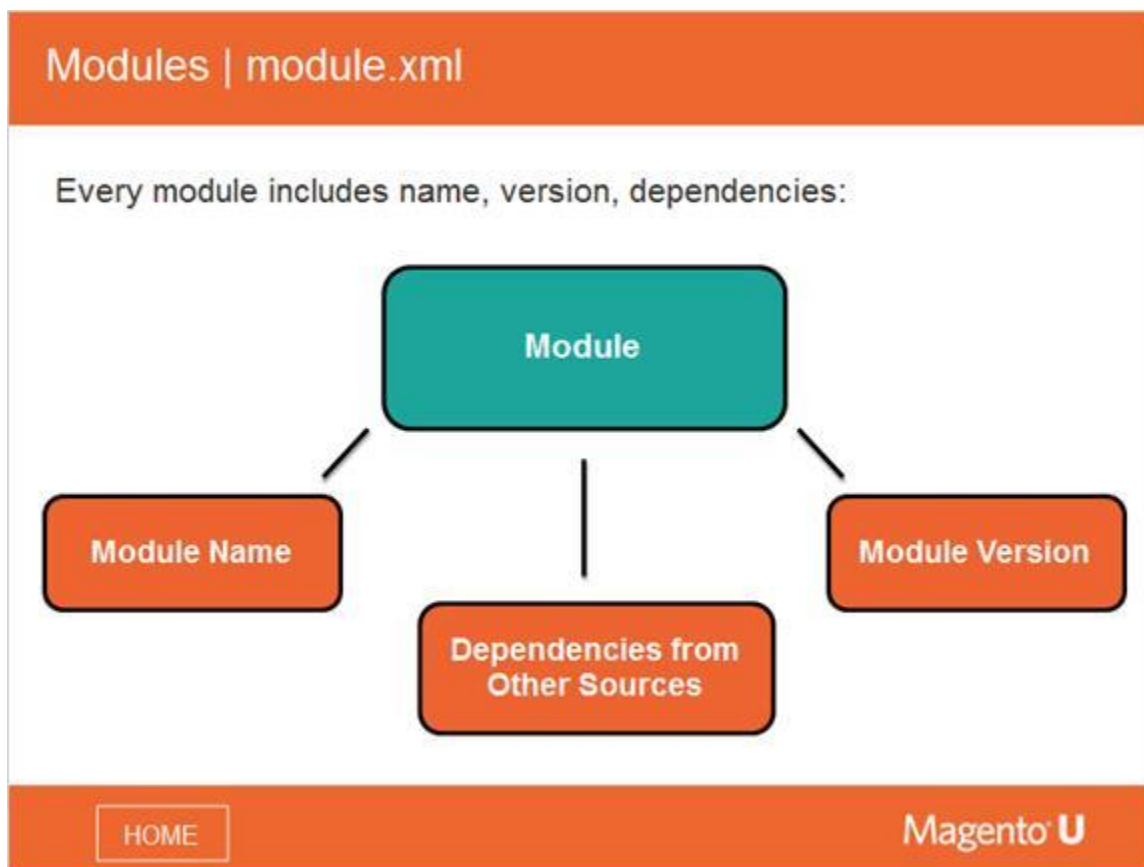
Magento 2 is built of modules. Its core is technically a set of modules. When a developer creates a customization, first thing to do is to create a module.

The slide describes a module. It is technically a folder located in a specific place (`app/code` or `vendor/<VendorName>/`) which has a few required elements, such as the `etc/module.xml` file and `registration.php` file.

Modules are grouped into vendors. So typically you define a vendor folder first (for example, `app/code/MagentoU` -- `MagentoU` is a vendor name here), and then a module name (ex: `app/code/MagentoU/MyModule` -- `MyModule` is a module name). In Magento modules are named `<Vendor>_<Module>`. For example `Magento_Catalog` (`Magento` = vendor; `Catalog` = module), or `MagentoU_MyModule`. **Note that module names are case sensitive.**

In the following slides we will cover the `module.xml` and `registration.php` files.

3.22 Modules | module.xml



Notes:

The `module.xml` file must be located in the `etc` subfolder of a module (ex: `Magento/Catalog/etc`).

Note that "etc" starts with a lowercase "e" while typically a module's subfolders start with an uppercase letter.

The `module.xml` file includes three pieces of information: name, version, and dependencies.

3.23 Modules | module.xml Example

Modules | module.xml Example

```
<?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 `composer.json` shows the `Magento_CatalogInventory` hard dependency on `Magento_Catalog` and other modules.

For more information, see: `app/code/Magento/CatalogInventory/composer.json` in your installation.

```
}
    "name": "magento/module-catalog-inventory",
    "description": "N/A",
    "require": {
        "php": "~5.5.0|~5.6.0|~7.0.0",
        "magento/module-config": "100.0.*",
        "magento/module-store": "100.0.*",
        "magento/module-catalog": "100.0.*",
        "magento/module-customer": "100.0.*",
        "magento/module-eav": "100.0.*",
        "magento/module-quote": "100.0.*",
        "magento/framework": "100.0.*",
        "magento/module-ui": "100.0.*"
    },
    ...
}
```

3.24 Modules | registration.php

Modules | registration.php

- Contains instruction on how to find a module, and usually follows this pattern:

```
<?php

\Magento\Framework\Component\ComponentRegistrar::register(
    \Magento\Framework\Component\ComponentRegistrar::MODULE,
    'Magento_Catalog',
    __DIR__
);
```


Module name is the only part of registration.php that varies

HOME

Magento U

Notes:

Here is an example of a registration.php file. This file follows a typical pattern -- the only thing you will change here for your module is the module name; for example, MagentoU_MyModule.

-  **Note:** The module name must match the module path, so the registration.php file and the MagentoU_MyModule should be located either in app/code/MagentoU/MyModule/registration.php or in the vendor folder: vendor/magento/module-mymodule.

3.25 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)

Only **soft** sort order dependencies on other Magento modules can be declared in `module.xml`. Other dependencies can be listed in a `module composer.json` configuration file.

Best Practice: You can lose historical information contained in a module if it is removed or disabled. Be sure to store such information before you remove or disable a module.

3.26 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 need 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.

3.27 Modules | 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 third-party library for a module -- you will need to use Composer. A module's "hard" 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 on 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 or installed is important.
- Only for modules -- no other component type is entered in the `<sequence>` section.

This assumes that using the `<sequence>` node is a best practice.

3.28 Reinforcement Exercise 1.3.1: Modules

Reinforcement Exercise 1.3.1: Modules

*See your course Exercises Guide for instructions
on how to complete this exercise, and the solution.*

Click "Next" when done

HOME

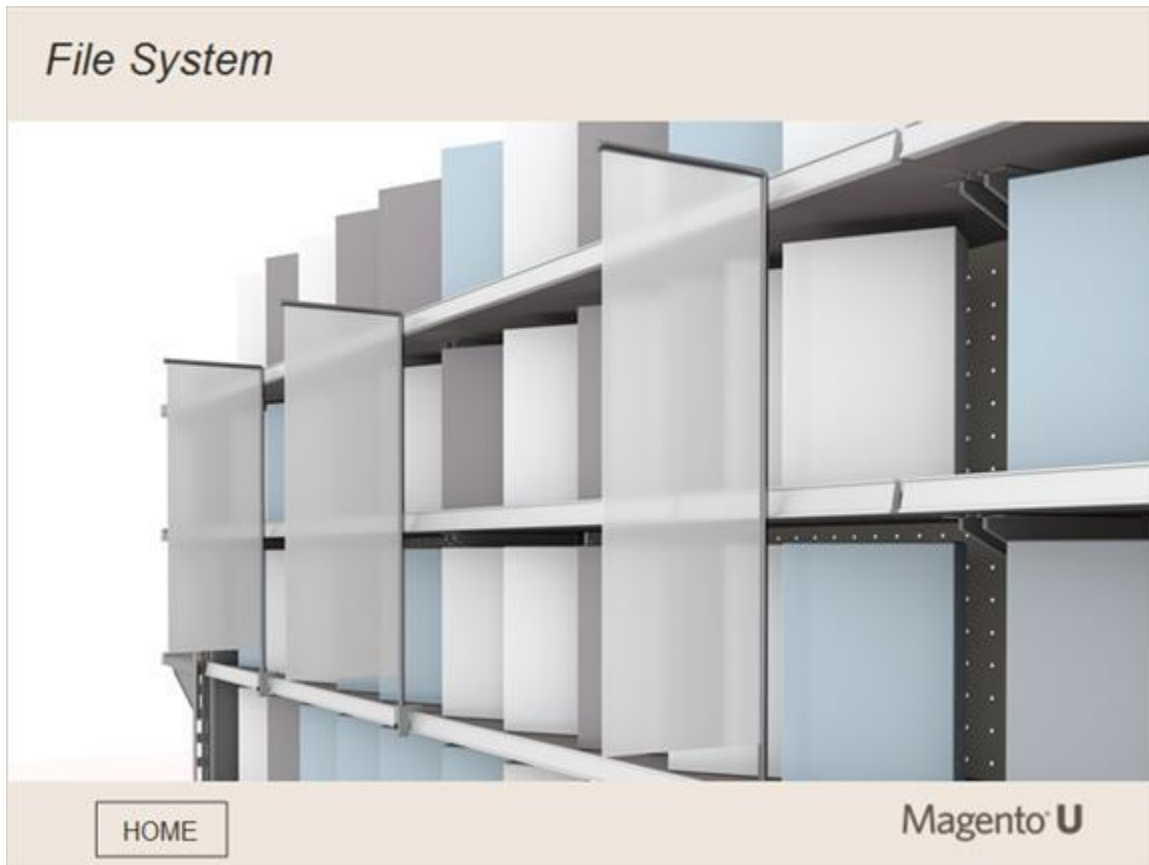
Magento U

Notes:

See your course Exercises Guide for instructions on how to complete this exercise, and the solution.

4. File System

4.1 File System

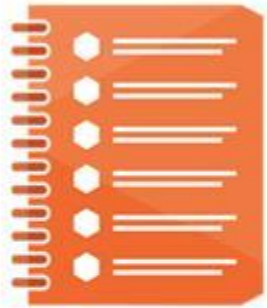


Notes:

Let's look at the Magento 2 file system.

4.2 File System | Module Topics

File System | Module Topics



In this module, we will discuss...

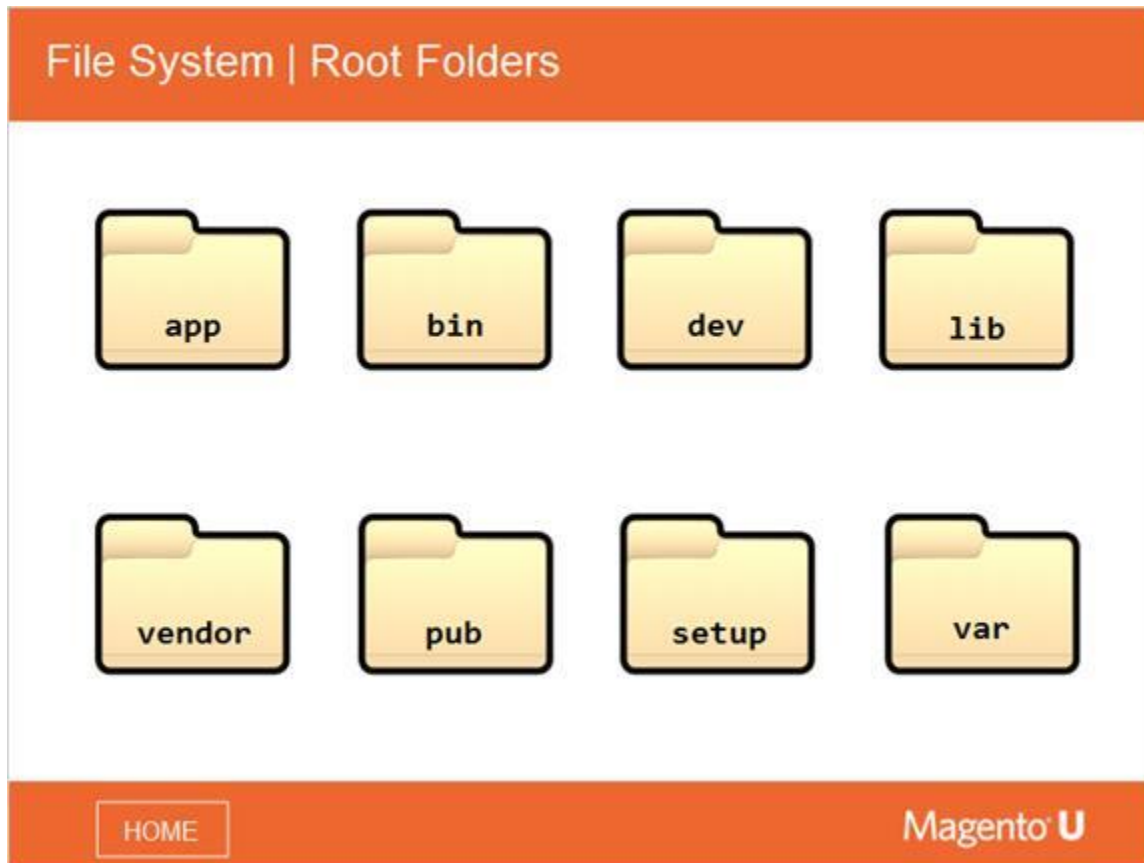
- Essential file system structures

[HOME](#)Magento U

Notes:

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

4.3 File System | Root Folders

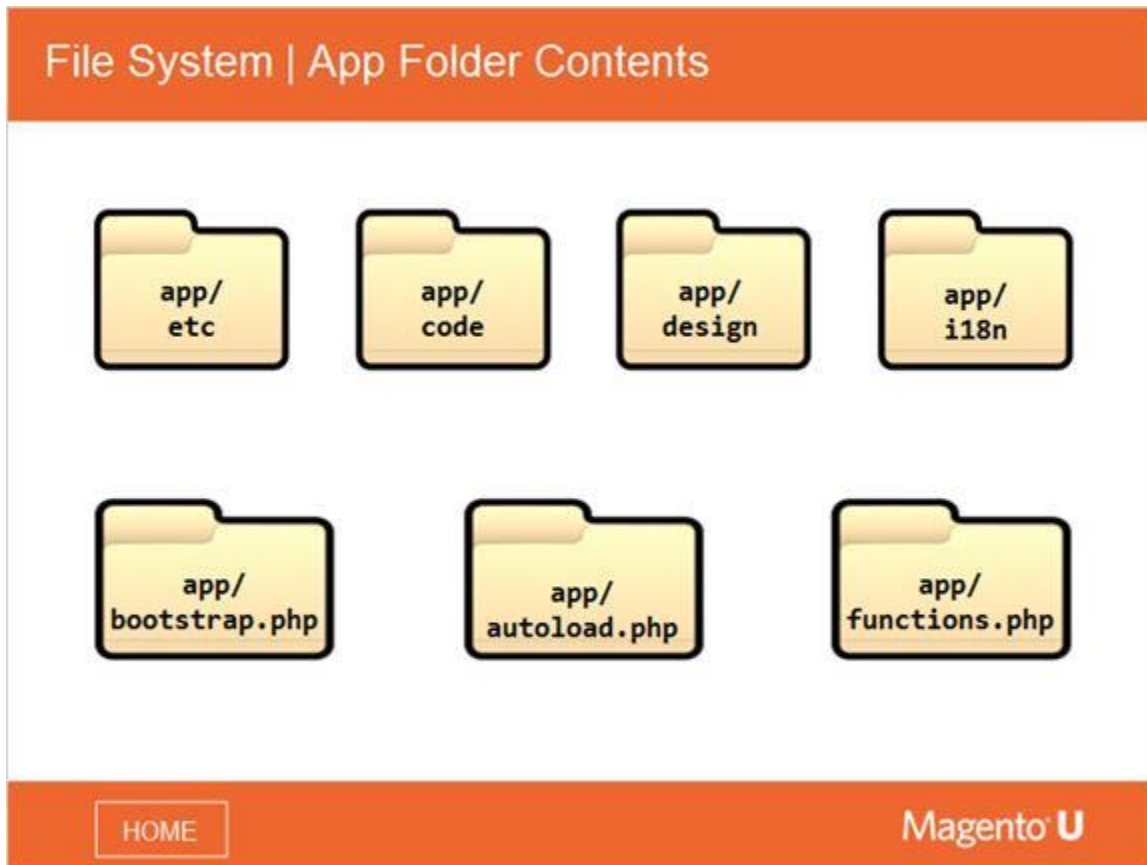


Notes:

Here are Magento 2's eight root folders.

1. **app**: includes core code (in the case of repo cloning installation), custom modules, themes, and global configuration.
2. **bin**: has the "magento" script, which is the Magento 2 command-line tool described earlier.
3. **dev**: includes different scripts and tools useful for developers.
4. **lib**: includes external libraries not installed through the composer. For the repo-cloning installation, it includes the Magento framework: `lib/internal/Magento/Framework`.
5. **vendor**: the composer folder, which contains all the dependencies. It may also include Magento framework and core modules for composer-based installation.
6. **pub**: folder that should be public on the host. This folder has a `static` subfolder where all static files need to be placed in production. Static files can be located in themes and modules, so moving them all to `pub/static` is not a trivial task. There is a special command available within the command-line tool which allows for that. However, as it takes a while to accomplish this, and is not convenient for development, there is also a `static.php` file that locates and returns static files from the modules and themes.
7. **setup**: folder that contains installation-specific files.
8. **var**: an important folder, where cache, generated code, logs and other files (like uploaded csv files) are stored.

4.4 File System | App Folder Contents



Notes:

Depicted here are the contents of an app folder.

app/etc: includes global configuration.

app/code: custom modules, and in the case of cloning-repo installation, core modules.

app/design: themes.

app/i18n: specific translation files.

app/Bootstrap.php, **app/autoload.php**, and **app/functions.php** are three special PHP files that must be included at the very beginning of execution process.

4.5 File System | Framework & Core Modules

File System | Framework & Core Modules

Magento 2 consists mainly of Core and Framework modules.

- Core is located at `app/code/Magento/` (or `<vendor>/Magento/module-*`).
- Framework modules located at `lib/internal/Magento/Framework` (or `<vendor>/magento/framework`).
- Even though Framework has a modular structure like Core, it is composed of folders only; Core folders are more like modules in that they contain information (version number, ...).

[HOME](#)Magento U

Notes:

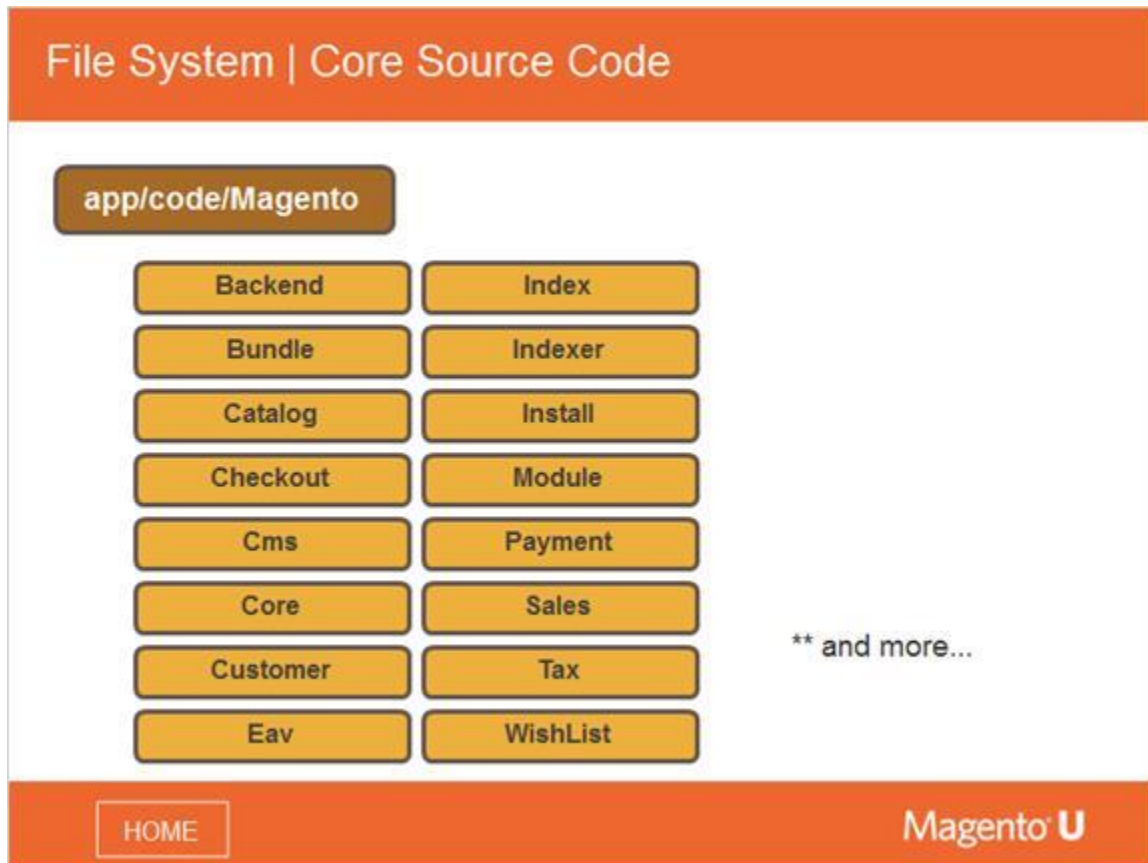
The slide describes where Magento 2 code is located. It can be separated to application code (main code) and framework code. Main code includes Magento 2 application logic, while framework contains general code not related to any specific feature.

Magento code is mainly organized as modules. A module is a folder with a specific structure that contains code (a structural unit of code). You can perform an operation with a module (update, disable and so on).

Framework is also *structured* as a set of folders with code, like Core, but they are just folders, not a module. This is an important point - although framework and the main/core code at first glance look similar (both a set of folders with code), they are not. Framework's folders are just folders, while main code folders are modules.

Main code located in the `app/code/Magento` folder (or `vendor/magento/module-*`) and framework is located at `lib/internal/Magento/Framework` (or `vendor/magento/framework`).

4.6 File System | Core Source Code



Notes:

Magento code is organized in the following way. There is a folder, `app/code`, which is where most of the PHP code is located. The code is packaged inside modules. Each module belongs 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, for code under `app/code`, the folder structure is `app/code/<vendor>`, and inside this folder is the list of modules.

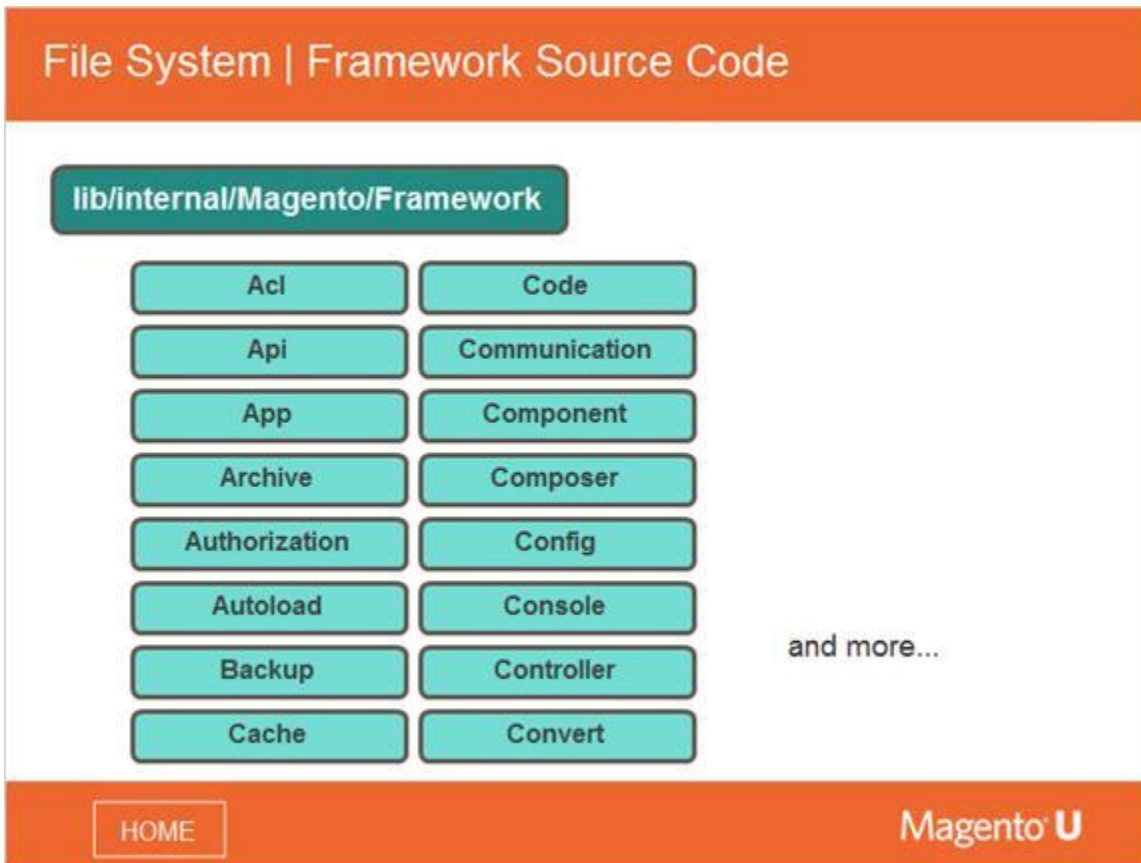
PHP code installed under the `vendor/` directory is located using the composer autoloader, so the vendor namespace and module name folders follow the composer convention of using all lower case. The module then adds itself to the composer autoloader list with its upper case Vendor and Module name.

For example, a module installed into `vendor/example-vendor/my-module` with its code inside of an `src` folder would add the following section to its `composer.json` file:

```
"autoload": {
    "psr-4": {"ExampleVendor\\MyModule\\": "src"},
    "files": ["src/module-registration.php"]
},
```

- Note:** There are many more available subfolders for the Core source code, Framework source code, and Module file structures (this and next two slides). The folders chosen for display reflect alphabetical order, not importance.

4.7 File System | Framework Source Code

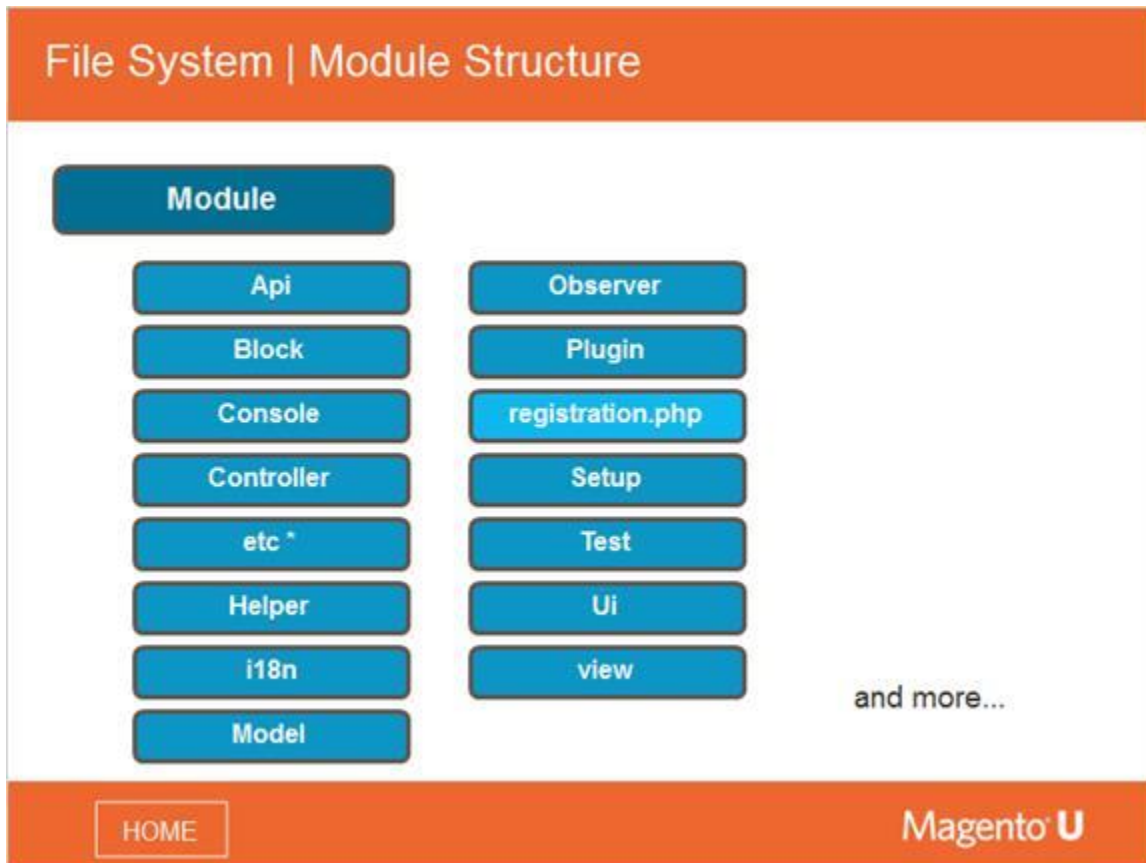


Notes:

Reference:

<https://github.com/magento/magento2/tree/develop/lib/internal/Magento/Framework>

4.8 File System | Module Structure



Notes:

The diagram shows possible subfolders of a module folder. * **Note that the only required folder is etc.**

The structure is not as strict as in Magento 1, so a module might have any subfolder. Also, a developer can place a file in any subfolder. However, the listed folders correspond to different class types, so this structure is historical and should be retained.

The most popular are:

- Block: For blocks
- Controller: For controller actions
- Helper: For helpers
- Model: Where models and resource models are located
- Setup: For upgrade scripts
- view: For layout XML files, templates, and static view files.

Note that the file name must repeat the folder structure, starting from <Vendor>/<ModuleName>. For example, the PHP class in the file `app/code/Magento/Catalog/Model/Product.php` must have the name `Magento\Catalog\Model\Product`.

The "capital-letter" folders usually have special meaning - they will be covered in more depth in the next unit, Unit 2.

The view folder is distinct, and will be covered extensively in Unit 3. However, we'll take a quick look at it now.


To see a typical module file structure, see:

<http://devdocs.magento.com/guides/v2.0/extension-dev-guide/build/module-file-structure.html>

4.9 File System | Module View File Types

File System | Module View File Types

- Ui_components (frontend)
- Templates
- Layouts
- Static files
- JavaScript modules
- JavaScript templates



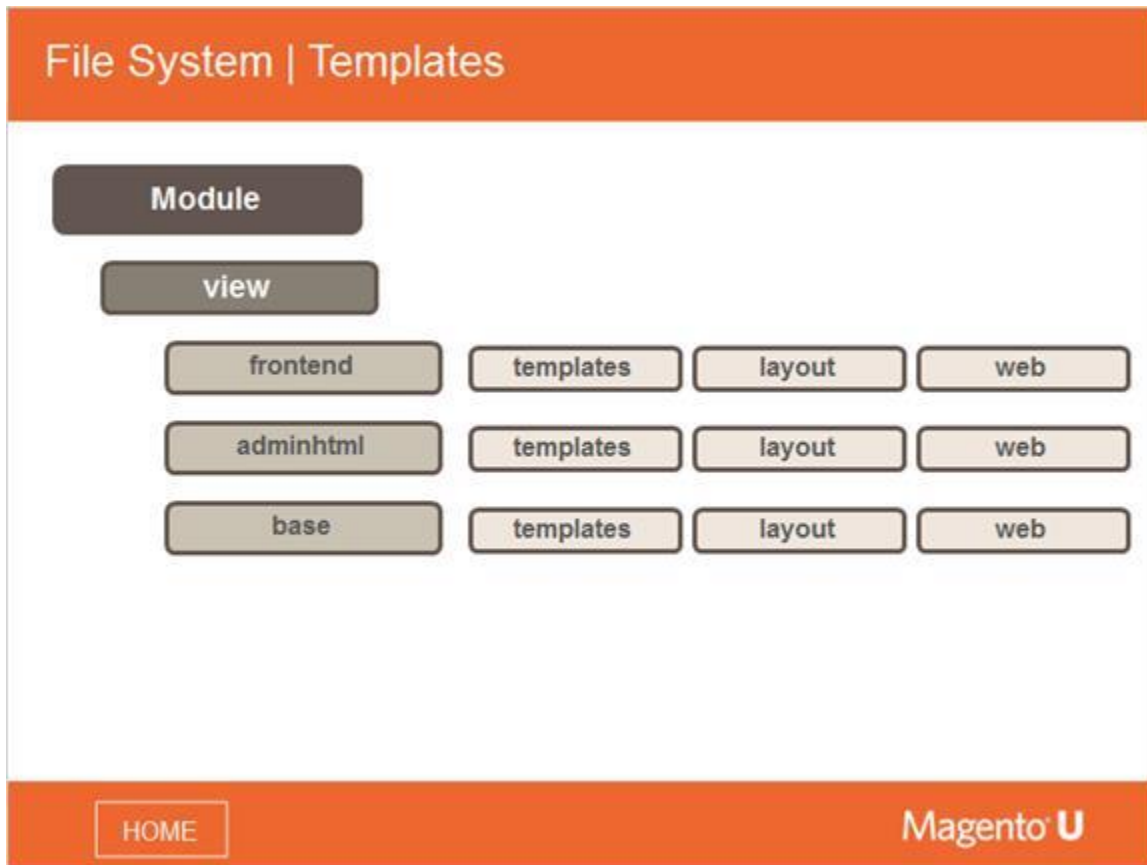
The diagram shows a list of file types: Ui_components (frontend), Templates, Layouts, Static files, JavaScript modules, and JavaScript templates. A large curly bracket groups the last three items (Static files, JavaScript modules, and JavaScript templates) and is labeled 'web'.

[HOME](#)Magento U

Notes:

This slide shows the different subfolders within the view folder for a module. Their names are self-explanatory, and we will discuss them in more detail in Unit 3.

4.10 File System | Templates



Notes:

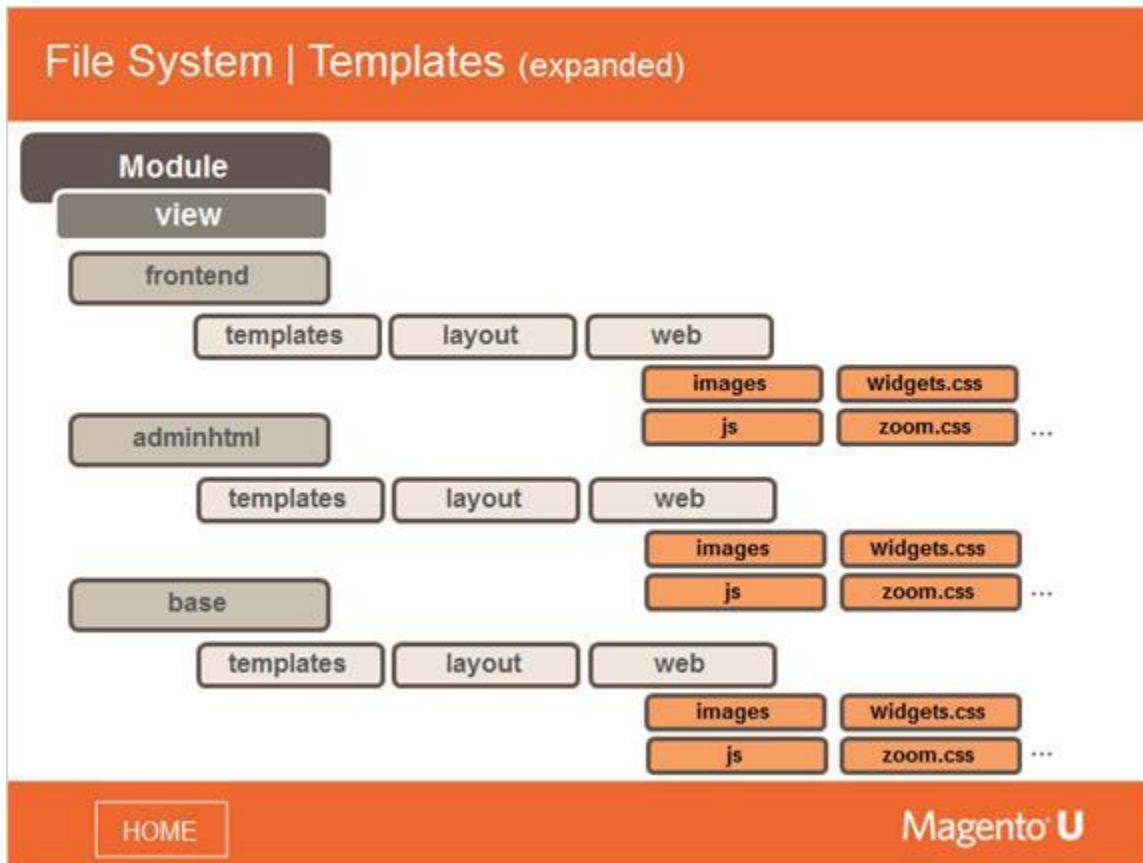
As we mentioned, modules are located in the `app/code` or `vendor` folder. All the native modules are located in the `app/code/Magento` folder.

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.

 **Note:** View elements, layout, and templates are all covered in Unit Three of this course.

Themes are also discussed to a limited extent in that unit; however, anyone wishing to really learn about themes should take Magento U's **Core Principles for Theming in Magento 2** course.

4.11 File System | Templates (expanded)

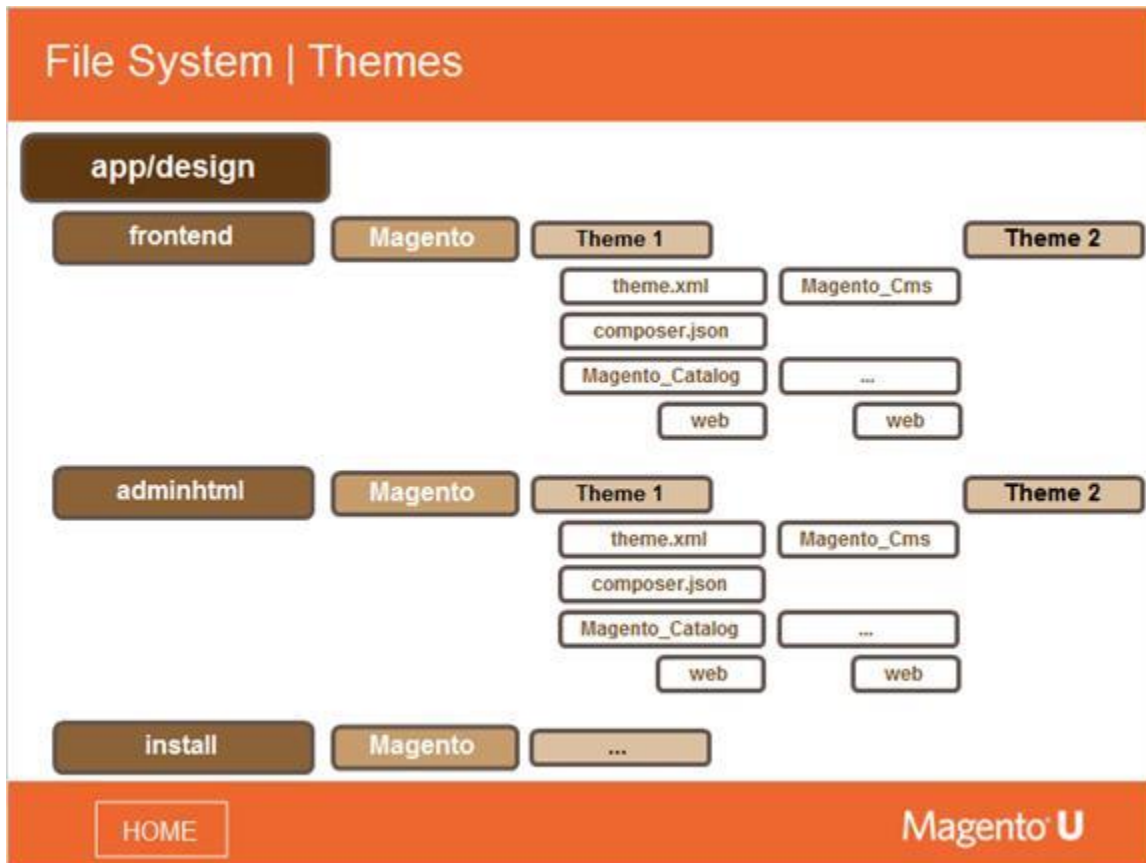


Notes:

Within the view folder are directories corresponding to design areas, namely frontend, adminhtml and base. They contain folders such as templates, layout, and web. The template folders contain the .phtml files, layout folders contain layout files, and the web folders contain files that are requested by the browser directly, like CSS and JavaScript.

From one point of view, this is a good thing because everything is within the module directory and can be easily packaged for distribution. On the other hand, there is no longer direct access for browsers to the module web page resource files, which makes things more complicated. Magento provides a special mechanism for accessing them in developer mode, and a command line tool to copy static files into the pub/ folder for production mode.

4.12 File System | Themes



Notes:

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

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.

4.13 Demo | Static Files



Notes:

In dev mode, any file available to the web can be accessed in the `pub/static.php` folder, and because it's a PHP file, it can go into the module and retrieve the file back.

However, this is not very good from a performance standpoint in a development environment. This is why a best practice is never to use dev mode in production, but instead to generate static content in dev and then use the `deployer.php` tool to deploy them to production.

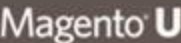
4.14 Check Your Understanding (1.4.A)

Check Your Understanding (1.4.A)

When should static files be placed in a theme rather than in a specific module?

Select your answer(s) and then click Submit.

- ☐ When the files apply to a specific application feature (ex: price box size on the product view page).
- ☒ When the files are general files that affect all the pages (ex: logo image).
- ☐ It is better to place all static files in a module rather than in a theme – it makes the overall structure better.
- ☐ It is better to place all static files in a theme rather than in a module, as having all the files in one place makes them easier to manage.

[HOME](#)


Correct	Choice
	When the files apply to a specific application feature (ex: price box size on the product view page).
X	When the files are general files that affect all the pages (ex: logo image).
	It is better to place all static files in a module rather than in a theme – it makes the overall structure better.
	It is better to place all static files in a theme rather than in a module, as having all the files in one place makes them easier to manage.

5. Development Operations

5.1 Development Operations



Notes:

In this module, we will discuss some key development operations, such as setting mode.

5.2 Development Operations | Module Topics

Development Operations | Module Topics



In this module, we will discuss...

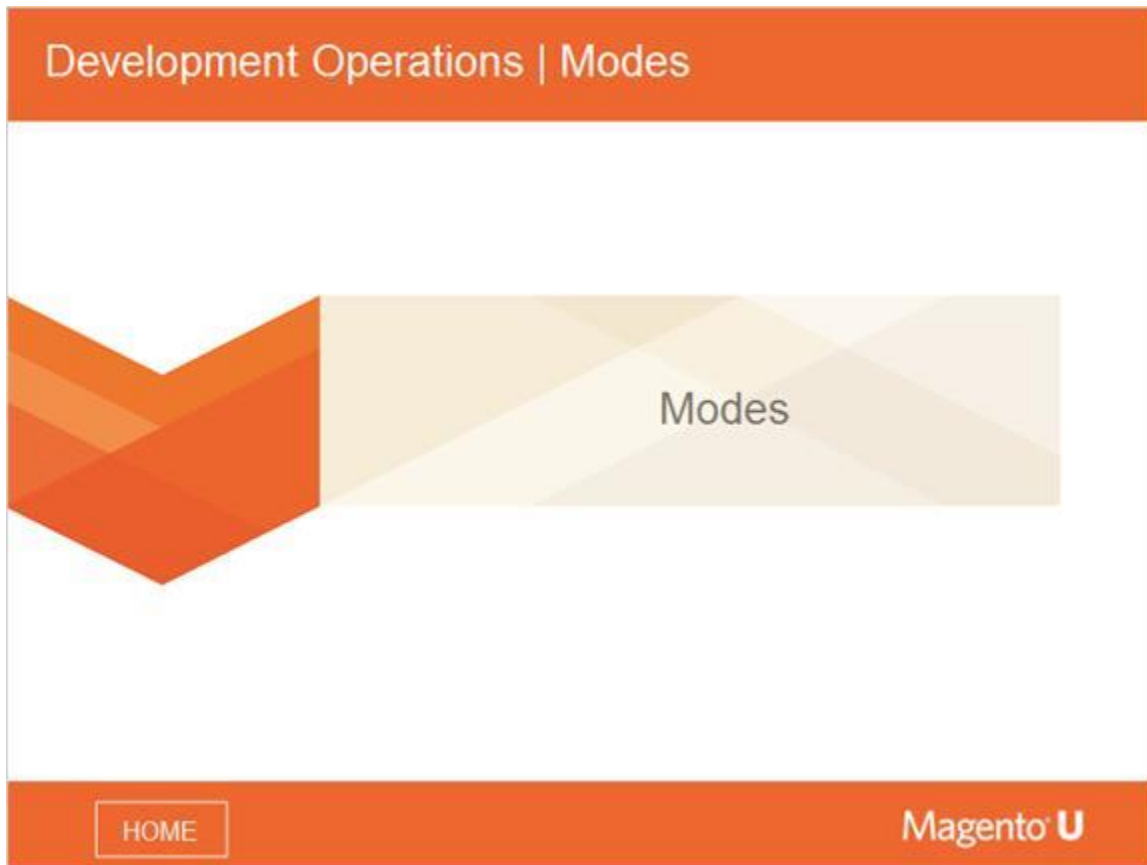
- Modes
- Command-line tools
- Caching

[HOME](#)Magento U

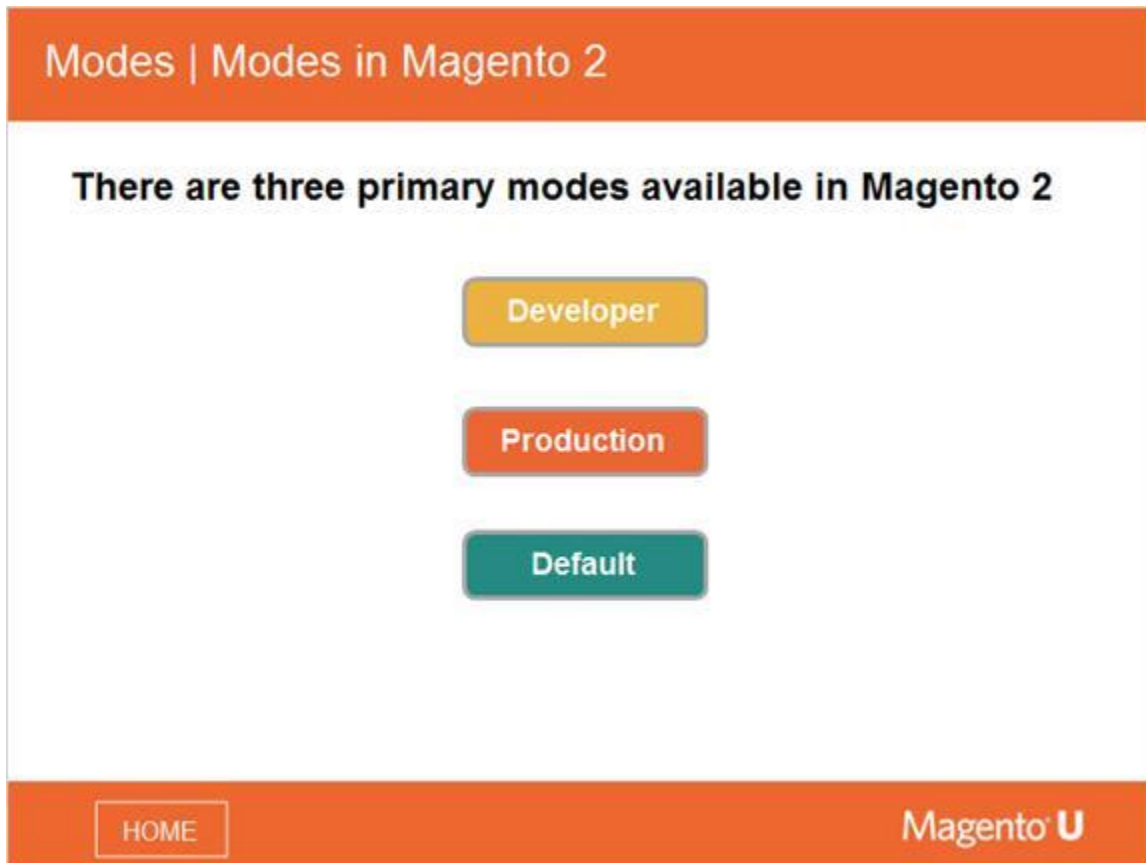
Notes:

This module will discuss the important procedures used during the development process in Magento 2: modes, command-line tools, and caching.

5.3 Development Operations | Modes



5.4 Modes | Modes in Magento 2



Notes:

Magento 2 can run in one of three primary modes -- developer, production, and default.

One of the biggest differences between the modes is how static view files get served. Static view files are CSS files, JavaScript, and images from modules that have to be processed before they can be delivered to a browser.

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

5.5 Modes | Developer Mode in Magento 2

Modes | Developer Mode in Magento 2

Developer (development phase)

- Static file materialization is not enabled.
- Uncaught exceptions displayed in the browser.
- Exceptions thrown in error handler, not logged.
- System logging in `var/report`, highly detailed.

[HOME](#)Magento U

Notes:

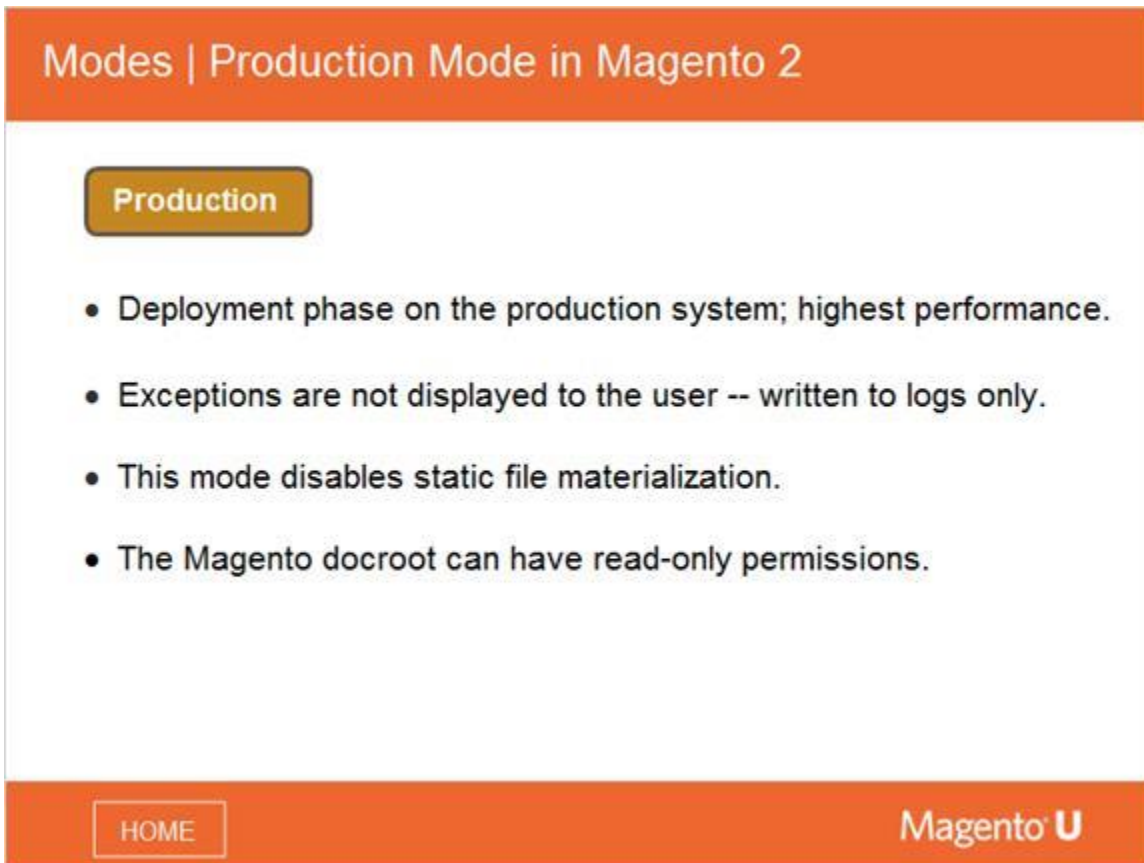
You should use the Developer mode while you are developing customizations or extensions. The main benefit to this mode is that error messages are visible to you. It should not be used in production because of its impact on performance.

In Developer mode, static view files are generated every time they are requested. They are written to the `pub/static` directory, but this cache is not used. This has a big performance impact, but any changes a developer makes to view files are immediately visible.

Uncaught exceptions are displayed in the browser, 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.

5.6 Modes | Production Mode in Magento 2



The screenshot shows the 'Modes | Production Mode in Magento 2' page. At the top, the title 'Modes | Production Mode in Magento 2' is displayed in white on an orange background. Below the title, a yellow button with the word 'Production' is highlighted. Underneath the button, there is a list of four bullet points describing the Production mode. At the bottom of the page, there is an orange footer bar containing a 'HOME' button on the left and the 'Magento U' logo on the right.

Modes | 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.

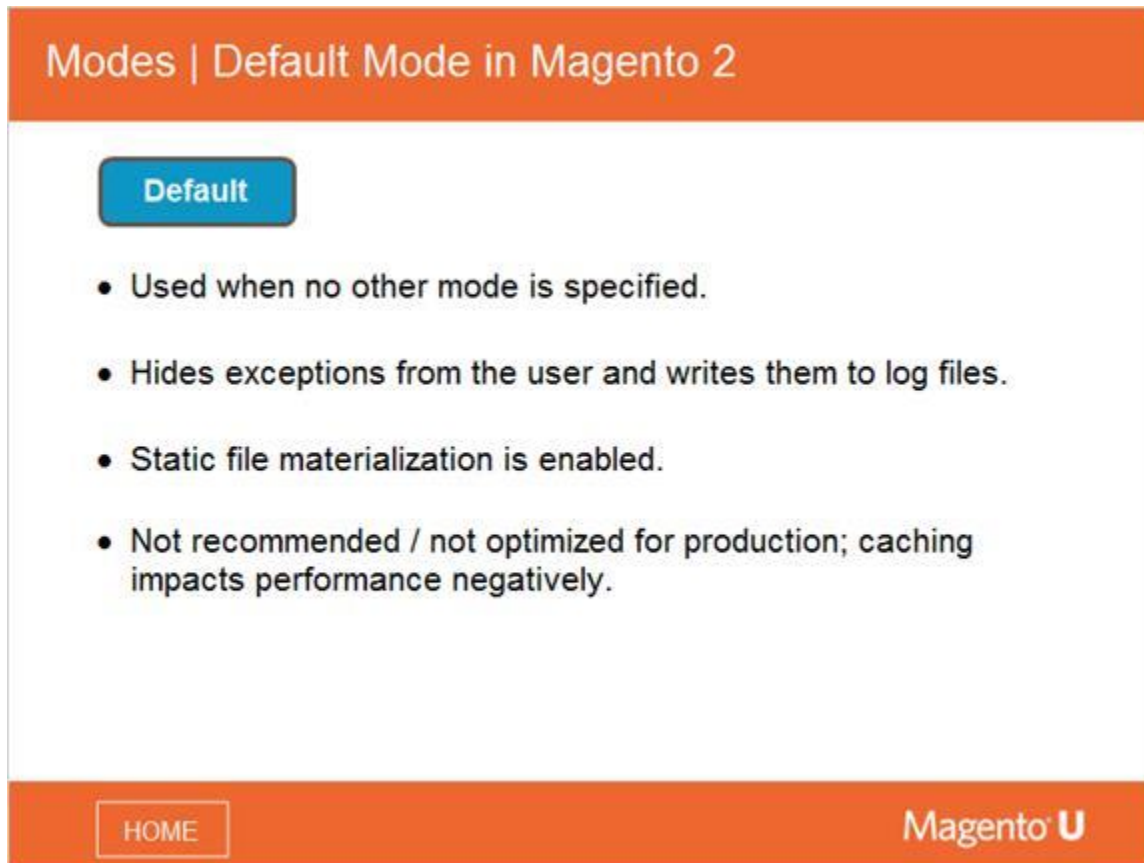
The most important aspect of this mode is that errors are logged to the file system and are never displayed to the user.

In this mode, static view files are not created on the fly when they are requested; instead, they have to be deployed to the `pub/static` directory using the command-line tool. The generated pages will contain direct links to the deployed page resources.

Any changes to view files require running the deploy tool again.

Because the view files are deployed using the CLI tool, the web user does not need to have write access. The Magento `pub/static` directory can have read-only permissions, which is a more secure setup on a publicly accessible server.

5.7 Modes | Default Mode in Magento 2



Modes | 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 files in `var/reports` and are never shown to a user. Static view files are materialized on the fly and then cached.

In contrast to the developer mode, view file changes are not visible until the generated static view files are cleared.

Default mode is not optimized for a production environment, primarily because of the adverse performance impact of static files being materialized on the fly rather than generating and deploying them beforehand.

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

5.8 Modes | Summary of Mode Features

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

Notes:

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

5.9 Modes | Maintenance Mode in Magento 2

Modes | 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 must 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.

You don't use the `MAGE_MODE` variable -- instead, you create a flag file (`var/.maintenance.flag`) to enable the mode (set to default). The method `Bootstrap::assertMaintenance()` controls this mode. When maintenance mode is enabled, the screen will show the error message "503 Internal Server Error".

You can specify a group of people to have access to the site while this mode is employed by placing the associated IPs as a comma-separated list (without whitespace) in the file `var/.maintenance.ip`.

5.10 Modes | Specifying a Mode

Modes | 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.

5.11 Modes | Specifying a Mode: Environment Variable

Modes | 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](#)

Notes:

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

The process to set an environment variable differs depending upon the web stack being used.

5.12 Demo | Specify Mode Using Environment Variable



Notes:

If you are using an Apache with `mod_php` webserver stack, you can follow along with the demo (for Nginx it is a little different).

- Open the `.htaccess` file. (Hopefully, your Apache is configured to allow setting options in `.htaccess` files).
- Set the environment variable `MAGE_MODE` to "developer" (`SetEnv MAGE_MODE 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.

5.13 Modes | Specifying a Mode: Web Server Environment

Modes | 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 web server setup is probably Apache with `mod_php`. Environment variables can be set in the main apache configuration, or in `.htaccess` files.

If you are running this type of stack, the easiest way to set a desired run mode is using the `SetEnv` configuration option, as shown on the slide.

5.14 Modes | Specifying a Mode: php-fpm Environment

Modes | 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.

On UNIX systems, the environment variable can also be set by exporting it into a shell before restarting the service:

```
export MAGE_MODE=developer
```

5.15 Reinforcement Exercise 1.5.1: Mode

Reinforcement Exercise 1.5.1: Mode

*See your course Exercises Guide for instructions
on how to complete this exercise, and its solution.*

Click "Next" when done

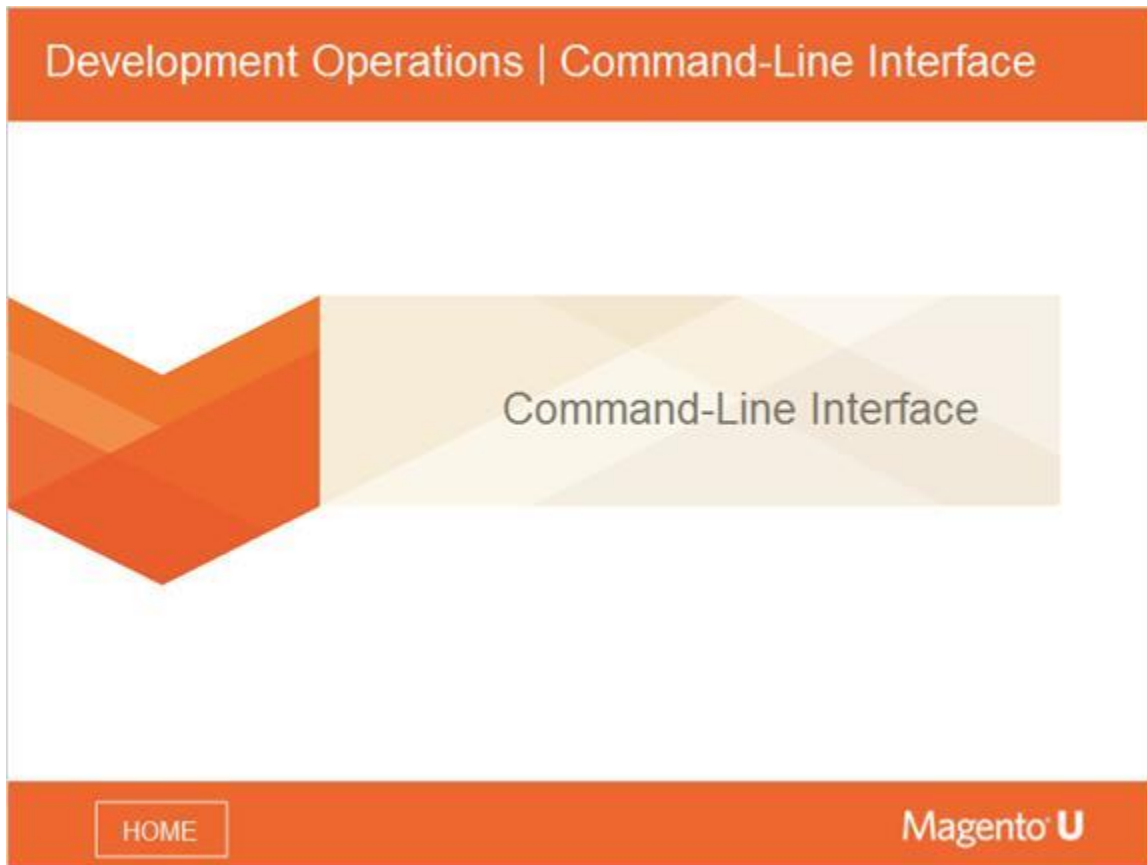
HOME

Magento U

Notes:

See your course Exercises Guide for instructions on how to complete this exercise, and the solution.

5.16 Development Operations | Command-Line Interface



5.17 Command-Line Interface | Magento 2 CLI

Command-Line Interface | Magento 2 CLI

Magento 2 Command-Line Interface (CLI)

The CLI performs both installation and configuration tasks.

- Install Magento.
- Clear cache.
- Manage indexes.
- Generate non-existent classes (factories, interceptors for plugins, DI configuration for the object manager).
- Enable/disable available modules.
- Deploy (or clear) static view files.

[HOME](#)Magento U

Notes:

Magento has one command-line interface, based on Symfony, that performs both installation and configuration tasks:
`<your Magento install dir>/bin/magento.`

The new interface performs multiple tasks, including:

- Install Magento
- Clear cache
- Manage indexes
- Generate non-existent classes (factories, interceptors for plugins, DI configuration for the object manager)
- Deploy static view files; clear static files (to comply with static file fallback rules)

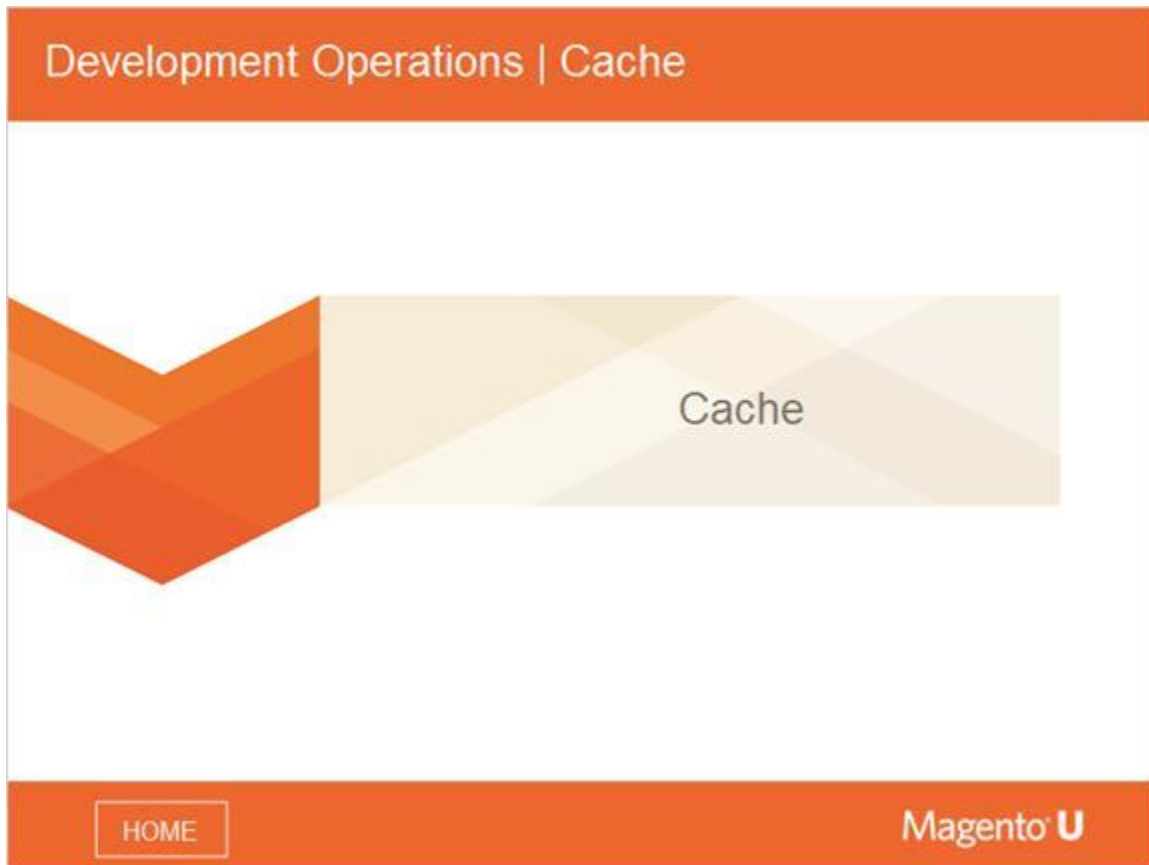
The following command will enable/disable available modules:

```
magento module:enable [-c|--clear-static-content] [-f|--force] [--all] <module-list>
magento module:disable [-c|--clear-static-content] [-f|--force] [--all] <module-list>
```

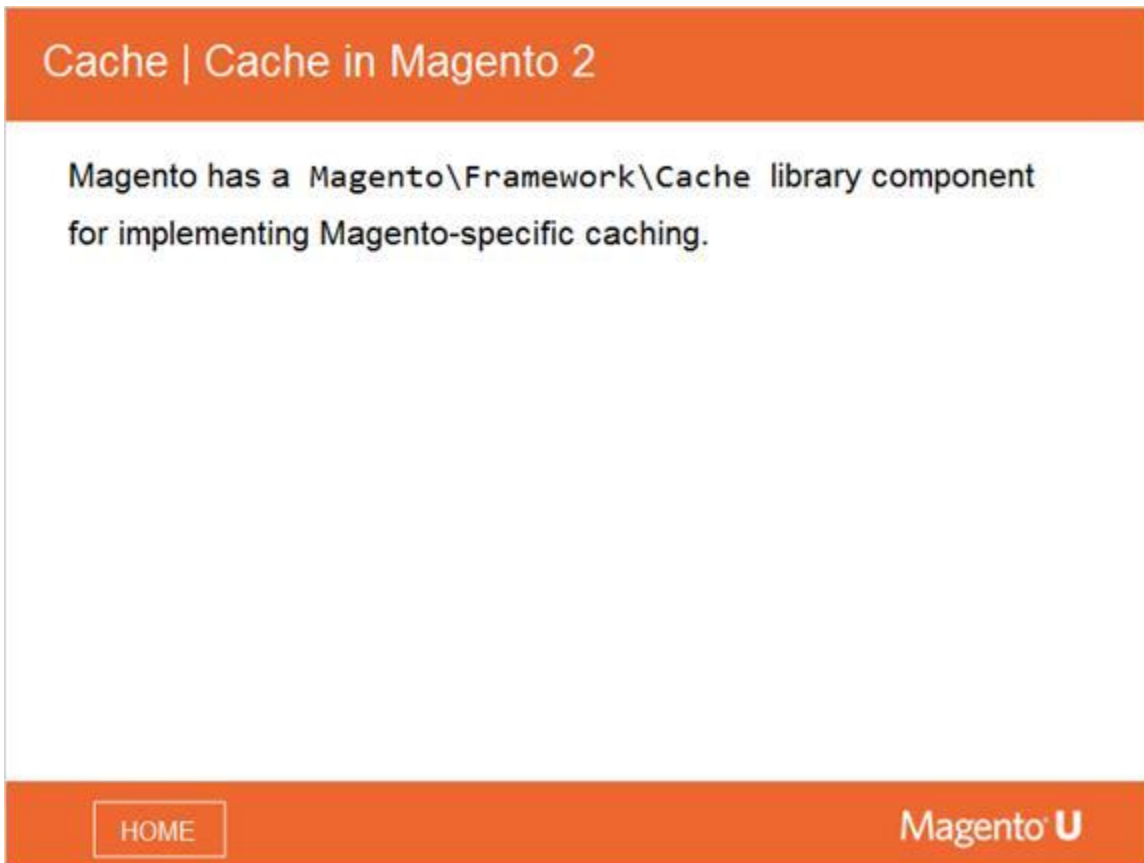
To list all available installation and configuration commands, use the command: `php <your Magento install dir>/bin/magento list`

Note that the CLI is extensible, allowing for easier integration with third-party applications.

5.18 Development Operations | Cache



5.19 Cache | Cache in Magento 2



Notes:

Caching in Magento 2 is similar to Magento 1.

In Magento 2, you use the `Magento\Cache` library component.

5.20 Cache | Cache Configuration

Cache | Cache Configuration

Magento 2 sets a default caching configuration with installation.

It is unlikely that you will want to change these settings, but if you do want to create a custom mechanism, the basic default list of cache types are located in: `app/etc/env.php`

[HOME](#)Magento U

Notes:

Magento 2 sets a default caching configuration with installation.

It is unlikely that you will want to change these settings, but if you do want to create a custom mechanism, the basic default list of cache types is located in: `app/etc/env.php`

5.21 Cache | Cache Type

Cache | Cache Type

Cache types group the cached data based on functional role.

Operations like clearing, disabling, or enabling the cache can be limited to specific cache types.

You can manage the cache via the Cache Management page in the Admin panel, or via the command-line tool `bin/magento`.

[HOME](#)Magento U

Notes:

In Magento 1, there are different cache types like Configuration, Layout XML, HTML Block Output caches, and so on.

In Magento 2, there are even more cache types -- for example, the Class Definition configuration.

5.22 Cache | Cache Cleaning

Cache | Cache Cleaning

There are three ways to clean the cache:

- From the backend (Admin).
- Using the command-line tool `bin/magento`.
- Manually removing the cache files.

To physically remove the cache files in Linux, run the following command in the Magento root folder: `rm -rf var/cache/*`

The command to clear all caches using the `bin/magento` command-line tool is: `./bin/magento cache:clean`

[HOME](#)

Magento U

Notes:

Cache cleaning in Magento 2 is similar to Magento 1.

There are three options for cleaning: using the Admin interface functionality, the `bin/magento` command-line tool, or manual removal.

It is preferable to clean the cache via the backend or the command-line tool because these processes thoroughly clean the cache, even if a non-default cache storage is configured.

5.23 Reinforcement Exercise 1.5.2: Cache

Reinforcement Exercise 1.5.2: Cache

*See your course Exercises Guide for instructions
on how to complete this exercise, and its solution.*

Click "Next" when done

HOME

Magento U

Notes:

See your course Exercises Guide for instructions on how to complete this exercise, and the solution.

6. DI & Object Manager

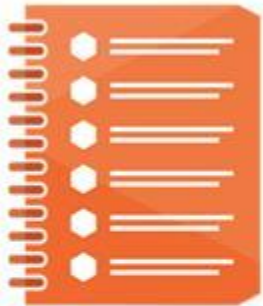
6.1 Dependency Injection & Object Manager

**Notes:**

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

6.2 DI & Object Manager | Module Topics

DI & Object Manager | Module Topics



In this module, we will discuss...

- Dependency injection
- Objects & object manager

[HOME](#)

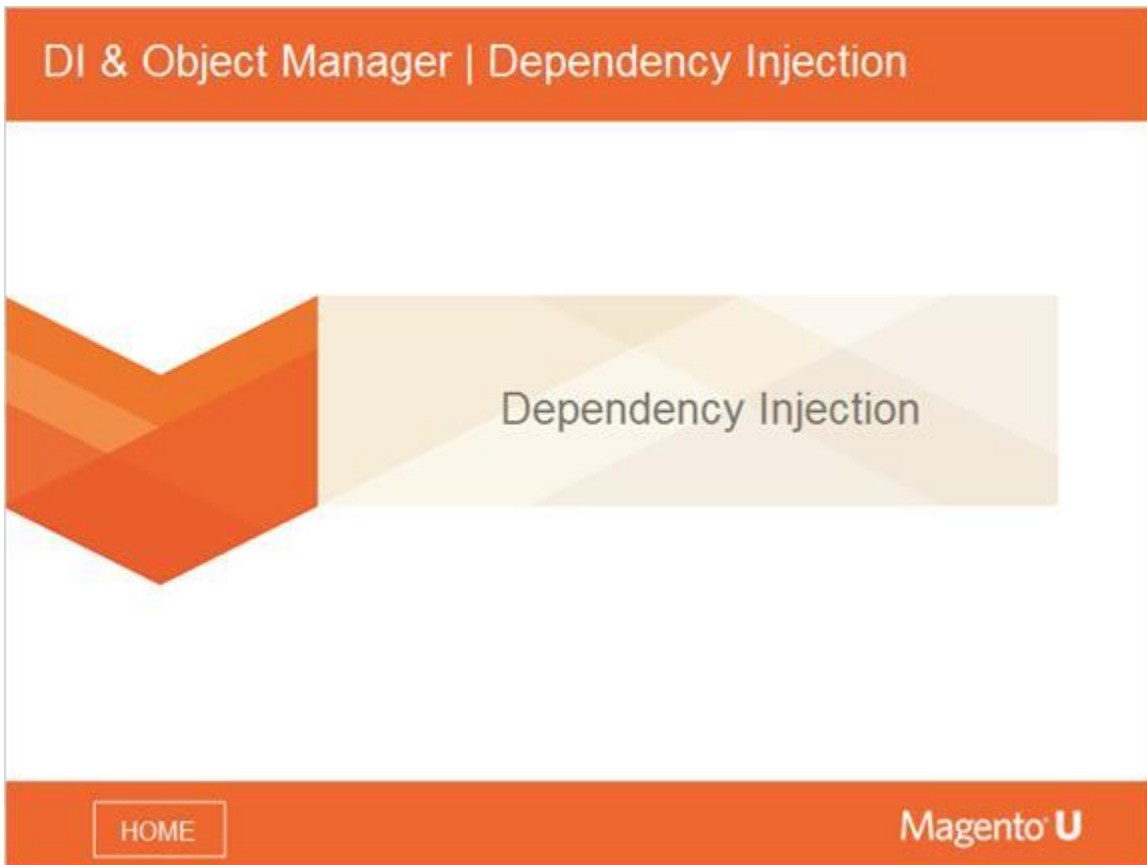
Magento **U**

Notes:

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

- Dependency injection
- Objects and object manager

6.3 DI & Object Manager | Dependency Injection



6.4 Dependency Injection | DI Pattern

Dependency Injection | DI Pattern

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

[HOME](#)

Magento U

Notes:

Dependency injection is a way to retrieve objects. In this process, classes receive objects they depend on as constructor arguments. In Magento 2, the object manager is responsible for creating all objects a class requires.

You define your class with the constructor receiving the argument objects that it needs. The dependency injection system – basically, the object manager -- will create them for you. All this is based on configuration.


Important: Dependency injection is a complex topic, which 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.

6.5 Dependency Injection | Overview

Dependency Injection | 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](#)

Notes:

Dependency injection is configuration that is XML-based, and validated by XSD.

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

Assume you need a `storeManager` instance in your `Product` class. You can declare an argument with the type `StoreManagerInterface` in the constructor of the `Product`. Now, using `di.xml`, you have to define which class will be substituted for that interface.

For that, you have two options:

1. Define a preference and it will work globally (which means that every other class that declares an argument with the type `StoreManagerInterface` will receive what you defined as the preference).
2. Assign a class in your `di.xml` for this particular object (`Product`), and the preference will **not** be global, but only apply to `Product`.

6.6 Reinforcement Exercise 1.6.1: Dependency Injection

Reinforcement Exercise 1.6.1: Dependency Injection

*See your course Exercises Guide for instructions
on how to complete this exercise, and its solution.*

Click "Next" when done

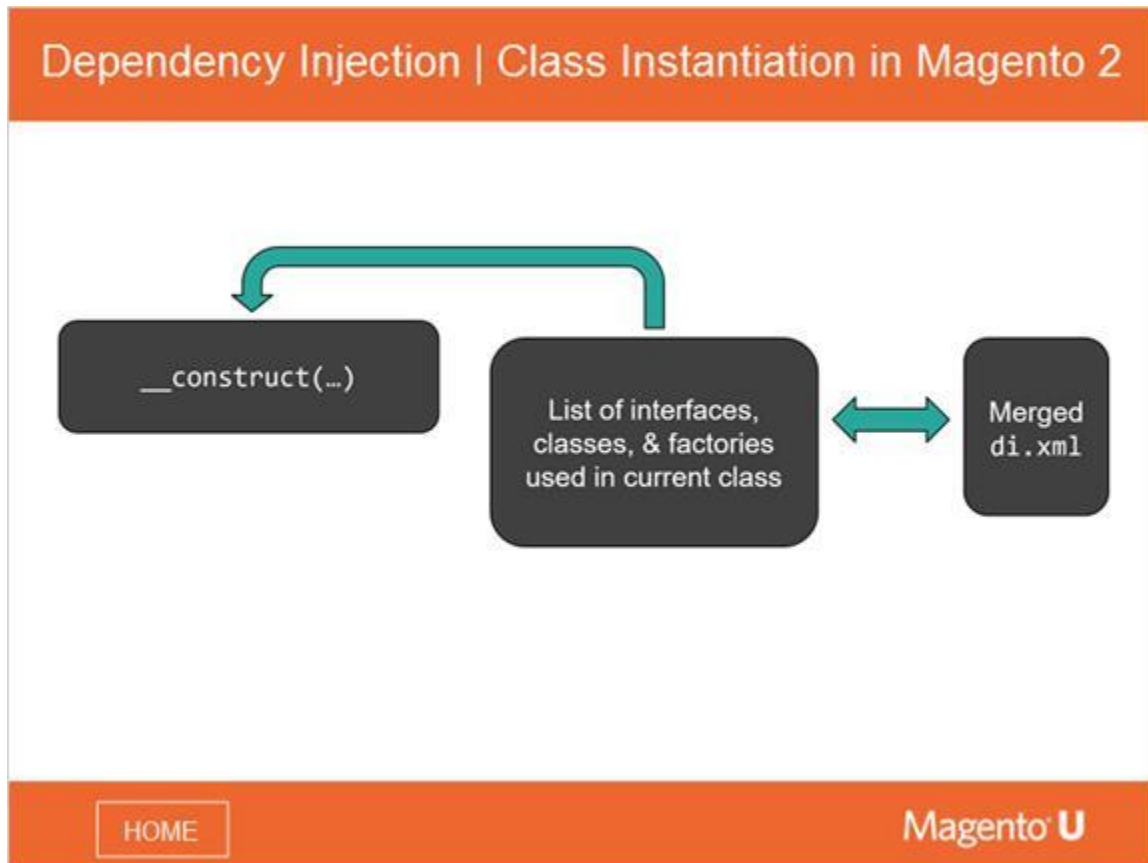
HOME

Magento U

Notes:

See your course Exercises Guide for instructions on how to complete this exercise, and the solution.

6.7 Dependency Injection | Class Instantiation in Magento 2



Notes:

This diagram shows how classes are typically instantiated in Magento.

The constructor (original) class is key. It contains a list of declarations (dependencies) that could be other classes, interfaces, classes, factories, and others.

When the original class is instantiated somewhere in the request processing flow, the DI mechanism reads its constructor and instantiates dependencies based on the configuration in the merged `di.xml` file.

This is a very powerful construction. A developer can declare a `di.xml` and change something in another module (example: adding an element to an array in the constructor of another class).

6.8 Dependency Injection | Different Classes Instantiation

Dependency Injection | Different Classes Instantiation

- Singleton-type classes... inserted by DI.
- Entry classes... created by factories or repositories.
- Factories... auto-generated classes.

[HOME](#)Magento U

Notes:

Of course, the best candidate to use with DI is a singleton-type class. By singleton-type class we mean a system class that usually exists in a single instance and is used in different places in the code. Examples could be cache, session, registry, helpers, and other classes. Note that factory or API class also matches this definition.

It is important to understand that not every class has to be injected using dependency injection -- for example, so-called entity classes like product. Which product to use really depends on the data in the database. For these classes, it is recommended to use factories for injecting.

For example, if you want to instantiate and then load the `Magento\Catalog\Model\Product` class, you would include a factory in the constructor, create a product instance using the factory, and then load it:

```
public function __construct(\Magento\Catalog\Model\ProductFactory $factory) {  
    $this->factory = $factory;  
}
```

and then later in the code:

```
public function someFunction() {  
    $product = $this->factory->create();  
    $product->load($someId);  
}
```


The example above demonstrates a best practice for Magento, to use API Interfaces for CRUD operations. We'll discuss these more in Unit 5.

Another important note -- the factory class may not exist.

When the DI mechanism identifies a class ending with factory among the parameters, and this class does not exist, it will generate such a class in the `var/generation` folder.

For our example, it would be `var/generation/Magento/Catalog/Model/ProductFactory.php`, with the `create()` method.

6.9 DI & Object Manager | Object Manager



6.10 Object Manager


Object Manager

Object manager is a class provided by Magento framework, which 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. The 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 specified through a config file and created via the Mage class. 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 variations of all four of them are implemented in Magento 1.

For all singletons in Magento 1, the registry was used, so Magento 1 created an object and put it in the registry. If you requested singletons, then it would retrieve them from the registry, creating them if not yet present.

In Magento 2, the object manager class has two methods: get and create. The get method will return a singleton object (called a "shared instance" in Magento 2) from the protected registry, while the create method will create a new instance of the given class. **This is an important difference.** In Magento 1, the class Mage was a static class. It was included at the beginning of the request flow, so you always had access to the factory methods it supplies. Now, the object manager is no longer a static or globally available.

Generally, it is best practice to avoid calling the object manager directly. Instead, you should create objects in another way. We will look at this a little later in the unit.

The object manager is concerned with the arguments that are passed in as a class's constructor parameters:

- It parses the parameter types and parameter variable names.
- It uses configuration to resolve which arguments should be used for a given parameter.
- It creates the required arguments recursively.

6.11 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/an existing instance (if it exists), 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](#)


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);
}
```

6.12 Object Manager | Object Manager Usage

Object Manager | Object Manager Usage

- To use Object Manager, include it in the class constructor:

```
public function __construct(  
    \Magento\Framework\App\RequestInterface $request,  
    \Magento\Framework\App\RequestInterface $response,  
    \Magento\Framework\ObjectManagerInterface $objectManager,  
    \Magento\Framework\Event\ManagerInterface $eventManager,  
    ...  
) {
```

- Then assign it to the protected property in constructor and use in your class:

```
$this->objectManager = $objectManager
```

[HOME](#)Magento U

Notes:

The slide shows an example of direct usage of the ObjectManager class. An object manager instance allows you to create any other class.

For example:

`$objectManager->get('Magento\Catalog\Api\ProductRepositoryInterface')` will return the same implementation of `ProductRepositoryInterface` as you would receive by including it in the constructor.

6.13 Object Manager | Magento 2 Best Practice



Object Manager | Magento 2 Best Practice

Do not use an `ObjectManager` instance in your code...

...instead, require all needed classes or interfaces in the constructor.

BEST
PRACTICE

HOME

Magento U

Notes:

Note that using `ObjectManager` in your code breaks the DI concept -- that is why it is not recommended.

6.14 Check Your Understanding (1.6.A)

Check Your Understanding (1.6.A)

- Look at the file constructors in your Magento installation at:
`Magento\Framework\View\Layout`
`Magento\Framework\View\Element\UiComponent\DataProvider\FilterPool`
- Note the following types of parameters there:
Classes * Interfaces * Factories * Arrays * Variables
- Examining these constructors should raise some questions for you.
See if you can answer any of the three questions on the next slide...

Click "Next" when done

HOME

Magento U

Notes:

6.15 Check Your Understanding (1.6.B)

Check Your Understanding (1.6.B)

- 1 How does Magento define which instance of an interface should be delivered?
- 2 When you check the file system, you notice there are almost no factory classes. So what, then, is delivered?
- 3 You look at the FilterPool class and notice it uses array elements in parameters. But it was created as a regular class by being included in other class controllers. So, how does Magento define what should be in the array?

Click "Next" when done

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

Notes:

6.16 Object Manager | Auto-generated Classes

Object Manager | Auto-generated Classes

- Some classes in Magento 2 are auto-generated – for example, factories.
- Generated code is located in the `var/generation` folder that holds the factory classes.
- Other auto-generated classes include interceptors and proxies.

[HOME](#)Magento U

Notes:

As discussed earlier, some classes in Magento are auto-generated.

Here are some other examples.

Interceptor is a class that allows the plugin functionality to work. When you require a class in your constructor, and Magento sees it has a registered plugin, Magento will then generate an interceptor with the same methods as the required class, but will call a plugin in that method. Search for interceptors in the `var/generation` folder, and take a look at their structure - note that they are all pretty similar. Interceptors use PHP traits to extend both the abstract Interceptor class and the original class.

Proxy is a tool that helps with circular dependencies, and relates to an internal implementation of how DI works. So, you won't have to deal with them directly.

6.17 Object Manager | Configuration

Object Manager | Configuration

Object Manager:

- Uses configuration from `di.xml` files to define which instances to deliver into the constructor of a class.
- Each module can have multiple `di.xml` files (global and specific for an area); all these files are merged.

See course notes for a constructor method signature example

[HOME](#)


Notes:

Magento 2 provides a new approach to working with objects. It not only encourages you to use the object manager, but also creates an object for you. Unlike Magento 1, you no longer register class groups in the configuration as there are no more special factory names. Now, you need to use the real class name -- including the PHP namespace -- when you require an instance of a class.

The creation of objects is such an important process that Magento 2 handles it automatically via the object manager and class dependencies declared as constructor parameters. The object manager creates the objects you need injected into your classes. It will recursively also create any arguments for the objects your object requires. When a developer declares an interface in the constructor, the object manager will automatically create the matching implementation for it, so you don't have to use it directly. So, when you create a class and you require an instance of an interface in this class as a parameter for the constructor, that interface is the only dependency your class will have.

You may not know what object you are getting for an interface, but if your class is programmed only using the interface, it does not need to know a class definition. Magento 2 uses preferences to define what class will implement the interface.

This is a very modern approach for the PHP community, thinking in terms of interfaces and not implementations. You can create a customization that will make Magento use your implementation of an interface 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/di.xml`
- `<core module dir>/etc/<areaname>/di.xml`

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

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

6.18 Object Manager Configuration | Specification

Object Manager | Configuration Specification

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:

Magento always uses only one `di.xml` configuration during any given request.

It includes the merged result of the `app/etc/di.xml`, each module's `etc/di.xml` file, and the area-specific `di.xml` files from every module where the area matches the current request area (for example, `frontend` or `adminhtml`).

6.19 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="urn: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 not only define preferences for interfaces, but also for regular classes. Preferences define which classes will be instantiated for a constructor argument of your class.

Example:

```
public function __construct(Magento\Catalog\Api\Data\ProductInterface $product) {
```

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

In Magento 2, it is a best practice to request only interfaces and not concrete classes. For example, in your code, you would request the `\Magento\Framework\App\Response\HttpInterface` instead of the concrete implementation `\Magento\Framework\App\Response\Http`.

The global file `di.xml` contains preferences for key interfaces.

6.20 Object Manager Configuration | Argument Example

Object Manager | Configuration Argument Example

```
<type name="Magento\Catalog\Helper\Product">
  <arguments>
    <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="tier_price_changed"
              xsi:type="string">tier_price_changed</item>
      </item>
      <item name="byDataChange" xsi:type="array">
        <item name="status" xsi:type="string">status</item>
        <item name="price" xsi:type="string">price</item>
        <item name="special_price" xsi:type="string">special_price</item>
        <item name="special_from_date" xsi:type="string">special_from_date</item>
        <item name="special_to_date" xsi:type="string">special_to_date</item>
        <item name="website_ids" xsi:type="string">website_ids</item>
        <item name="gift_wrapping_price"
              xsi:type="string">gift_wrapping_price</item>
        <item name="tax_class_id" xsi:type="string">tax_class_id</item>
      </item>
    </argument> ...
  </arguments>
</type>
```

see complete code example in course notes...

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 be used as an argument when a new instance is created.

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

In some cases, the object manager will not be able to deliver exactly the required object. For example, when you need a specific product with specific data, the class dependency can only require a generic product object - It can't provide a loaded entity. As a result, this type of object is not injectable. There is no hard distinction between injectables and non-injectables.

The question is, how can I require a specific entity? 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 the object manager, then load them, and so on. This will be discussed more in the unit about repositories.

Complete Code Example

```
<type name="Magento\Catalog\Helper\Product">
  <arguments>
    <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="tier_price_changed" xsi:type="string">tier_price_changed</item>
    </item>
    <item name="byDataChange" xsi:type="array">
        <item name="status" xsi:type="string">status</item>
        <item name="price" xsi:type="string">price</item>
        <item name="special_price" xsi:type="string">special_price</item>
        <item name="special_from_date" xsi:type="string">special_from_date</item>
        <item name="special_to_date" xsi:type="string">special_to_date</item>
        <item name="website_ids" xsi:type="string">website_ids</item>
        <item name="gift_wrapping_price" xsi:type="string">gift_wrapping_price</item>
        <item name="tax_class_id" xsi:type="string">tax_class_id</item>
    </item>
</argument>
<argument name="reindexProductCategoryIndexerData" xsi:type="array">
    <item name="byDataChange" xsi:type="array">
        <item name="category_ids" xsi:type="string">category_ids</item>
        <item name="entity_id" xsi:type="string">entity_id</item>
        <item name="store_id" xsi:type="string">store_id</item>
        <item name="website_ids" xsi:type="string">website_ids</item>
        <item name="visibility" xsi:type="string">visibility</item>
        <item name="status" xsi:type="string">status</item>
    </item>
</argument>
<argument name="productRepository"
xsi:type="object">Magento\Catalog\Api\ProductRepositoryInterface\Proxy</argument>
</arguments>
</type>

```

6.21 Demo | Dependency Injection



Notes:

Let's take a look at how dependency injection works and how classes are sent to the constructor.

First, we open the class `Product.php`, which is: `\Magento\Catalog\Model\Product.php`

It's a product model and as we look at its constructor, we can see a lot of different parameters, so it may not be clear what class will be passed to the constructor. The dependency injection mechanism is responsible for generating instantiated classes and passing them as parameters.

Now we'll look at how we can use dependency injection to affect the parameters of a class. There are basically three ways.

The first way is to define which classes correspond to certain interfaces. For example, you see here `\Magento\Store\Model\StoreManagerInterface` and `\Magento\Catalog\Api\ProductAttributeRepositoryInterface`. Each of these has a parameter.

Of course, an interface cannot be instantiated, so there should be a class that implements the interface. Using DI, we can figure out exactly which class is being used.

To do that, we need to look at the `di.xml` file, specifically `\Magento\Catalog\etc\di.xml`, and search for `ProductAttributeRepositoryInterface`.

Here you can see a line in the `di.xml` file that references `\Magento\Catalog\Api\ProductAttributeRepositoryInterface`, for

which it substitutes the `Magento\Catalog\Model\Product\Attribute\Repository` class; and this specific class is assigned to this parameter, `$metadataService`.

A second example of what can be done in the `di.xml` is to (*define*)(*assign*) a specific parameter for a specific class. In the example we've just seen, every class that requires the interface will get an instance of this class. Sometimes, though, you want to be able to define a specific instance for a specific class. For example, for `Product`, you may need a particular class that is applicable only for products. A common example in Magento is the `data` parameter. As you can see, the `data` parameter has nothing inside of it, but by using `di.xml`, we can add something to that array. You can do this for any model, which makes it a very powerful mechanism.

Here we have another class -- `\Magento\Catalog\Model\Product\ReservedAttributeList.php` -- with the type-less parameter `$productModel`. So, what will the parameter be in this case? Once again, we need to take a look at the `di.xml` file and search for `$productModel`. Here we see a declaration for the argument class `Magento\Catalog\Model\Product\ReservedAttributeList`. This argument, `productModel`, will deposit the string.

So here you can see where this string will be deposited, as `productModel`. And you can see it will do some work with that string.

The next example is for arrays. Here is a similar example to the `data` array for `Product` we saw earlier. We have an array of reserved attributes and an array of allowed attributes. Again, they are empty, but by using the `di.xml` file, for this or any other model, you can send something to these arrays. It is a very important practice in Magento 2 and a very powerful mechanism for customization.

Let's look into this a little further. We have a second argument -- reserved attributes of type array -- and it specifies an item. The item is also of type string, and the item is "position." We have a list of allowed attributes here in the `di.xml`. So, in reality, when an instance of this class is created, the string for `productModel`, which is `Magento\Catalog\Model\Product`, will be set here. The array of reserved attributes will now contain one item, "position," and the array of allowed attributes will also not be empty, but will contain three items: `type_id`, `calculated_final_price`, and `request_path`. What is especially nice is that we can create our own model, and then in the `di.xml`, add another item to the array to customize it.

Hopefully, this demo has helped you to better understand how dependency injection works in Magento 2, and how we can use it for customization via the `di.xml` file.

6.22 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 want to use the same instance of a class within several other classes.

In the example on the slide, the instance of `Magento\Indexer\Model\IndexerInterface` that is injected into `Magento\Catalog\Model\Indexer\Product\Price\Processor` will not be shared. The argument shared on the argument node is set to `false`. If it set to `true`, the same instance would be injected every time the price processor is instantiated.

It is possible to use the shared argument on both type nodes and argument nodes with the `xsi:type` set to "object."

6.23 Reinforcement Exercise 1.6.2: Object Manager

Reinforcement Exercise 1.6.2: Object Manager

*See your course Exercises Guide for instructions
on how to complete this exercise, and its solution.*

Click "Next" when done

HOME

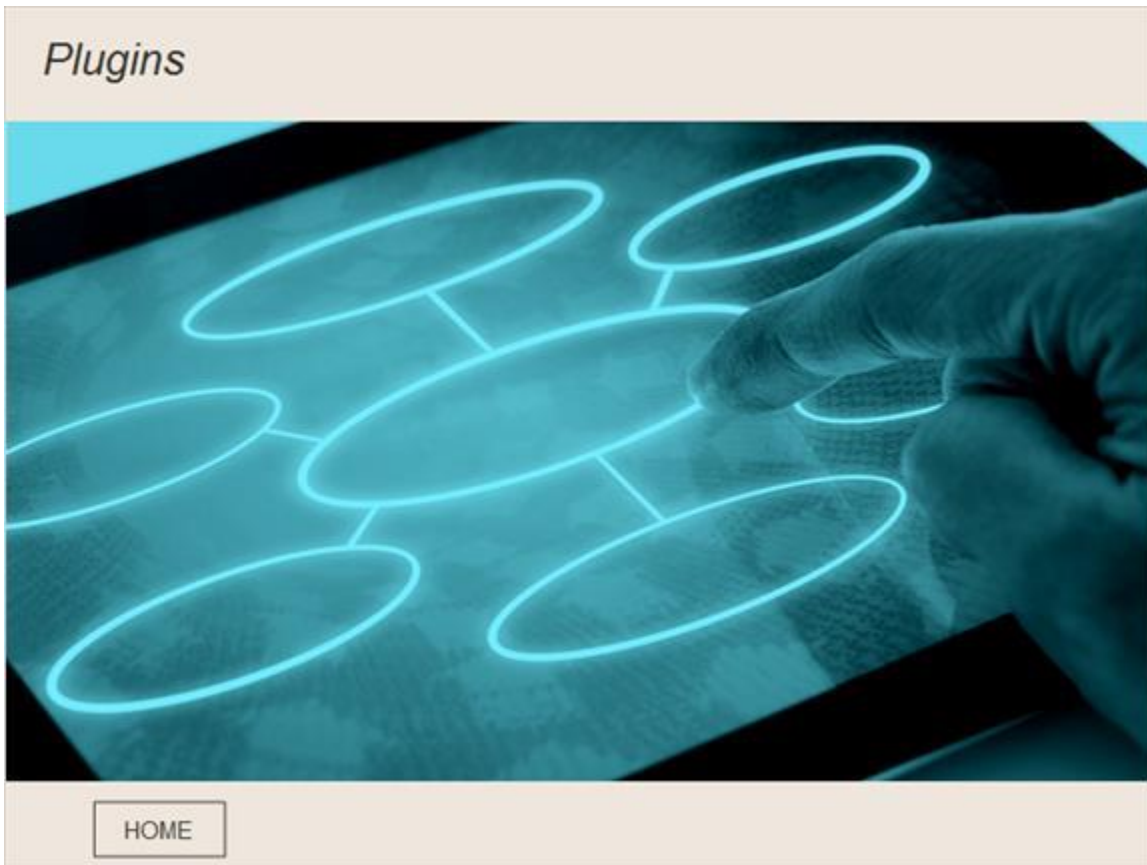
Magento U

Notes:

See your course Exercises Guide for instructions on how to complete this exercise, and the solution.

7. Plugins

7.1 Plugins



Notes:

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

7.2 Plugins | Module Topics

Plugins | 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.

7.3 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 customizing Magento simpler, and to decrease the probability of conflicts between extensions.

Plugins are used to extend (change) the behavior of any native public 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 Plugin classes to change the behavior of the target methods. Plugins change the *behavior* of the original class, but not the class itself.

You cannot use plugins for final methods, final classes, private methods, or 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.

In other frameworks, interception is known as AOP (Aspect Oriented Programming).


7.4 Plugins | Customizations

Plugins | Customizations

In Magento 2, **customizations** can be accomplished through:

- **Events:** Commonly handle external actions or input.
- **Plugins:** Allow you to customize a method. This method is basically rewrites and events at the class level.

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

[HOME](#)

Notes:

In Magento 1, there are two major ways to customize: through events and rewrites. Events are useful 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 and so on, which can hurt response time. Class rewrites are a powerful tool but require a solid understanding of what you are doing and how to fix possible issues.

In Magento 2, customizations can be accomplished through events and plugins.

Events are the implementation of a well-known development pattern, event-observer. Events are covered in detail in the next module.

Plugins allow you to customize a single method. This process is basically rewrites and events on a method level.

Plugins offer the ability to create methods that execute before a target method, to modify the arguments passed to the target method.

A plugin can also execute after the target method in order to modify the returned result. A plugin can be called before, after, or even instead of the target method. Plugins do not conflict with each other because they are executed sequentially.

7.5 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`.

The following arguments are optional and should only be specified if needed. It is a best practice to omit these arguments by default.

- **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.

7.6 Plugin Example | 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.

For example:

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

7.7 Plugin Example | 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.

For example:

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

[HOME](#)

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.

7.8 Plugin Example | Around-Listener Method

Plugin Example | Around-Listener Method

To change both the arguments and 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. For example:

```
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:

- The \$subject parameter will provide access to all public methods of the original class.
- The \$proceed parameter is a lambda that 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.

7.9 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 and 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, meaning other extensions that declare a plugin for the same method might cease to function. 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 the `sortOrder` argument.
2. The *around* listener in a plugin with the highest priority - that is, with the smallest value of the `sortOrder` argument.
3. Other *before* listeners in plugins according to the 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 `sortOrder` argument().
6. Other *after* listeners in plugins, in the reverse sort order specified for them - that is, from the greatest to the smallest value.

7.10 Configuration Inheritance & Plugins


Configuration Inheritance & Plugins

Configuration inheritance affects the order of plugin loading.

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.

Configuration files from modules loaded later can change values declared by configuration files loaded earlier. This is why specifying the sequence is important.

[HOME](#)

Notes:

Inheritance affects the order of plugin loading.

Each Magento 2 module has its own set of configuration files, gathered into the module's `etc/` directory. Depending on the needs of your module, you might have the following configuration files:

- `acl.xml`
- `config.xml`
- `di.xml`
- `module.xml`
- `webapi.xml`

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

Configuration files from modules loaded later can change values declared by configuration files loaded earlier. This is why specifying the sequence is important. Configuration inheritance involves some expensive operations in runtime; the compiler tool can be used to minimize the performance impact of these operations.

7.11 Plugins | Interception

Plugins | Interception

Interception is a concept based on the plugin functionality within the Magento core.

Object Manager checks whether there are any plugins registered for any methods of a required class, and if so, generates an interceptor class and delivers the interceptor rather than an original class.

Interceptor extends the original class, but wraps its methods to allow a plugin to be called before, after, or instead of a required method. You can find interceptors in the `var/generation/` folder.

[HOME](#)Magento U

Notes:

Interception provides the ability to observe public method calls in an application, and is therefore used to reduce conflicts among extensions that change the behavior of the **same** class or method.

Plugins can be created for any object in an application created through Object Manager. These plugins will observe method calls for that object and allow for adding behavior to these observed methods.

Multiple extensions can add plugins to the same object independently, without causing conflicts. These plugins are called sequentially, based on configured sort order.

7.12 Reinforcement Exercise 1.7.1: Plugins 1

Reinforcement Exercise 1.7.1: Plugins 1

*See your course Exercises Guide for instructions
on how to complete this exercise, and its solution.*

Click "Next" when done

HOME

Magento U

Notes:

See your course Exercises Guide for instructions on how to complete this exercise, and the solution.

7.13 Demo | Plugins



Notes:

Here is an example of a plugin. You can see it is just a class that does not extend anything, which means it doesn't have access to protected properties or methods.

The after-plugin will have the original object and result of the target method as a parameter. The plugin can now modify the result, and whatever the plugin method returns will be the new result.

The around plugin takes all the parameters that the original plugin takes.

In the `interceptor.php` file, you can see the call to the before plugin that is to be executed. Then it checks for an around plugin or executes the original method, depending on what is found.

7.14 Reinforcement Exercise 1.7.2: Plugins 2

Reinforcement Exercise 1.7.2: Plugins 2

*See your course Exercises Guide for instructions
on how to complete this exercise, and its solution.*

Click "Next" when done

HOME

Magento U

Notes:

See your course Exercises Guide for instructions on how to complete this exercise, and the solution.

8. Events

8.1 Events



Notes:

This module focuses on events and related topics.

8.2 Events | Module Topics

Events | Module Topics



In this module, we will discuss...

- Events

HOME

Magento U

Notes:

The topic to be addressed in this module is events.

8.3 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

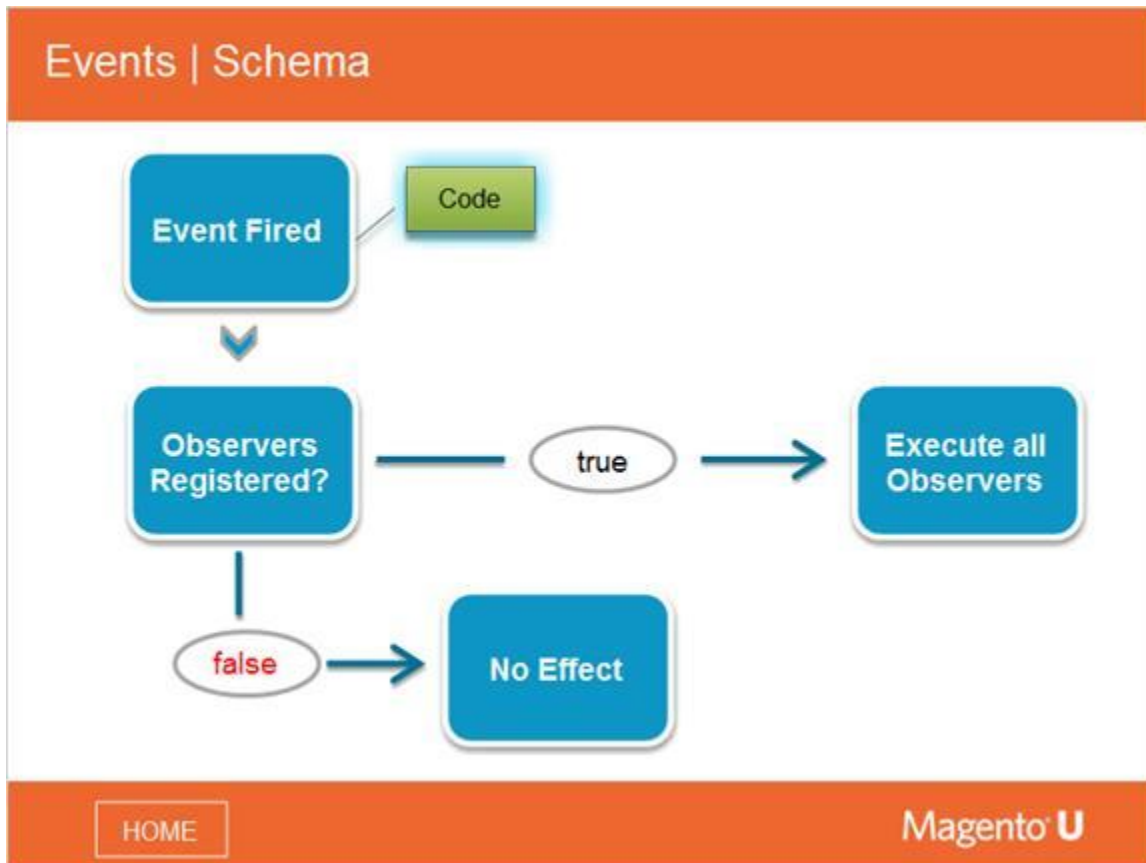
Notes:

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 a very common programming concept that works well to decouple the observed code from the observers.

Magento 1 shares the same concept, although the execution is different. If you need to trigger an event at a certain place in your code in Magento 1, it would be fired by calling `Mage::dispatchEvent()`, while in Magento 2 there is a special event manager class that fires events.

8.4 Events | Schema



Notes:

The code fires an event, and the event manager checks whether there are any observers registered.

Events can be global or registered only for a specific area, such as frontend or adminhtml. Events are declared in an `events.xml` file. If there is an observer registered, the event manager will call this observer method 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.

8.5 Events | Core Example: Saving an Order Process

Events | Core Example: Saving an Order Process

- Class `Magento\Checkout\Model\Onepage`, method `saveOrder()`, fires an event:

```
$this->_eventManager->dispatch(
    'checkout_submit_all_after',
    [
        'order' => $order,
        'quote' => $this->getQuote()
    ]
);
```

- `Magento/CatalogInventory/etc/events.xml` contains observer declaration:

```
<event name="checkout_submit_all_after">
    <observer name="inventory" instance="Magento\CatalogInventory\Observer\
        CheckoutAllSubmitAfterObserver"/>
</event>
```

[HOME](#)Magento U

Notes:

This example is taken directly from the Magento code. First, you can see how an event is fired in the code. Note its parameters: order and quote objects.

Next, you can see how an observer is declared in `events.xml`.

(continued on the next slide)

8.6 Events | Core Example: Saving an Order Process

Events | Core Example: Saving Order Process

An example from the core of the process to save an order:

- Observer class

Magento\CatalogInventory\Observer\CheckoutSubmitAllAfterObserver
and its execute() method:

```
public function execute(EventObserver $observer)
{
    $quote = $observer->getEvent()->getQuote();
    if (!$quote->getInventoryProcessed()) {
        $this->subtractQuoteInventoryObserver->execute($observer);
        $this->reindexQuoteInventoryObserver->execute($observer);
    }
    return $this;
}
```

[HOME](#)

Magento U

Notes:

And finally, an observer instance. It must implement the execute(EventObserver \$observer) method.

The \$observer object has an \$event object [available through \$observer->getEvent()], which contains the event's parameters (quote and order in our example).

8.7 Demo | Registering an Event



Notes:

Let's look at this example in the app. We'll go to `catalog/etc/frontend/events`.

We have here the event, `CustomerLogin/CustomerLogout`. The event name is what was fired in the quote. The name of the observer can be whatever you want. An instance is a class and a method processes this event.

Now let's look at the `item.php` file. In this file, we find the `bindCustomerLogin()` function. This method will be executed at the moment of firing an event. It will give access to its parameters.

In here, we have the observer object that has the `getEvent()` method. This is the same as in Magento 1. It will return data from the event object.

8.8 Reinforcement Exercise 1.8.1: Events

Reinforcement Exercise 1.8.1: Events

*See your course Exercises Guide for instructions
on how to complete this exercise, and its solution.*

Click "Next" when done

HOME

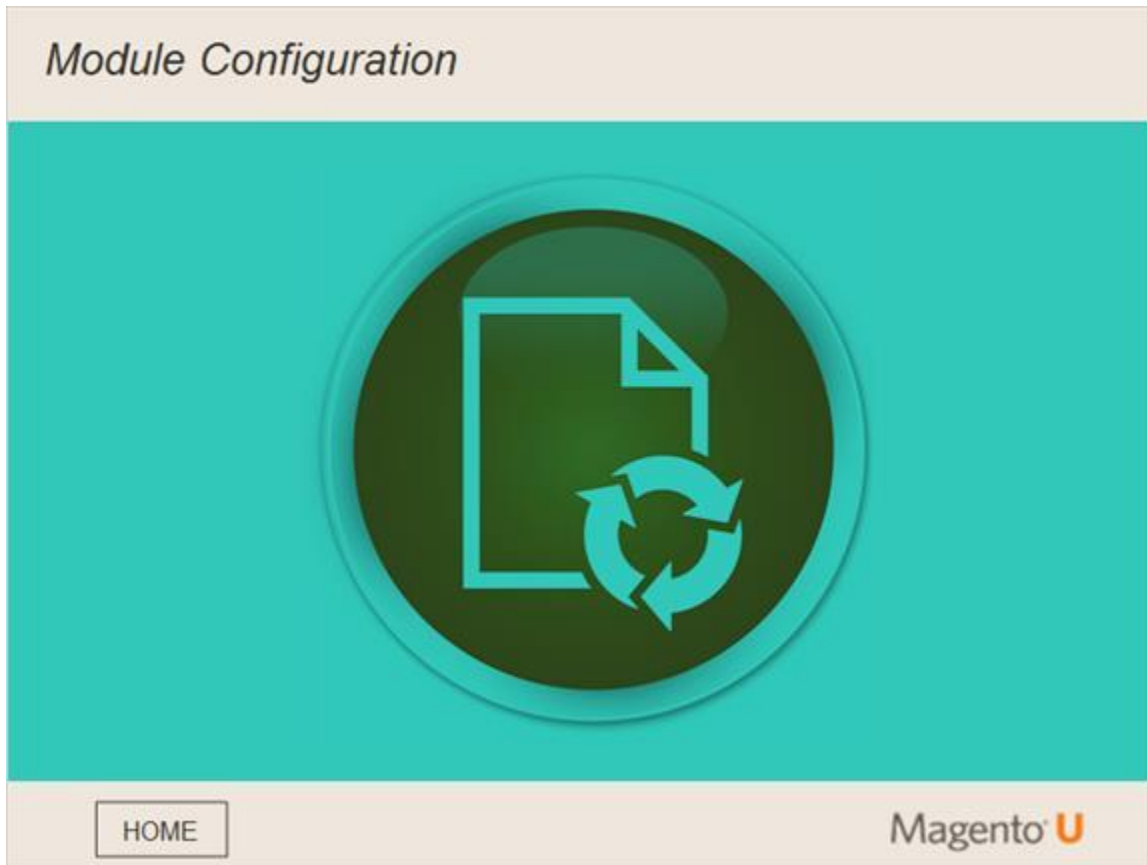
Magento U

Notes:

See your course Exercises Guide for instructions on how to complete this exercise, and the solution.

9. Module Configuration

9.1 Module Configuration



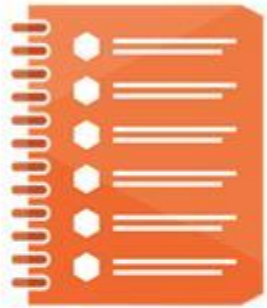
Notes:

We now move on to discuss Magento 2 module configuration. Magento 2 has made substantial changes to how configuration is handled relative to Magento 1.

Files are split by function and meaning, so you are left with a large number of XML files. Each type of config file has one specific purpose and syntax. For example, each module has a `module.xml` for module declaration, a `config.xml` for default settings of system configuration fields, an `event.xml` for events, and so on. These XML files tend to be rather small, and each file follows a very strict syntax that is validated using an XSD. This is helpful for developers because modern IDEs often offer auto completion.

9.2 Module Configuration | Module Topics

Module Configuration | Module Topics



In this section, we will discuss...

- Module configuration files
- Error reporting settings

[HOME](#)

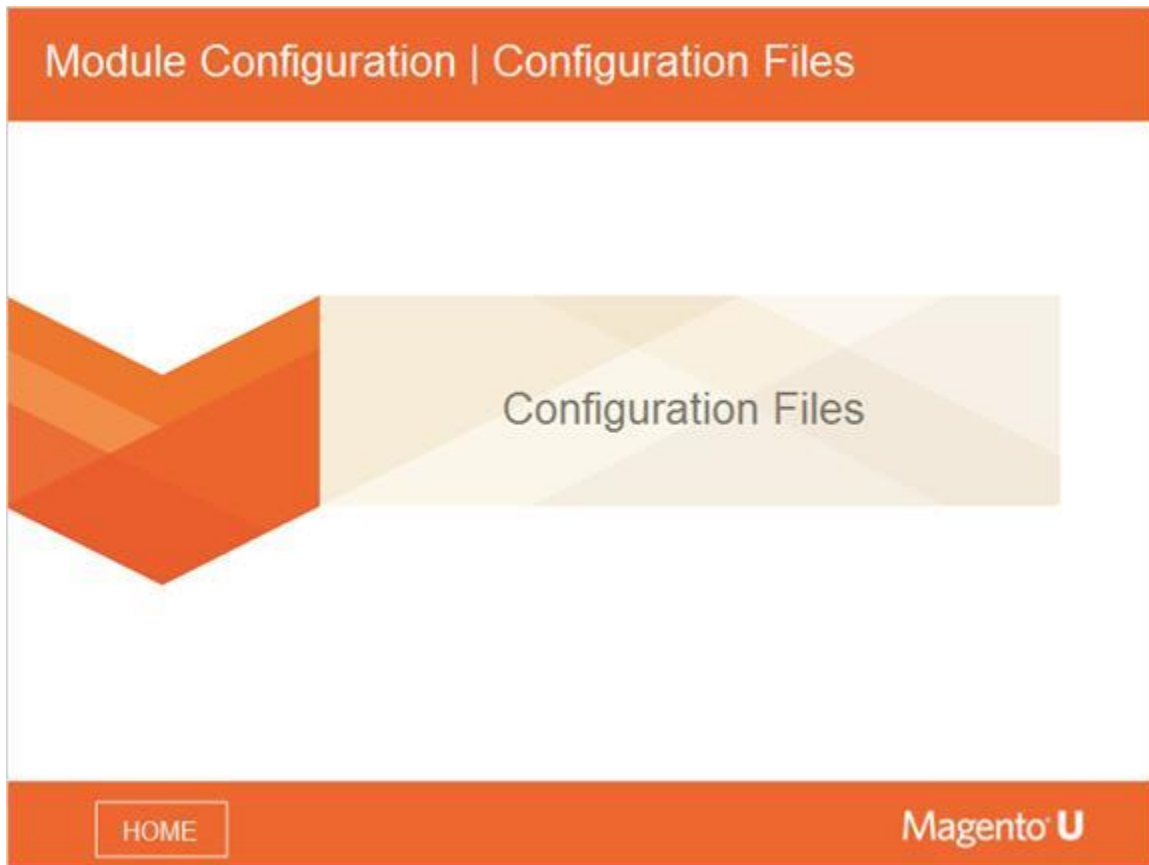
Magento U

Notes:

Within this section, we will discuss these key topics:

- Module configuration files
- Error reporting settings

9.3 Module Configuration | Configuration Files



9.4 Configuration Files | Overview

Configuration Files | Overview


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](#)


Notes:

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

`app/etc/config.php`:

- Contains declaration of all modules.
- `app/etc/env.php`:
- Contains database connection information.

`etc/config.xml`:

- Contains the default values for configuration options visible in the **Stores > Configuration** menu in the Admin panel.
- 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.
- `etc/events.xml`:
- Lists observers and the events to which they are subscribed.

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

-

9. Module Configuration

Magento loads different files and different areas separately. It has different loaders for each file type, and uses on demand loading per config file type.

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

9.5 Configuration Files | Storage

Configuration Files | Storage

There are two main ways to store configuration values:
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 values: in a database (for merchants) and in XML files (for developers).

Storing the configuration in a database (in the table `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 using the backend interface.

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`, and `routes.xml`. Magento 2 has added some important new XML files, most notably the `module.xml` and the `di.xml`.


9.6 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)

[HOME](#)


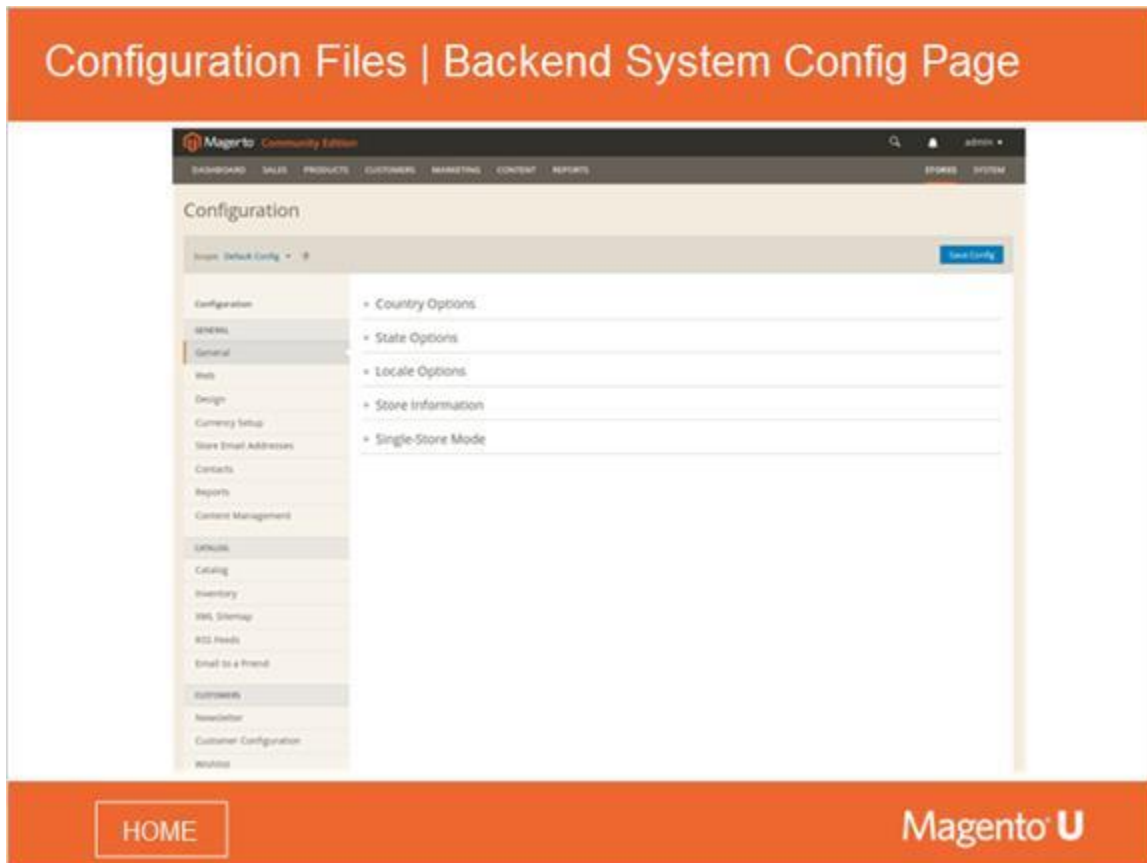
Notes:

This diagram displays the core_config_data table structure, which stores the configuration data a merchant specifies using the Admin interface.

The table core_config_data is the same as in Magento 1. It includes scope and scope_id fields. Scope can be global, website or store.

If scope is set to website, the scope_id is treated as website_id; if scope is set to store, it is a store_id; for global scope settings, it does not apply.

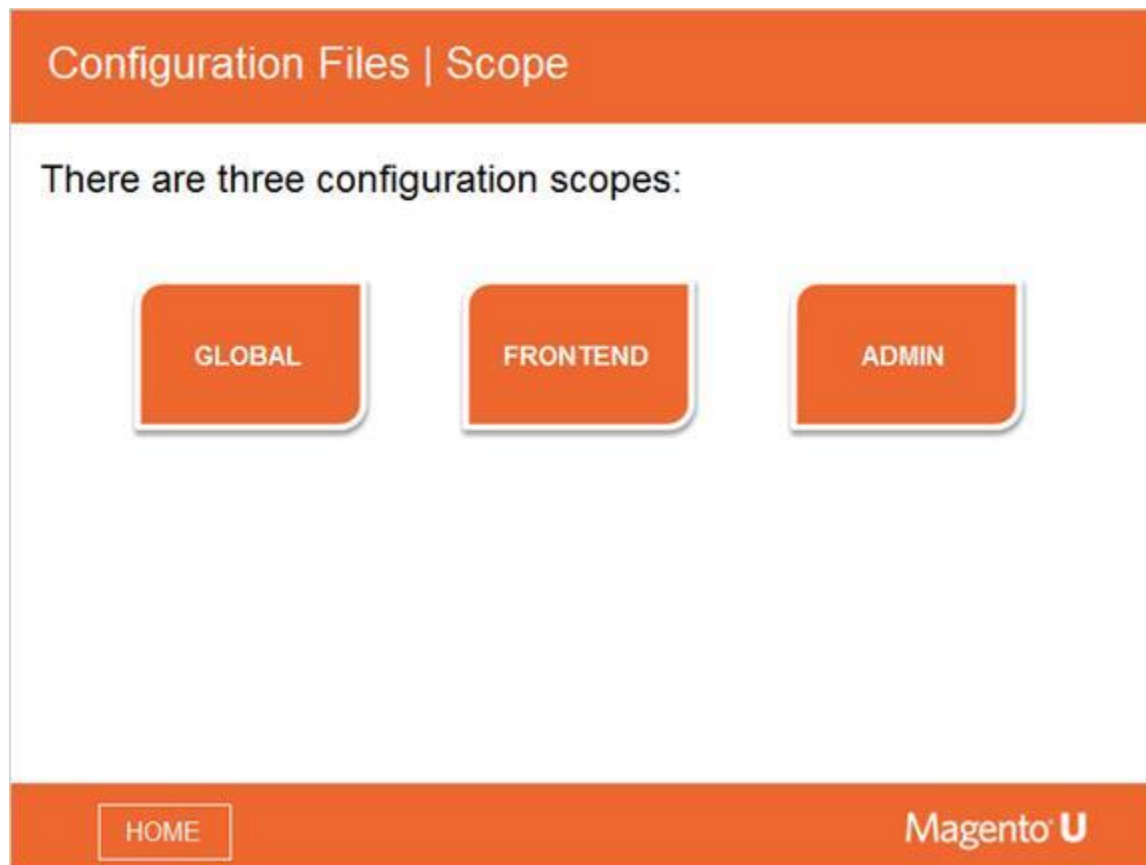
9.7 Configuration Files | Backend System Config Page



Notes:

Here is a picture of the backend (Admin) system configuration settings interface.

9.8 Configuration Files | Scope



Notes:

For 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 whereas the `module.xml` file can only be in the global scope.

If a configuration file is placed directly within the `etc/` directory of a module, its values are applied in the global scope. To specify frontend or adminhtml scope values, the configuration file needs to be placed in the folders `etc/frontend` or `etc/adminhtml`, respectively. This is a new concept compared to Magento 1, where each scope corresponded to a different branch of XML within a single configuration file.

This separation into folders with Magento 2 is an advance over Magento 1, as now the system will load only the configuration data that is needed. For frontend requests, it will not load values from the `adminhtml` folder, and vice-versa. If you need to specify values for both `adminhtml` and `frontend`, a global scope file can be used. If the same configuration file is present in both the global scope and the request scope directories, 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).


9.9 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](#)


Notes:

There are three steps in loading configuration files:

1. **Primary, system-level files:** These files are required for the application to start and installation-specific configuration (example: `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.

9.10 Configuration Files | Merging

Configuration Files | Merging

Merging of Config Files

- Nodes in configuration files are merged based on their fully qualified XPath's.
- 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](#)

Notes:

Let's now discuss the merging of configuration files.

In Magento, nodes are merged based on their XPath's and then identifier attributes. The assigned identifier must be unique for all nodes nested under the same parent node; otherwise, an error occurs.


9.11 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](#)

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 most important `*.xsd` files are located under `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`

9.12 Configuration Files | Magento\Framework\Config

Configuration Files | Magento\Framework\Config

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

The `Magento\Framework\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\Framework\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. The classes provided by `Magento\Framework\Config` should be used to introduce a new configuration type.

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

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

The configuration stored in the file system, database, or other storage options are merged according to the merging rules, and then validated with the validation schema.

9.13 Configuration Files | Magento\Config Loading

Configuration Files | Magento\Config Loading

There is a group of classes used to load XML configuration:

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 the 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.

9.14 Demo | Schema Files



Notes:

Let's take a look at how to find those XSD schema files.

Here in `lib/internal/Magento/Framework/Event/etc/events.xsd` is where we find them, and they show how `events.xml` works in Magento.

9.15 Configuration Files | Validating Configuration XML

Configuration Files | 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 file type: one for validation before and one for validation after merging. 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 ease development, it is useful to specify the relative path to the before-merge XSD file in the XML root node of an XML file. This will be used by good IDEs to provide auto-completion and validation.

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">
```

At runtime, Magento does not use the XSD specified in the XML files, but relies on the file returned by the schema locator class.

9.16 Configuration Files | Creating a Custom Config File

Configuration Files | Creating a Custom Config File

Creating a custom config file requires that you create the following elements:

- XML file
- XSD schema
- Config PHP file
- Config reader
- Schema locator
- Converter

[HOME](#)Magento U

Notes:

This slide lists elements that need to be created to customize a config file.

We'll examine each of them using the `Magento_Catalog` module and its config file `product_types.xml` as an example. `Magento_Catalog` adds a new config file, `product_types.xml`, which means any other module can now also include this file (and will be merged).

XML, XSD: Referencing the list on the slide, let's start with the `*.xsd` file. In our example, this would be `Magento/Catalog/etc/product_types.xsd`. It is too large a file to present here, but you should open and browse through it in your installation. Note there is another xsd file, `product_types_merged.xsd`, that defines the structure of the merged XML file.

Config.php: Next, you need a config PHP file (a critical file) to access data from the file. In this example, it is: `Magento/Catalog/Model/ProductTypes/Config.php`. To be able to access data from `product_types.xml`, you need to include the `Magento\Catalog\Model\ProductType\ConfigInterface` into the class constructor, and the Magento DI mechanism will deliver a `Magento\Catalog\Model\ProductTypes\Config` instance to the constructor. The interface and its implementation define which methods you can use.

Config reader: You should note that `Config.php` includes the config reader class in its constructor. That class is: `Magento/Catalog/Model/ProductTypes/Config/Reader.php`. This is very small class, with only an `_idAttributes` property and constructor. The constructor is the most important part here. Examine this code in your installation. Note that all it does is call parent, using a specific file name, converter and schema locator files.

Schema locator: `Magento/Catalog/Model/ProductTypes/Config/SchemaLocator.php` is a typical `SchemaLocator` file. It implements two methods: `getSchema` and `getPerFileSchema`, which returns paths to merged XSD and regular XSD files.

Converter: The last class is `Converter`: `Magento/Catalog/Model/ProductTypes/Config/Converter.php` for our example. The main goal of this class is to read merged XML and convert it into a convenient structure for later use.

9.17 Configuration Files | Custom Config Files Example

Configuration Files | Custom Config Files Example

Example: Magento/Catalog/etc/product_types.xml & product_types.xsd

```

<?xml version="1.0" encoding="UTF-8"?>
<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>

```

HOME
Magento U

Notes:

If you need custom configuration for a module, you should 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" encoding="UTF-8"?>
<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>

```

```
        </customAttributes>
    </type>
    <composableTypes>
        <type name="simple" />
        <type name="virtual" />
    </composableTypes>
</config>
```

9.18 Configuration | Error Reporting Settings



9.19 Error Reporting Settings | Overview

Error Reporting Settings | Overview

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:

Magento 2 uses the highest level of error reporting so that even a PHP notice will cause an exception.

Note that the way errors are displayed is determined by the mode:

- In Developer mode, errors will be returned as a stack trace, allowing you to see the exception path 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 a log file.

9.20 Check Your Understanding (1.5.A)

Check Your Understanding (1.5.A)

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

Correct	Choice
X	the .xml config file
X	a loader
X	the .xsd schema

9.21 Check Your Understanding (1.5.B)

Check Your Understanding (1.5.B)

If instead you wanted to *extend* an existing configuration type in your module, which of these three things would you *definitely* have to create?

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

[HOME](#)Magento U

Correct	Choice
X	the .xml config file
	a loader
X	the .xsd schema

9.22 Reinforcement Exercise 1.9.1: Module Configuration

Reinforcement Exercise 1.9.1: Module Configuration

*See your course Exercises Guide for instructions
on how to complete this exercise, and its solution.*

Click "Next" when done

HOME

Magento U

Notes:

See your course Exercises Guide for instructions on how to complete this exercise, and the solution.

9.23 End of Unit One

**Notes:**

Congratulations! You have completed Unit One.

You can review the material in this unit, or advance to the next course unit, Unit Two.