Magento® U

Student Guide

Fundamentals of Magento 2 Development
Unit Two

magento.com/training/overview

v.1.0

Magento® U

training@magento.com

magento.com/training/overview

magento.com/training/catalog/certification

Unit Two Contents

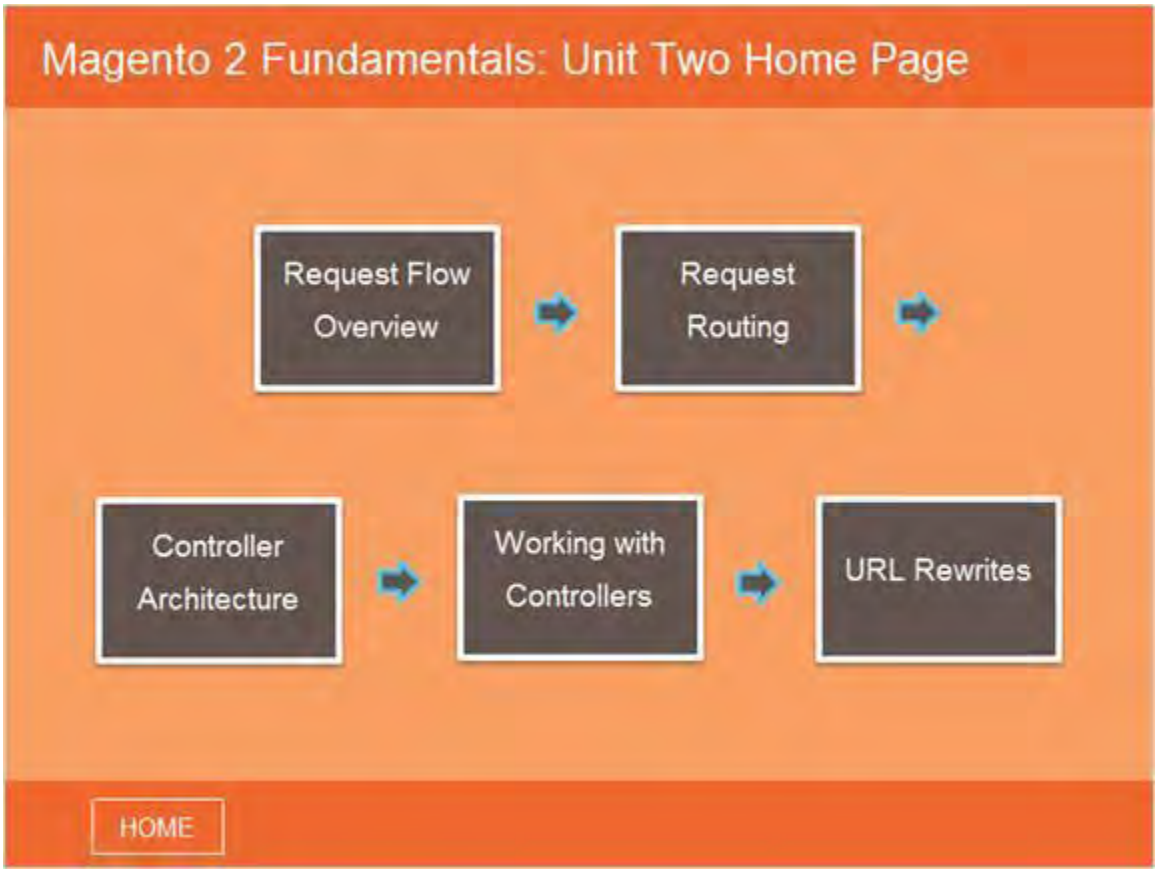
About This Guide

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 of Magento 2 Development: Unit Two

1.1 Home Page



Notes:

Unit Two of the Magento 2 Fundamentals course contains five modules.

The suggested flow of the course is indicated by the arrows. However, you are free to access any of the modules, at any time, by simply clicking the Home button on the bottom of each slide.

2. Request Flow Overview

2.1 Request Flow Overview

Notes:

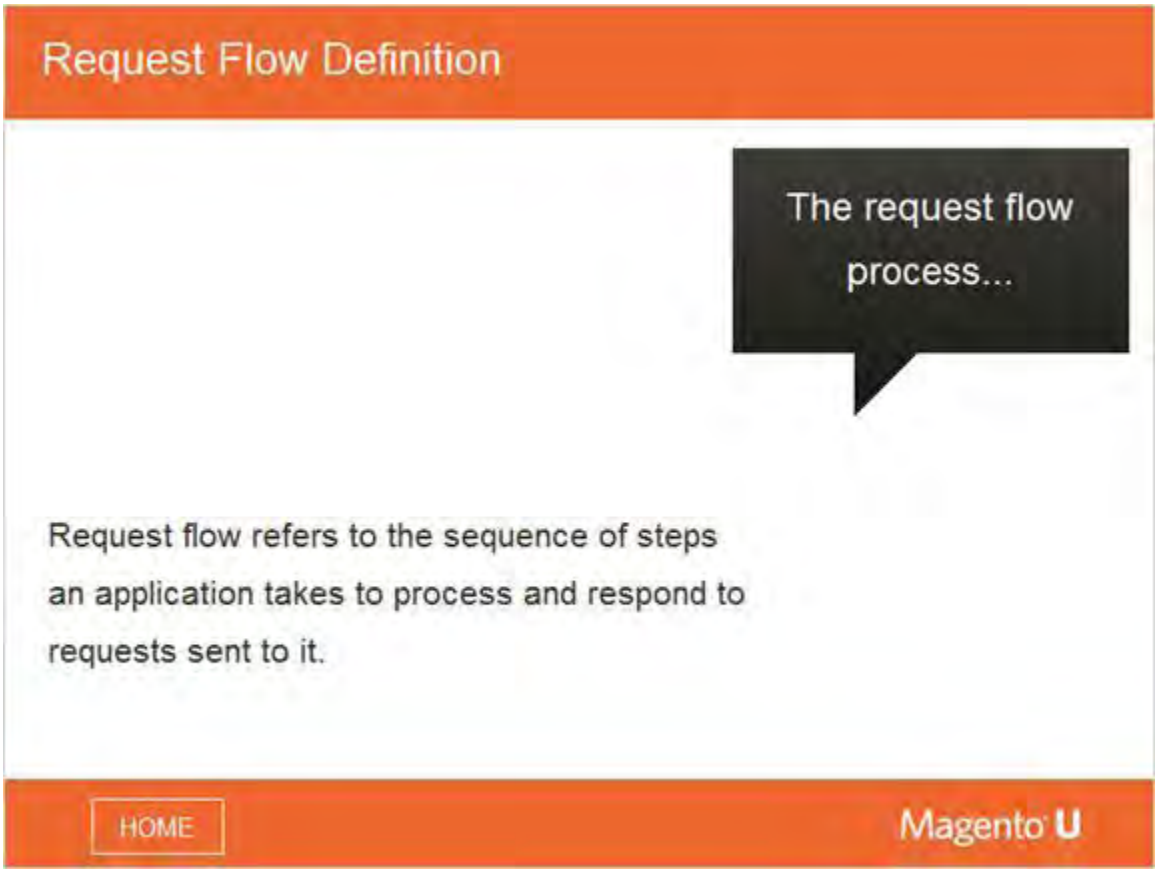
In this module, we discuss the request flow process.

2.2 Module Topics

Notes:

In this section of the course, we discuss important aspects of the request flow process, including the initialization phase, the routing phase, aspects of controller processing, and rendering and flushing output.

2.3 Request Flow Definition

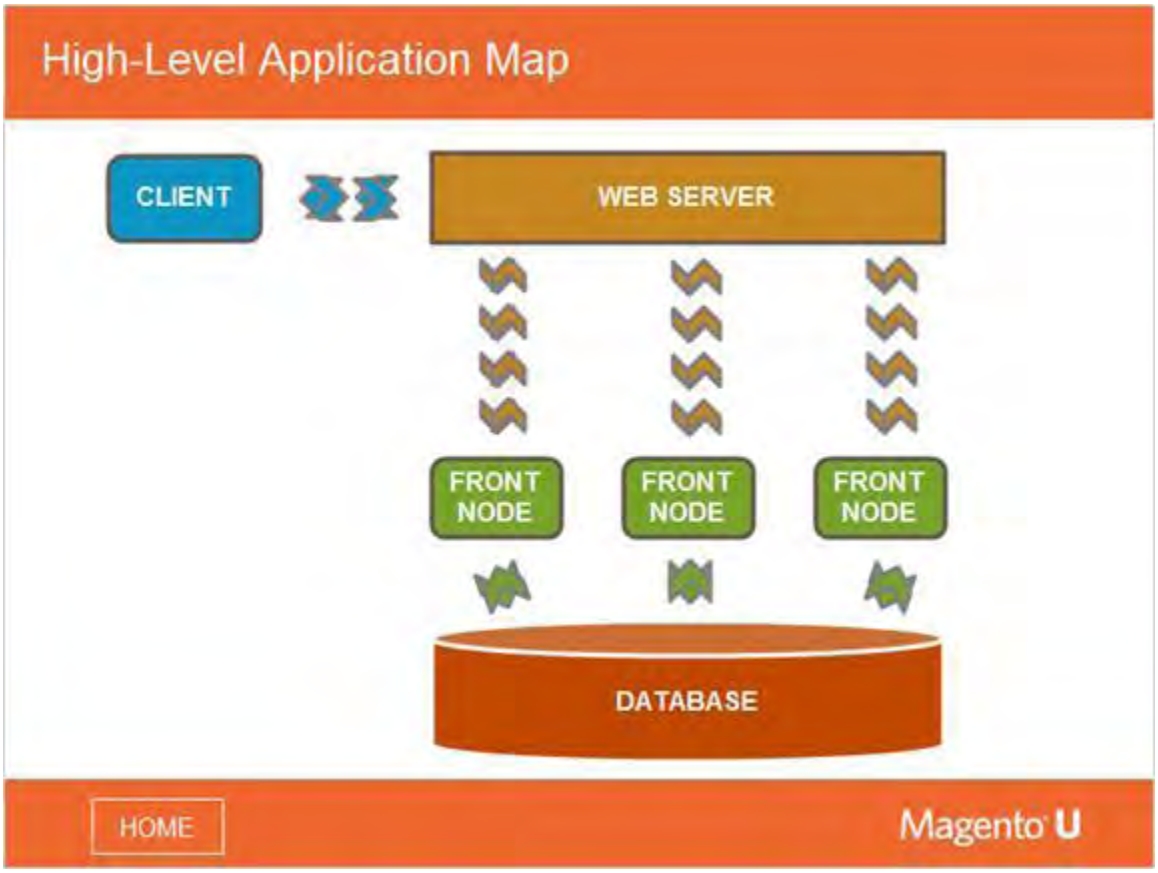


Notes:

Request flow refers to the sequence of steps an application takes to process and respond to requests sent to it.

The goal of this section is to give you a high level understanding of the steps Magento uses to receive and process these HTTP requests and then generate HTML responses back to the browser.

2.4 High-Level Application Map



Notes:

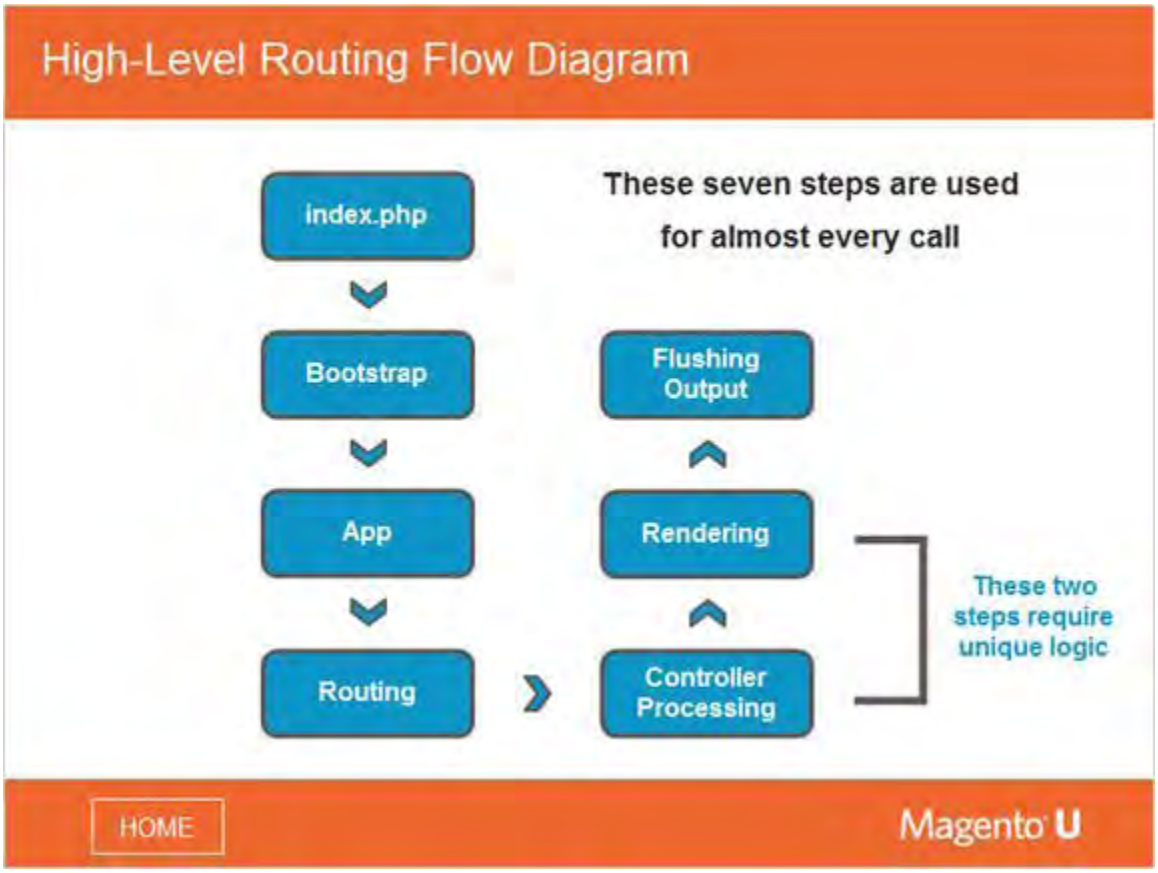
The diagram shows a high-level application map and a general client-server exchange.

The web server sends a request to the application, which is deployed into each front.

The application connects to the database and then sends back a response.

This all works through a series of calls.

2.5 High-Level Routing Flow Diagram



Notes:

For every request Magento receives, the application will take certain steps. Some of these steps are performed with every request, while others are unique to certain types of requests.

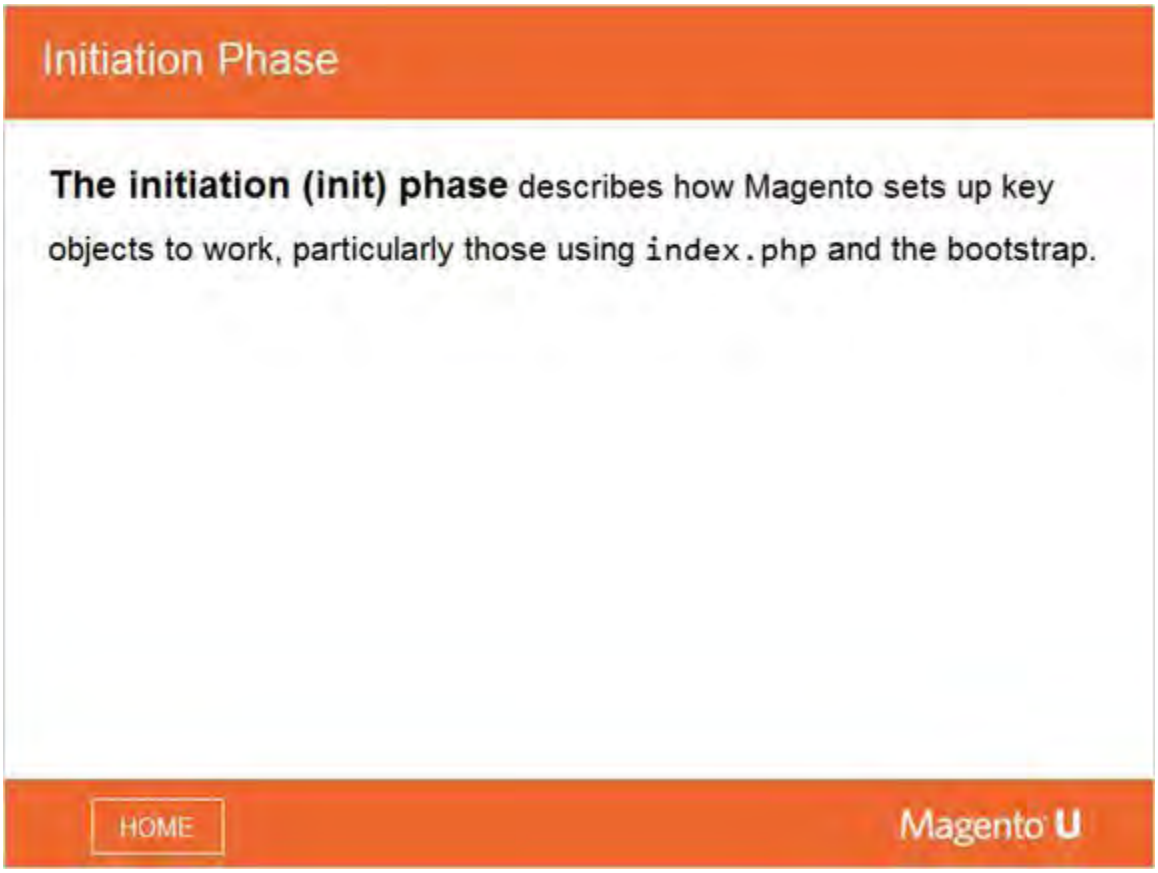
In the left column of the diagram, all the steps displayed are always performed – a request is sent to the `index.php`, which initiates the bootstrap. The bootstrap then initiates the `App` class - in our case, `Magento\Framework\App\Http`, into which Magento loads configuration.

Next, the routing process is launched, which goes through every route in the loop until it finds one to handle the current request. How the routing progresses and the loops involved are covered later in this section.

After the routing process is complete, there are two steps that are unique to every page – controller processing and rendering. In controller processing, the main goal is to find the class that will process the URL. Once defined, the class will demand specific code for this page. Then, the rendering of the containers and blocks – which are all unique to the specific page – occurs. Rendering generates HTML that is returned to the browser (flushing output).

In general terms, this is how a request moves through the different routing stages until a response is sent. While the diagram makes the process look simple, it is actually quite complex, so we will cover each of these steps in detail.

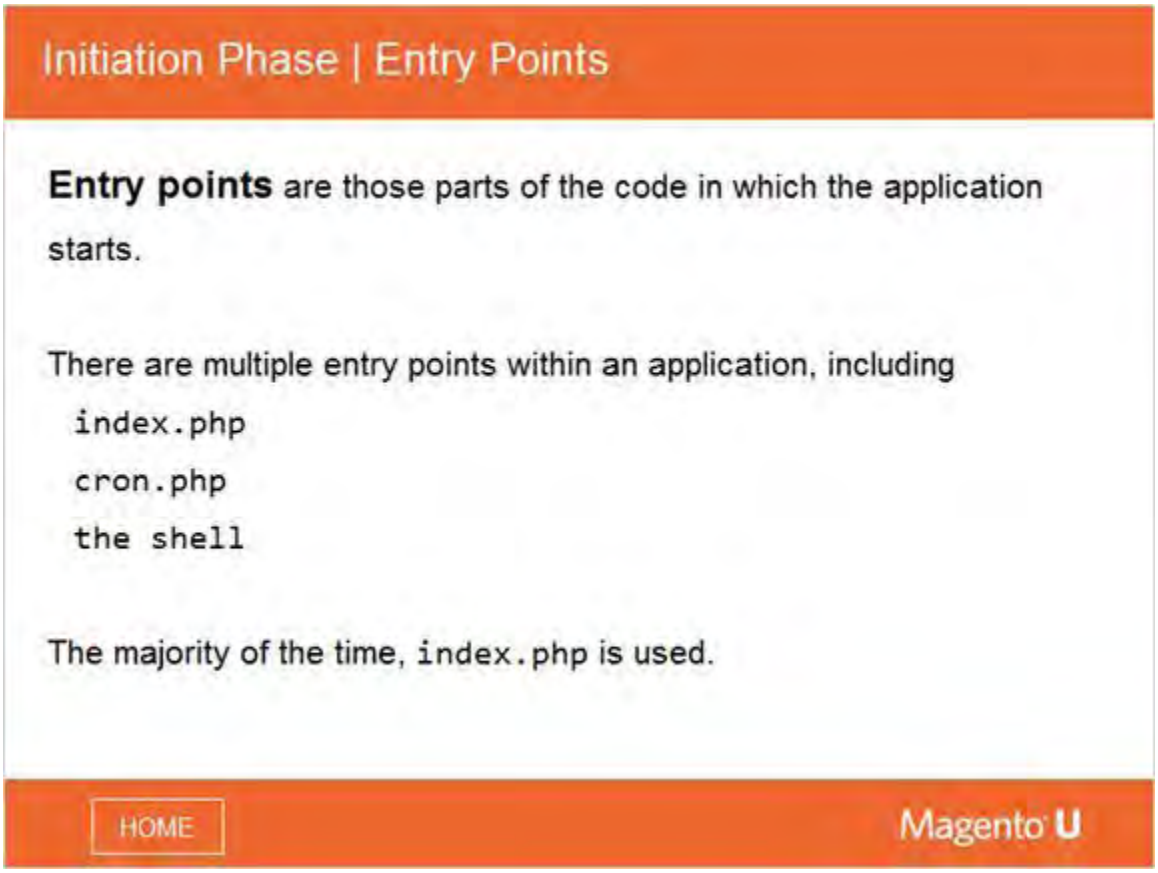
2.6 Initiation Phase



Notes:

The initiation phase describes how Magento sets up key objects to work – the bootstrap object, the app object, dependency injection, object manager, log configs, and more. These are all set up in the initiation phase.

2.7 Initiation Phase: Entry Points



Notes:

Entry point is a somewhat new concept that describes the point at which Magento starts processing a request. Magento 2's entry points are set up so that almost every request goes to the `index.php`. In some cases, they may go to `cron.php`.

You can develop shell applications in PHP and run Magento in the shell, with shell apps and extensions, but they are not fully featured yet.

2.8 Initiation Phase: index.php

Initiation Phase | index.php

The key code within `index.php` that starts application initiation is:

```
$bootstrap = \Magento\Framework\App\Bootstrap::create(BP,$_SERVER);

/** @var \Magento\Framework\App\Http $app **/

$app = $bootstrap->createApplication('Magento\Framework\App\Http');

$bootstrap->run($app);
```

HOME

Magento U

Notes:

Here is a look at the key code in `index.php` that initiates the application. This should help you to understand the concept of entry points and how you can develop your own entry point, if needed.

2.9 Initiation Phase: index.php

index.php

There are two different `index.php` files in Magento:

- one in the `<root>/index.php` [Development Mode]
- the other in the `<root>/pub/index.php` [Production Mode]

HOME

Magento U

Notes:

There are two `index.php` files in Magento – one for the development mode and one for production.

Development Mode: `<root>/index.php`

Production Mode: `<root>/pub/index.php`

2.10 Code Demonstration: index.php

Notes:

So we go to the Magento code base. Here we are.

We'll initiate a request by going to the `index.php` file. The appropriate `index.php` is the one used in the developer mode. You may have other `index.php` files in other modes – in other words, you need to use as an entry point the `index.php` file in the root, not the one in the `pub`.

As you can see, the first part of the code is similar to the code we had in Magento 1. It loads the `bootstrap.php` file, which requires `autoload.php`. Note that `autoload.php` registers the autoloader. The autoloader is very important because it will load all your classes. Magento has a couple of autoloaders but the main one is similar to the one we had in Magento 1 – it takes the path to the class and converts it to the physical path to the file. In Magento 2, we don't have a factory method anymore – that's why it calls all classes by full names, making the process a bit easier.

In the `index.php` folder, Magento loads the `bootstrap.php` file and it includes `autoload.php` and initiates autoloader. Then it goes to the bootstrap::create(). If we go deeper in the bootstrap program, we see that bootstrap has the `ObjectManager` factory. It is the factory that creates the object manager.

Back in the `index.php` folder, the following line creates the application:

```
$app = $bootstrap->createApplication('Magento\Framework\App\Http');
```

The `app/http.php` has the ability to have multiple apps for different situations. One for php requests, one for cron requests – potentially it could have one for shell requests. We'll look at this part of the code in some detail.

As you can see, the `app/http.php` folder implements the `AppInterface`.

Looking at the `appinterface.php`, you have simple code so it's very easy to implement if you'd like to develop your own app Interface.

Note that the interface does not have a constructor at the moment but most likely you'll have to include a constructor. You find a constructor in the `app/http` folder.The app constructor looks a lot like the DI constructor – it has many classes.

Bootstrap initiates `objectManager` first, and it makes `objectManager` create an application.

`PublicFunction::launch()` is the analog of $major- run() and $app- run() methods in Magento 1. We see in detail the launch method using the DI. So, in Magento 2, the launch method takes `arealist`, which is a parameter. This parameter defines the DI and it will be affected by some variables. In other words, we have classes and we use the constructors to define objects that we want to process and then the object manager will create the objects for the appropriate class; we can then use the created objects.

The idea is to declare objects that you want in the constructor. The object manager will generate them for you, and then you can affect them when new modules are involved. For example, in this class the object will be something different or you can say, for every class the object will be something else, by redefining the preferences in your interface. Another example: if it's an array, then you can say, "I'd like to add something else to the array."

Let's take a look at `arealist.php` folder. As you can see, this line emphasizes the example I just gave you. The module populates areas through the arrays.

Going back to our `http.php` folder, you can see that it loads configurations for areas. In addition it loads modules. We have seen the list of loaded modules in a previous topic. After, it will create the front controller and front controller dispatch. It will find the routing, the controller, and allow rendering. Then it will return a result for this request. In the next line, it returns an instance of the result interface, then does something with this – otherwise, the `httpinterface`. Then we can have the exception, which is a result that is totally wrong. We will analyze this in the controller section.

So what should the controller return? You can see the controller returns several results. This will be seen in the controller section a little bit later. Note that some controllers don't return anything, which is the case of the second one here – it doesn't return anything. It returns a response object.

We have any event front controller, which sends the response. We'll see later how the response is returned and where it goes.

To summarize, the launch class accomplishes several important tasks. The first major phase is to upload `objectManager` and configuration. Then it does the application launch. The application launch creates the frontend controller.

2.11 Initiation Phase: Application Execution

Initiation Phase | Application Execution

Execution starts from the initiation phase:

```
$this->_objectManager->configure($this->_configLoader->load($areaCode));

initObjectManager()

private function initObjectManager()
{
     if(!$this->objectManager) {
        $this->objectManager = $this->factory->create($this->server);
        $this->maintenance = $this->objectManager->get(Magento\Framework\App
                                                        \MaintenanceMode);
     }
}
```

The `factory->create()` method is where DI gets initialized.

HOME

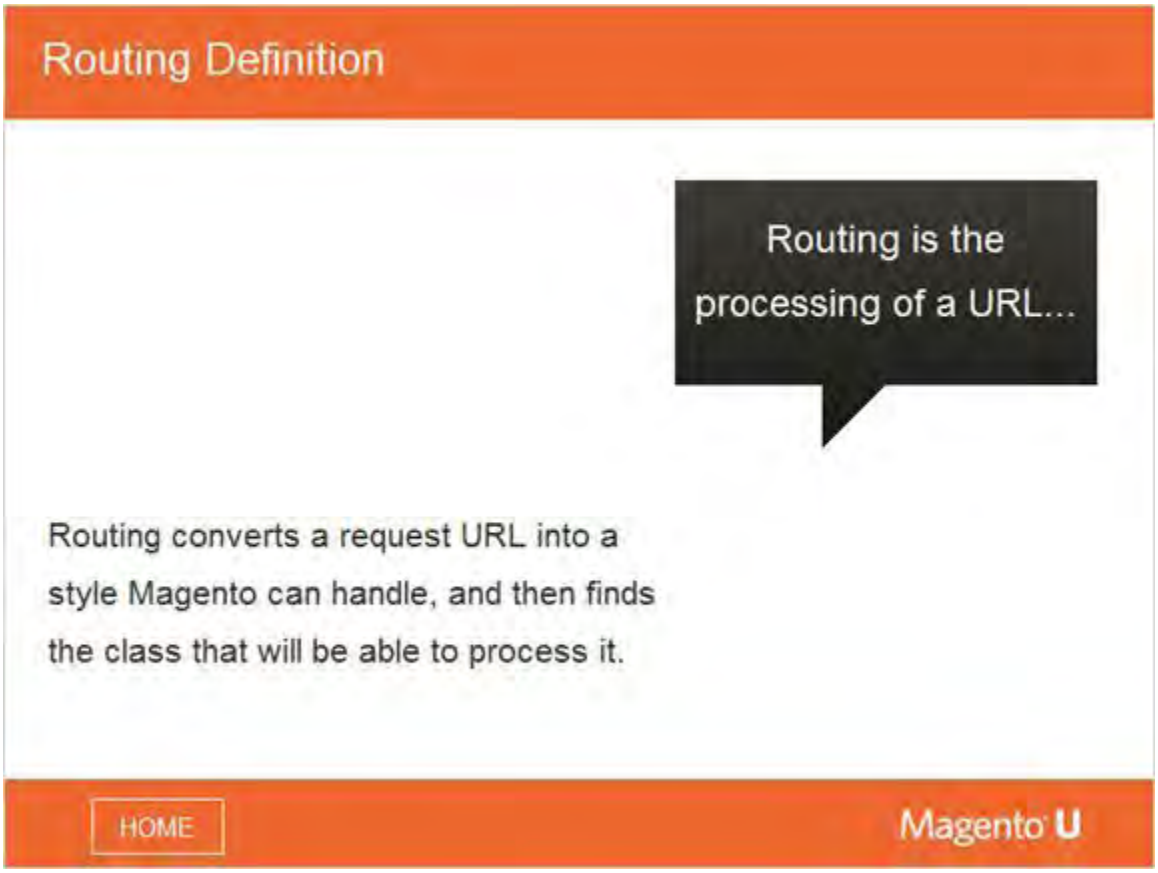
Magento U

Notes:

This example displays the key code where execution starts.

In the code, you see how `objectManager` is created by the `ObjectManagerFactory::create()` call. It is important to note that all DI initialization happens in the same call.

2.12 Routing Definition



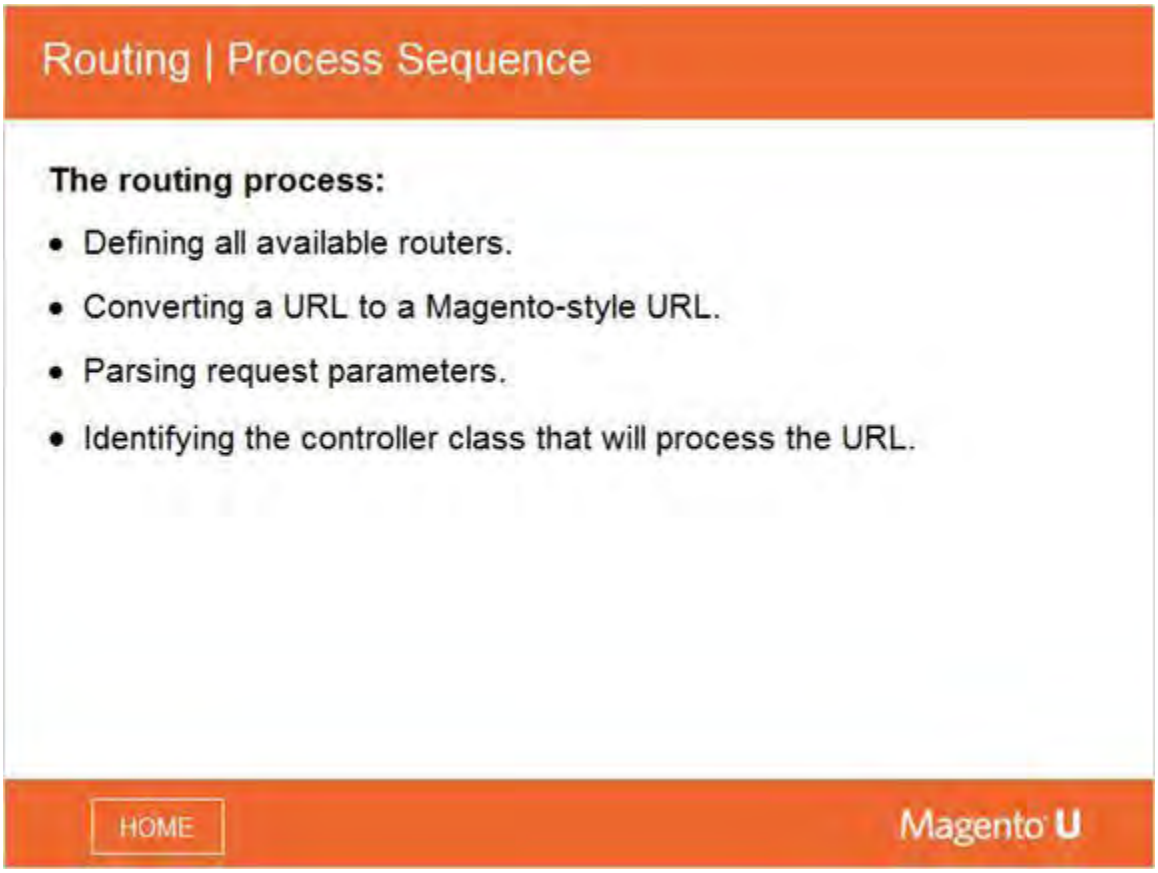
Notes:

Routers are classes responsible for recognizing different types of URLs.

Magento 2 is more consistent than Magento 1 in this process of defining the classes that process the URLs.

Routing converts a request URL into a style Magento can handle, and then finds the class that will be able to process it.

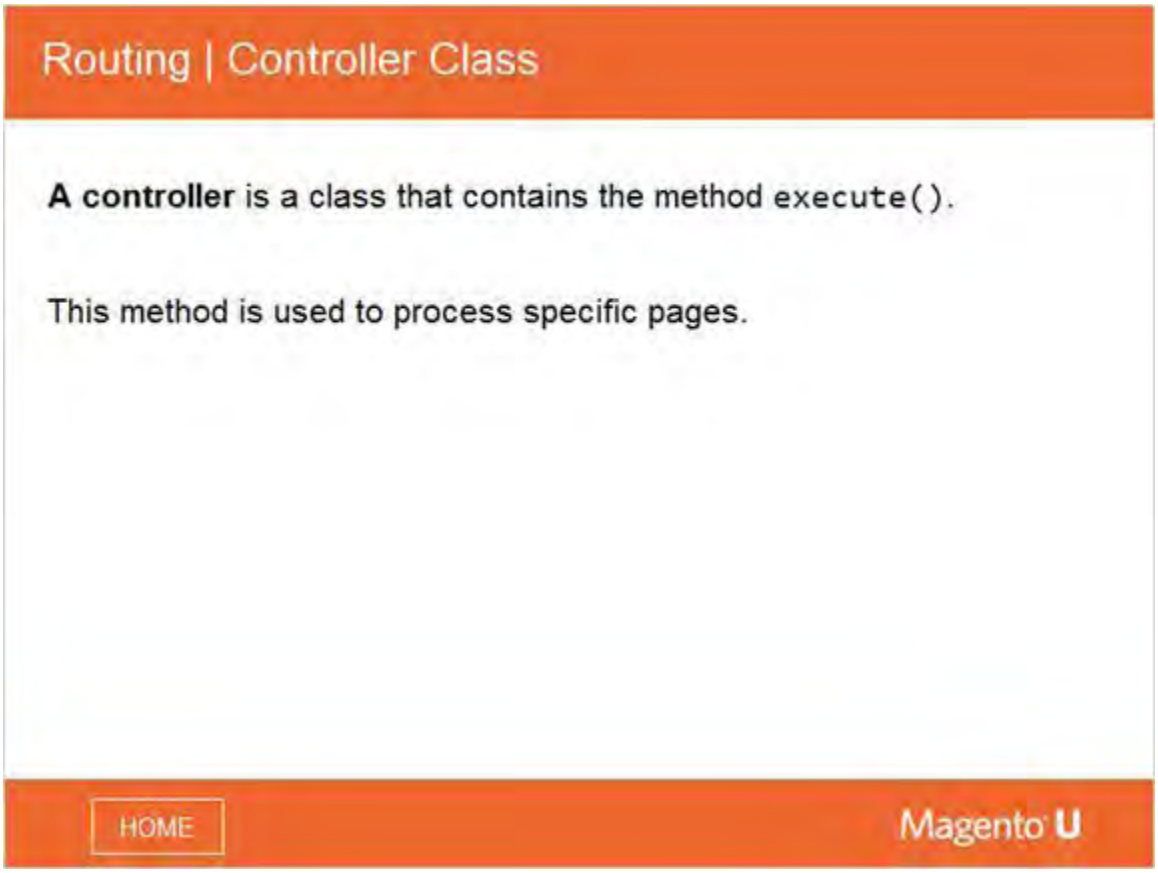
2.13 Routing: Process Sequence



Notes:

The routing process has a set of goals – defining all available routers, converting the URLs to Magento-style URLs, parsing request parameters, and identifying the controller class that will process the URL.

2.14 Routing: Controller Class

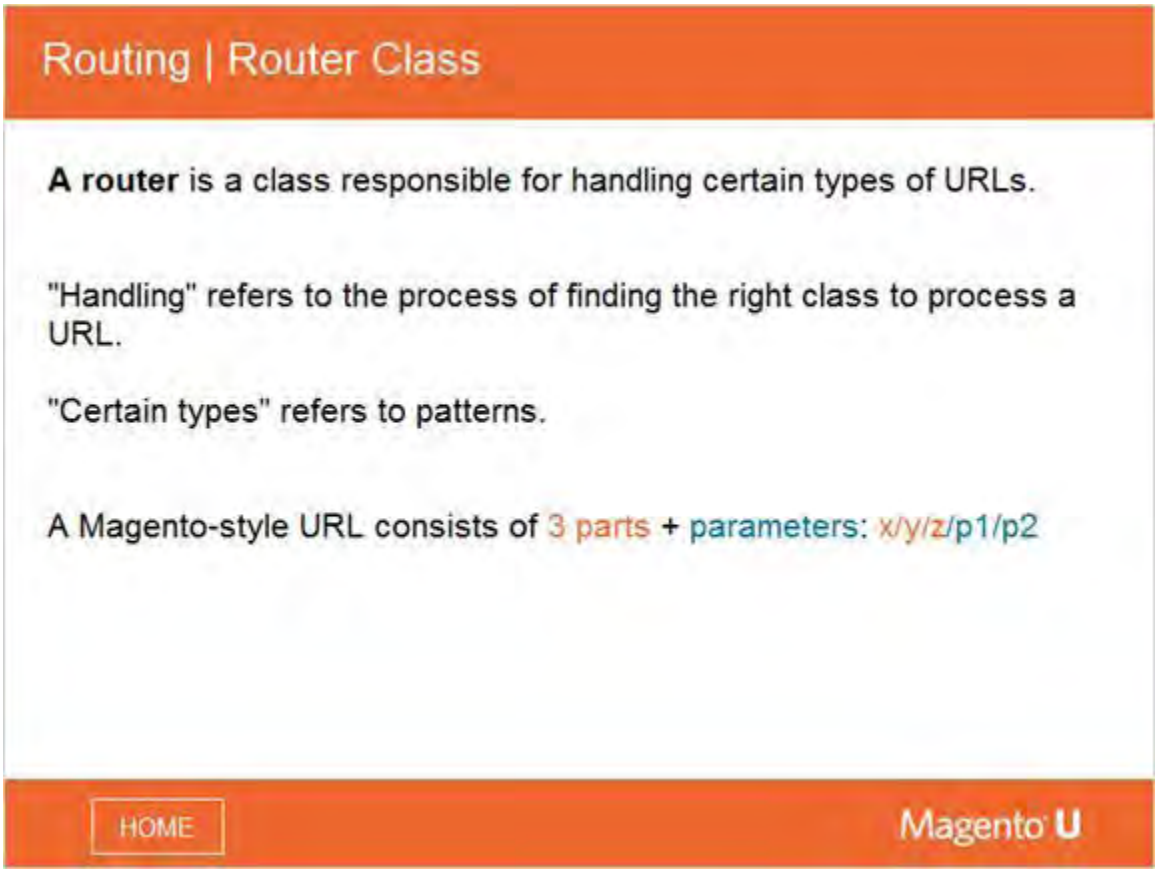


Notes:

Every action is contained in a controller class. This is why these classes are sometimes referred to as the action classes.

The class contains the `execute()` method, where you implement your logic. The `execute()` method processes specific pages.

2.15 Routing: Router Class



Notes:

A router is a class responsible for handling certain types of URLs.

Each URL has to be processed by a router class (controller or action class).

"Handling" refers to the process of finding the right class to process a URL.

"Certain types" refers to patterns.

A Magento-style URL consists of 3 parts + parameters.

For example: `catalog/product/view/id/5/`
- 3 parts = `catalog/product/view`
- Parameters = `/id/5`

So, one router will recognize this URL because of this pattern. Other URLs could have other patterns; those would be processed by different routers, matching controllers to URLs.

The concept behind routers in Magento 2 is to make the router class more granular and flexible so that it can be used in a variety of situations. When we say it handles different types of URLs, we are referring to Product View URLs, such as `Product/View/id1, /id2, /id5`, etc.

They are different URLs, but the same page, so the router class that processes these pages is the same.

2.16 Routing: Route Declaration

Routing | Route Declaration

A route is the declaration of a set of pages belonging to a particular module.

Routes are defined in the `routes.xml` file.

HOME

Magento U

Notes:

Routes are defined here in the `routes.xml` file.

Magento1 requires five or six lines of code to define a route.

Now, the syntax to declare a route is much easier, although all the concepts are the same.

2.17 Check Your Understanding

Check Your Understanding...

```
initObjectManager()

private function initObjectManager()
{
   if(!$this->objectManager) {
       $this->objectManager = $this->factory->create($this->server);
       $this->maintenance = $this->objectManager->get(Magento\Framework\App\
                                                       MaintenanceMode);
   }
}
```

Identify the class & method that contains this code shown

enter the class here...

enter the method here...

Now, complete the execution flow for that class and method:

index.php > bootstrap::createApplication() > bootstrap:: ? ?

HOME

Class: `Magento\Framework\App\Bootstrap`

Method: `createApplication()`

Flow: `Bootstrap::initObjectManager()`

2.18 Controller Processing

Controller Processing

Historically, explanations about the request flow process have focused on controllers.

Controllers:

- Belong to the Model-View-Controller (MVC) concept.
- Are the part of the application that actually processes requests with request parameters.
- Start the rendering process (View).
- Sometimes initiate models (Model).

HOME

Magento U

Notes:

Our discussion now turns to controller processing within Magento.

A "controller" is part of the MVC concept. The Magento implementation of MVC is non-standard. Here, the controller does not manage data - it manages parameters that come from the request and possible exceptions, and it may do things like initiate a main class for product category pages.

It then launches the rendering process, which manages the relation between view and data layers in production, executing tasks like loading models, collections, and templates.

2.19 Controller Processing: Code Example

Controller Processing | Code Example

```
public function execute()  Method that responds to URL
    {
        // Get initial data from request
        $categoryId = (int) $this->getRequest()->getParam('category', false);
        $productId = (int) $this->getRequest()->getParam('id');
        $specifyOptions = $this->getRequest()->getParam('options');
        if ($this->getRequest()->isPost() &&
            $this->getRequest()->getParam(self::PARAM_NAME_URL_ENCODED))  {

        $productId = $this->_initProduct();
        if (!$product)  {
            return $this->noProductRedirect();
        }
        if ($specifyOptions)  {
            $notice = $product->getTypeInstance()->getSpecifyOptionMessage();
            $this->messageManger->addNotice($notice);
        }
        $resultRedirect = $this->resultRedirectFactory->create();
        $resultRedirect->setRefererOrBaseUrl();
        return $resultRedirect;
    }
```

HOME

Magento U

Notes:

This code example presents a native controller that uses `categoryId()`.

The first highlight focuses on the `execute()` method, which processes a URL and initiates the class, `initProduct`.

The second highlight displays how the `$request` object can be accessed from within a controller and provides access to the `$request` object, the `$response` object, and parameters.

2.20 Controller Processing: Code Example

Controller Processing | Code Example (continued)

```
protected function _initProduct()
    {
        $categoryId = (int)$this->getRequest()->getParam('category', false);
        $productId = (int)$this->getRequest()->getParam('id');

        $params = new \Magento\Framework\Object();
        $params->setCategoryId($categoryId);

        /** @var \Magento\Catalog\Helper\Product $product */
    $product = $this-> objectManager->get('Magento\Catalog\Helper\Product');
        return $product->initProduct($productId, $this, $params);
    }
```

HOME

Magento U

Notes:

Continuing with the same code example, this blue highlight displays how the controller communicates with models.

The `\Magento\Catalog\Helper\Product` method, `$product`, is created and loaded by `objectManager`, providing `productId` and other information.

2.21 Controller Processing: Code Example

Controller Processing | Code Example (continued)

```
// Render page
    try {
        $page = $this->resultPageFactory->create(false, ['isIsolated' => true]);
        $this->viewHelper->prepareAnd Render($page, $productId, $this, $params);
        return $page;
    } catch (\Exception $e) {
        if ($e->getCode() == $this->viewHelper->ERR_NO_PRODUCT_LOADED) {
            return $this->noProductRedirect();
        } else {
            $this->_objectManager->get('Psr\Log\LoggerInterface')->critical($e);
            $resultForward = $this->resultForwardFactory->create();
            $resultForward->forward('noroute');
            return $resultForward;
        }
    }
```

HOME

Magento U

Notes:

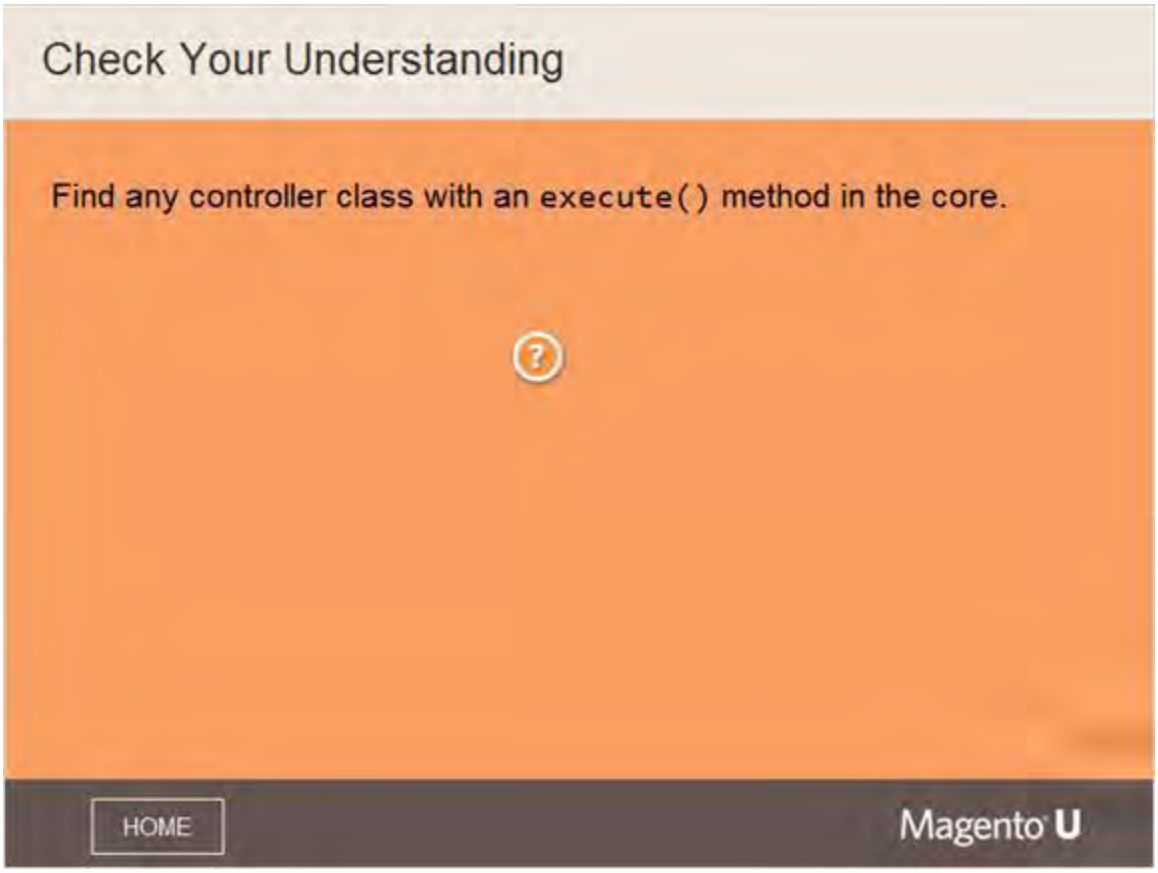
Still using the same controller example, this blue highlight displays how a controller communicates with the view layer – an example of how rendering might work.

In this case, a page object is created and rendered by `PageFactory`.

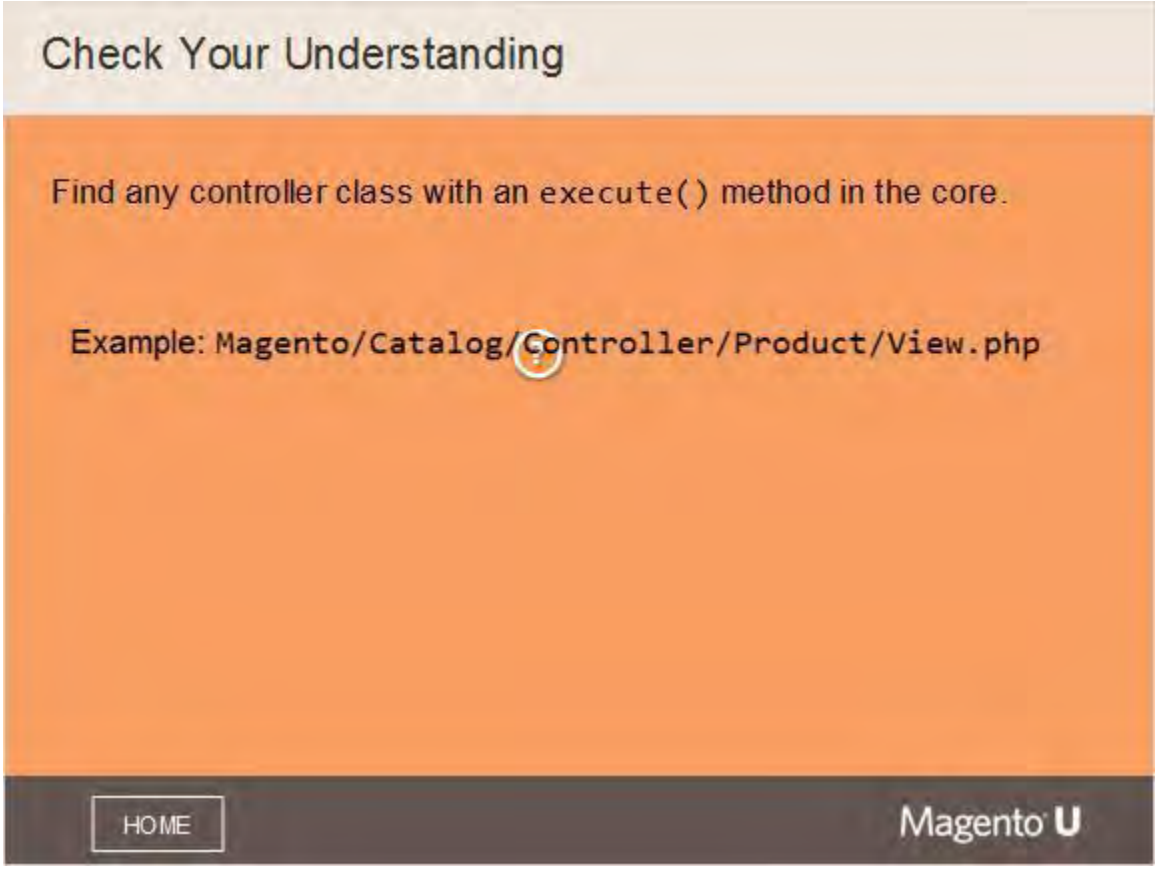
There are other ways to do this – the example presents just one way a controller might render in Magento 2.

This code also illustrates how a controller function can catch exceptions.

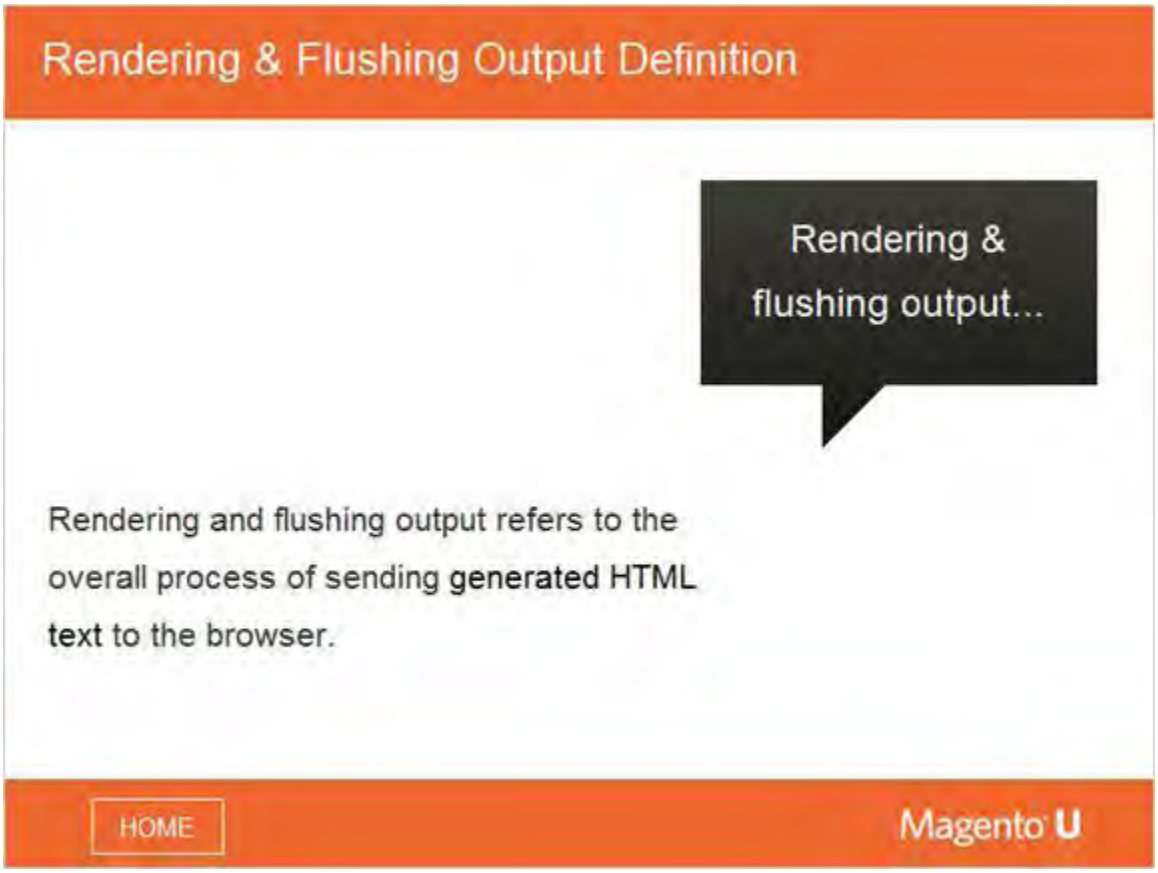
2.22 Check Your Understanding



Answer 2.22 (Slide Layer)



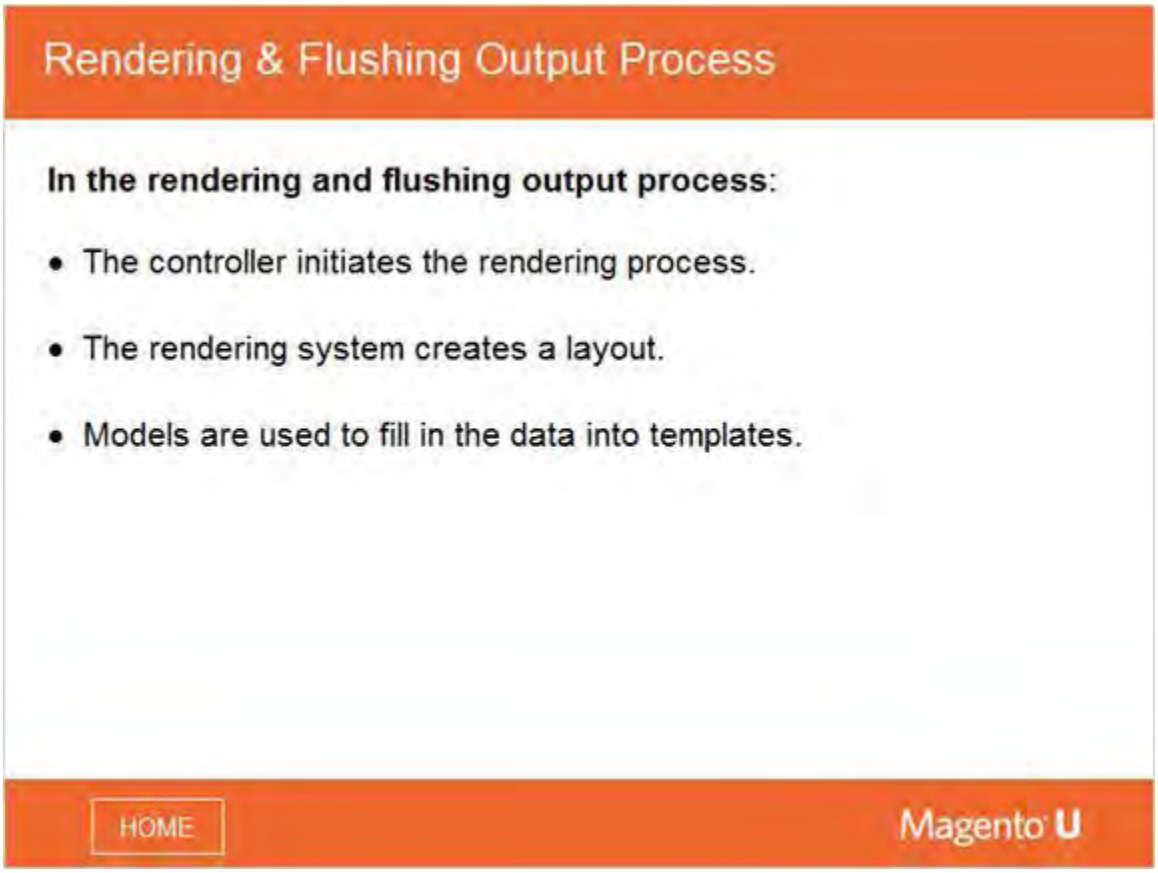
2.23 Rendering & Flushing Output Definition



Notes:

Once the rendering phase is complete, the HTML will be sent back to the browser.

2.24 Rendering & Flushing Output Process



Notes:

The physical process of rendering is *conceptually* the same as in Magento 1, but it differs on the code and architectural levels. Including templates caches them into the buffer and places them all in one string in the response object. This explains the flushing process – we have created the object, but it is in the string inside, so we have to flush it.

This process impacts performance to varying degrees. For example, if you have 100KB of text and store it every time, it will cost you somewhat in performance versus caching it in the buffer.

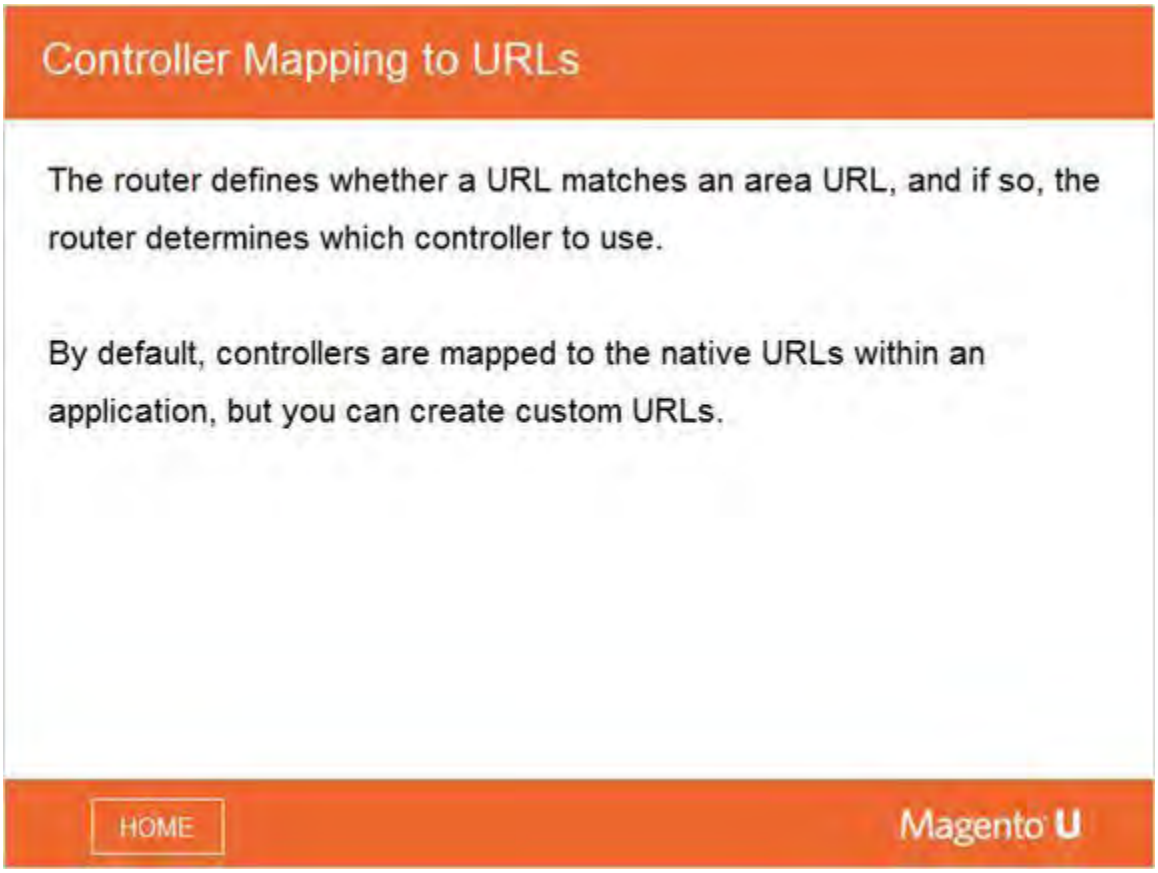
In general terms, the following steps are taken during the rendering and flushing output process:

- The front controller (`Magento\Framework\App\FrontController`) dispatches a request and gets a result object.
- The app (`Magento\Framework\App\Http`) in the `launch()` method copies HTML to the response object.
- The bootstrap (`Magento\Framework\App\Bootstrap`) flushes that HTML from the response object to the browser.

Rendering and Flushing Output Process

```
\Magento\Framework\Profiler::start('magento');
          $this->initErrorHandler();
          $this->initObjectManager();
          $this->assertMaintenance();
          $this->assertInstalled();
          $response = $application->launch();
          $response->sendResponse();
          \Magento\Framework\Profiler::stop('magento');
          } catch (\Exception $e) {
```

2.25 Controller Mapping to URLs



Notes:

The router defines whether a URL matches an area URL, and if so, the router determines which controller to use and sets control to the controller action class to process it, as a class object has been populated previously by a router or app.

2.26 Code Demonstration: Controller Mapping to URLs

Notes:

Here is an example of a controller.

To access the controller, start by entering the following:
`www/magento/m2/app/code/magento/catalog/controller/product/view.php`

Before in the view controller, we had three parts of the URL and you had to create a class which corresponded to the second part and a method which corresponded to the third part. Now in Magento 2, you have to create a class that corresponds to the third part of the URL, method `execute()`, but the second part should be a folder.

In this example we have `Catalog/Controller/Product/` - the product corresponds to the second part of the URL and the view class corresponds to the third part of the URL, and method `execute()` when it has been processed.

2.27 Exercise 2.2.1

Reinforcement Exercise (2.2.1)

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

Find a place in the code where output is flushed to the browser.
(By flushing output, we mean a "send" call to the response object.)

Now, create an extension that captures and logs the file-generated page html.

HOME

Magento U

3. Request Routing

3.1 Request Routing

Notes:

Now let’s discuss the phases of request routing introduced in the previous sections.

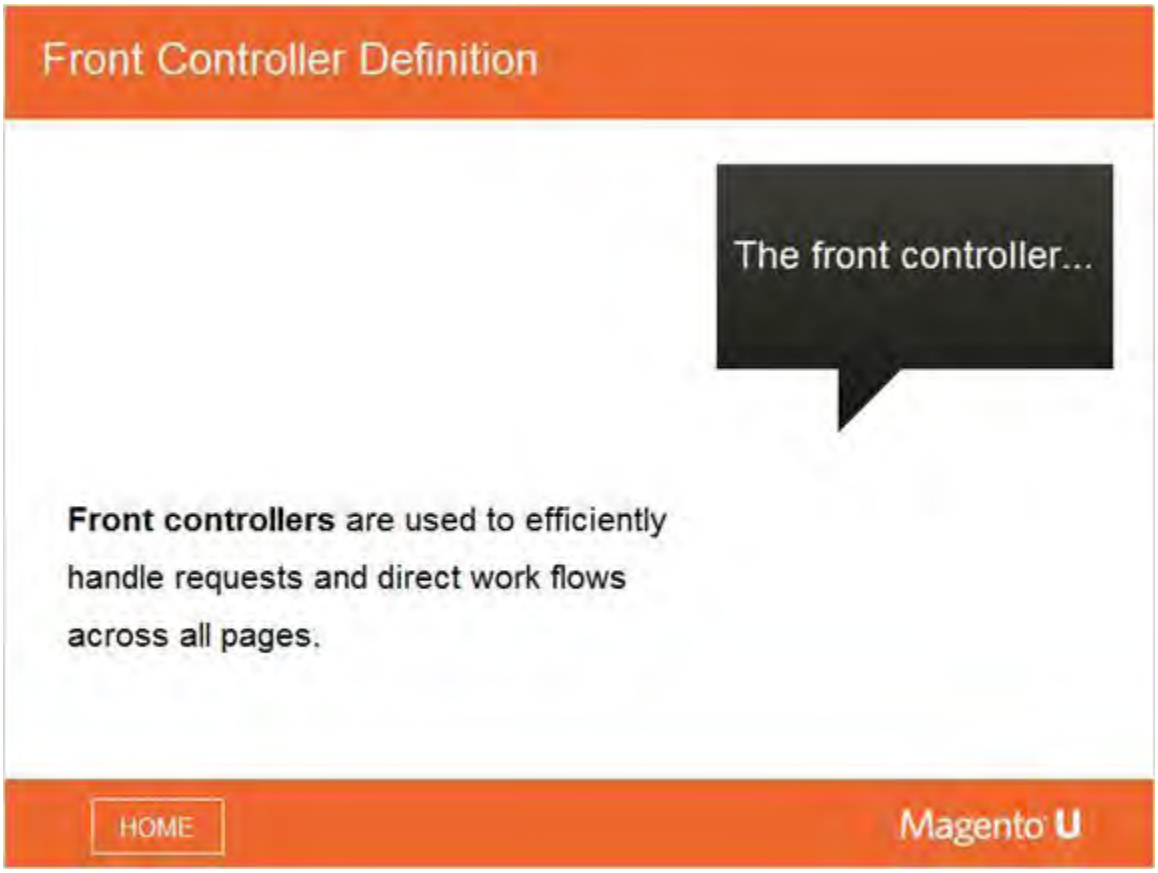
3.2 Module Topics

Notes:

Request routing refers to the process of managing how client HTTP requests are processed within an application. This module will cover the following related topics:

- Front controller
- Routing mechanisms
- URL processing

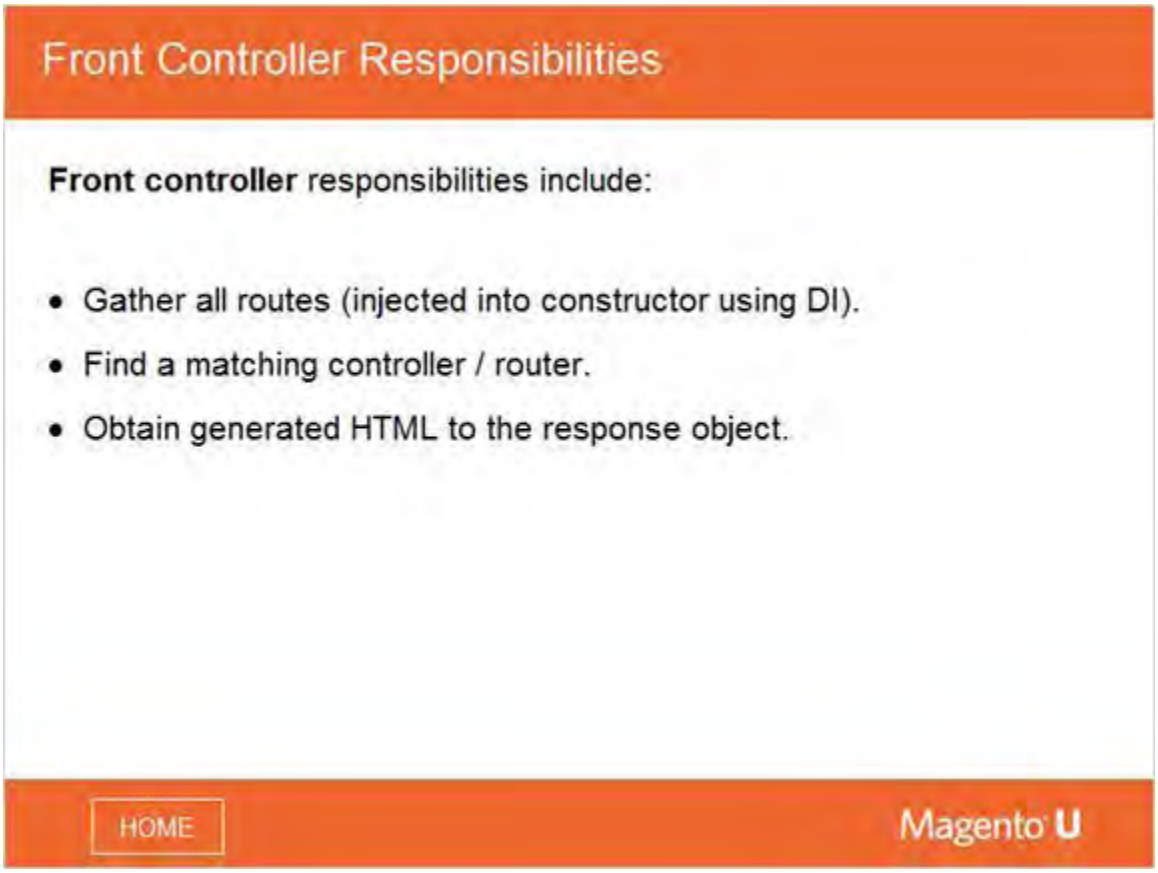
3.3 Front Controller Definition



Notes:

Front controllers are the first step in handling requests and work flows across all pages.

3.4 Front Controller Responsibilities



Notes:

Usually an app will have only one controller class, but when the application has many pages, they can't all be handled by one controller. In this case, we have two levels of URL processing: the front controller/router, and then the action class. As the front controller contains the routing function, it is a component that is almost never customized.

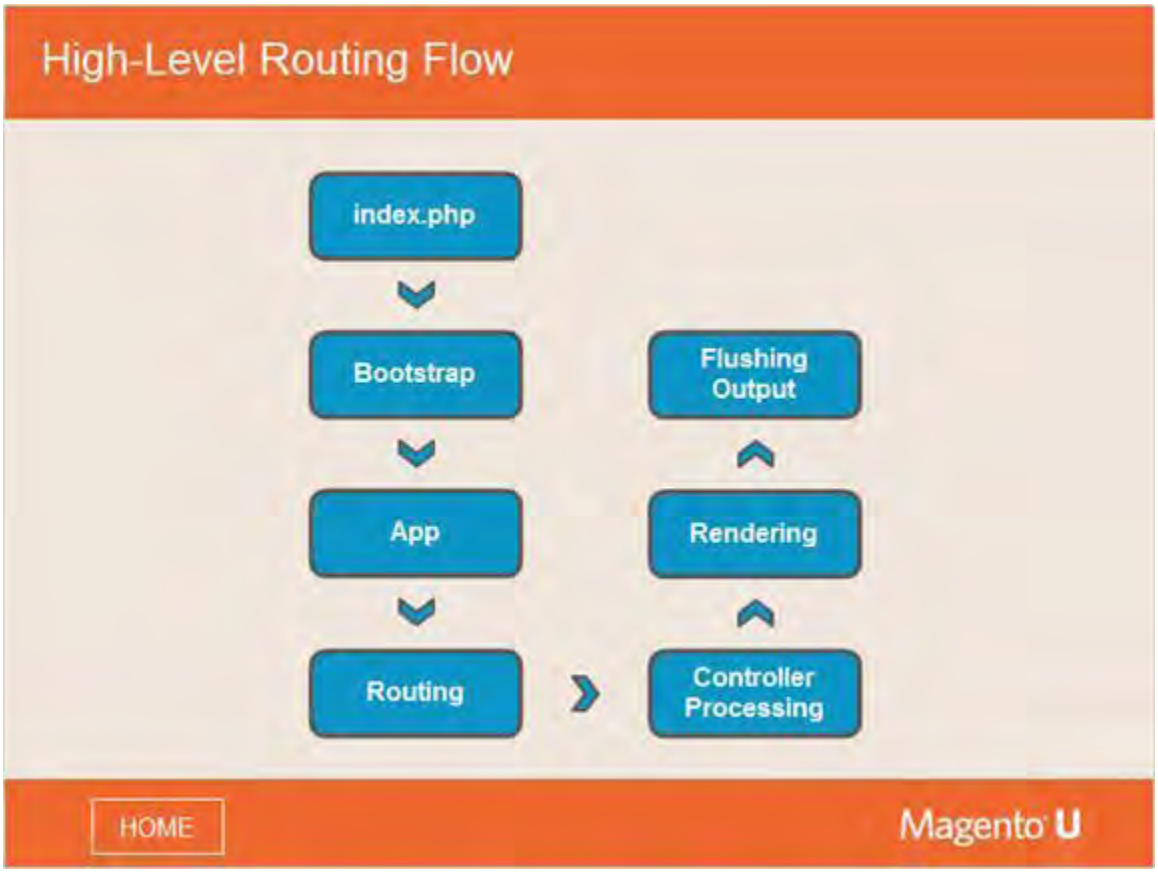
The router may be modified, but it is unlikely unless you have some new scheme of URLs.

Basically, the front controller controls all the other controllers. In Magento 2, it gathers routes, matches controllers, and obtains the HTML generated to the response object.

In Magento 1, the front controller has a whole class and does a variety of functions, but in Magento 2 it simply furnishes routers created by DI and runs controllers- that’s all it does.

We do still have URL rewrites, but they are used elsewhere, as we will see later.

3.5 Front Controller: High-Level Flow

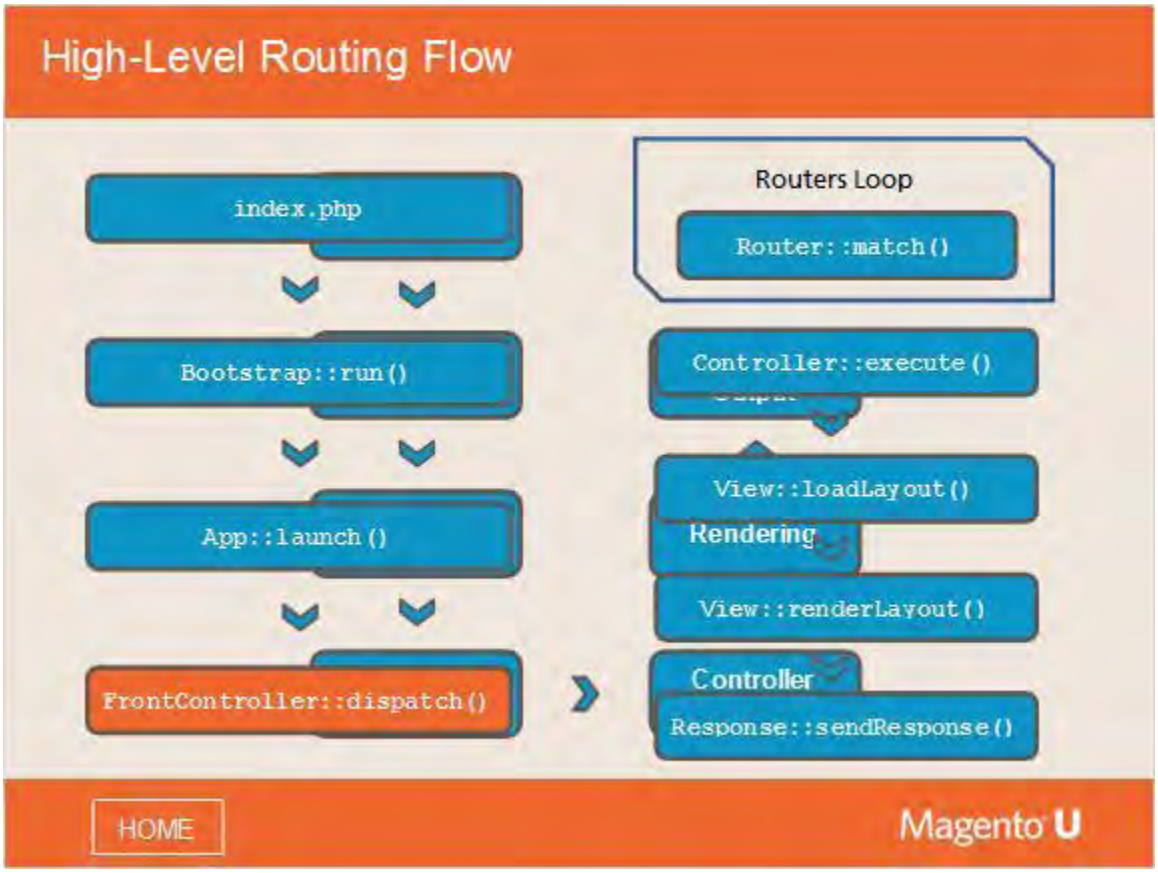


Notes:

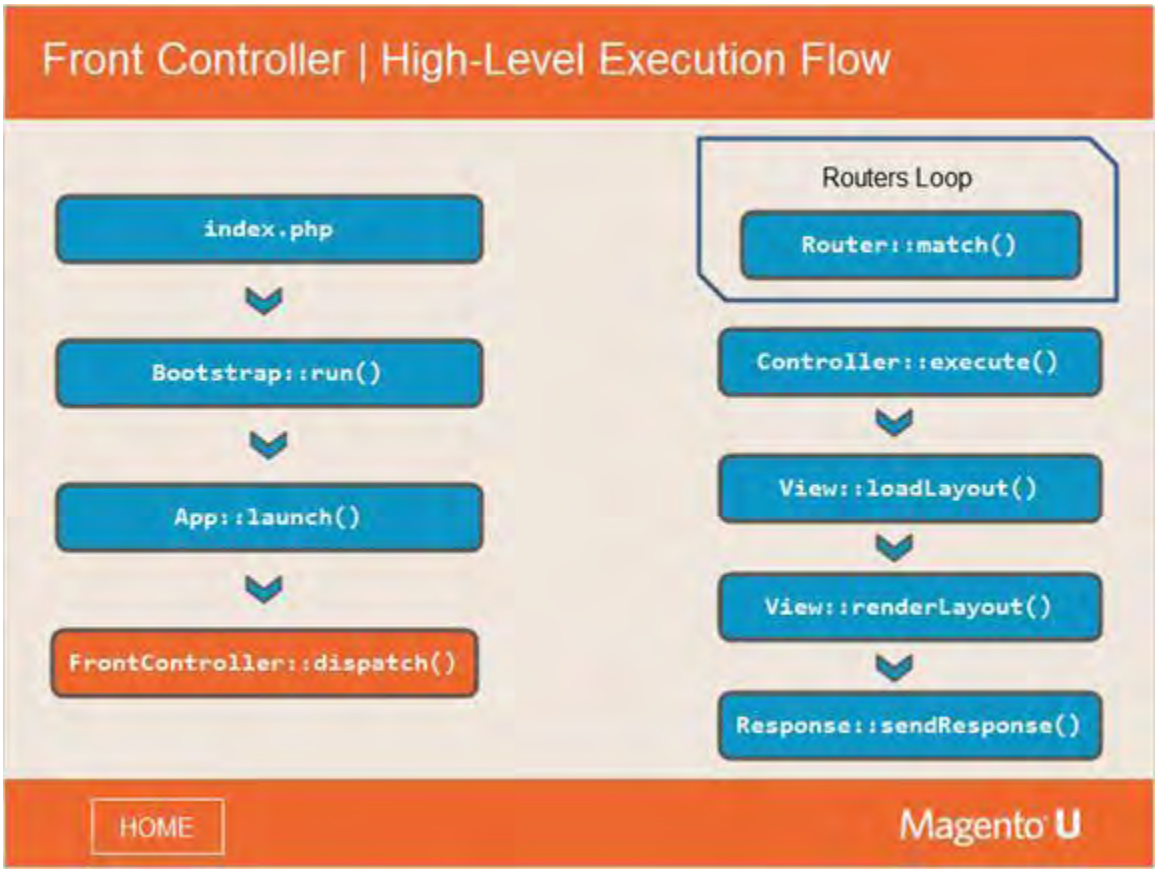
This diagram should be familiar by now. It is the same diagram we looked at in the Overview module. In that section of the course, we looked at key code snippets corresponding to application initiation.

Now, we are going to look at key code corresponding to front controllers and routing.

High-Level Execution Flow (Slide Layer)



3.6 Front Controller: High-Level Flow



Notes:

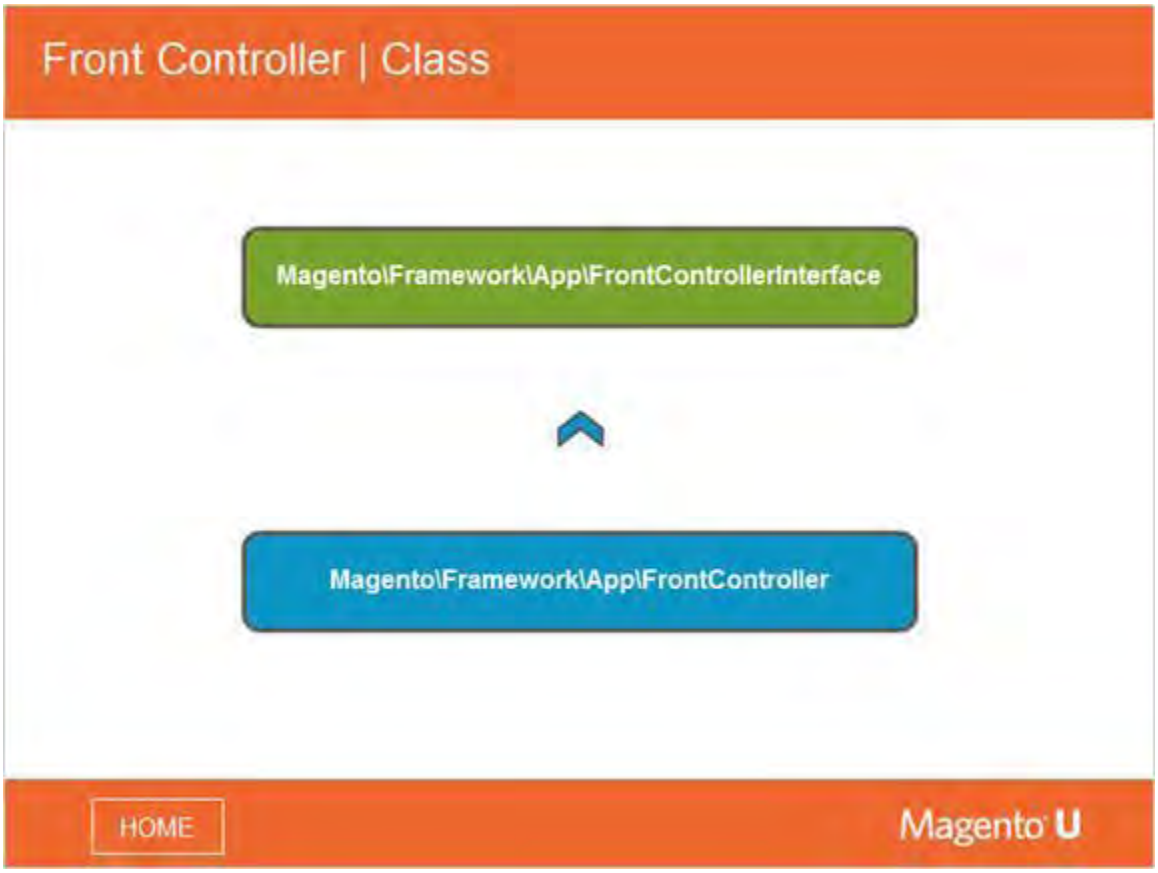
This updated version of the routing flow diagram now shows you the methods that correspond to key steps in the actual routing process, beyond application initiation.

Note the calls like `Bootstrap::run()` and `App::launch()`, as well as `FrontController::dispatch()`, which is where the front controller starts its processes.

The front controller is not displayed on the generalized execution flow slide shown previously because it is actually just a small step in the routing phase of the overall process.

We will be using this diagram throughout this unit to discuss these steps in more detail, starting with `FrontController::dispatch()`.

3.7 Front Controller: Class



Notes:

The Front Controller class implements an interface, as shown in the diagram (note that the interface is in green).

It is one class, with one method that implements one standard interface.

3.8 Front Controller: FrontControllerInterface

Front Controller | FrontControllerInterface

```
<?php
// Application front controller responsible for dispatching application
// requests

namespace Magento\Framework\App;
interface FrontControllerInterface
{
    /**
    * Dispatch application action
    * @param RequestInterface $request
    * @return ResponseInterface
    */
    public function dispatch(RequestInterface $request);
}
```

HOME

Magento U

Notes:

This example shows sample code for the `FrontControllerInterface`.

3.9 Front Controller: Implementation of the Dispatch Method

Front Controller | Implementation of the Dispatch Method

```
public function dispatch(RequestInterface $request)
   {
       \Magento\Framework\Profiler::start('routers_match');
       $routingCycleCounter = 0;
   $result = null;
       while (!$request->isDispatched() && $routingCycleCounter++ < 100) {
           /** @var \Magento\Framework\App\RouterInterface $router */
           foreach ($this->_routerList as $router) {
               try {
                   $actionInstance = $router->match($request);
                   if ($actionInstance) {
                       $request->setDispatched(true);
                       $actionInstance->getResponse()->setNoCacheHeaders();
                       $result = $actionInstance->dispatch($request);
                       break;
                   }
```

HOME

Magento U

Notes:

Finally, the highlighted code shows the HTML generation step. Unlike in Magento 1, where the router would implement everything, in Magento 2 the action instance implements `dispatch()`. We will see later that it must implement `execute()` as well.

The first highlight shows the cycle counter set to less than 100, to keep the algorithm from getting stuck in an infinite loop.

The next highlight shows the action for finding the correct router for request processing.

The third highlight shows the action to catch exceptions. It will return either a result interface or a response interface; most likely it will be an instance of `ResultInterface` or `HttpInterface`, which is just a response object.

3.10 Front Controller: Implementation of the Dispatch Method

Front Controller | Implementation of the Dispatch Method

```
        } catch (Action\NotFoundException $e) {
                $request->initForward();
                $request->setActionName('noroute');
                $request->setDispatched(false);
                break;
            }
        }
    }
    \Magento\Framework\Profiler::stop('routers_match');
    if ($routingCycleCounter > 100) {
        throw new \LogicException('Front controller reached 100
                                   router match iterations');
    }
    return $result;
}
```

HOME

Magento U

Notes:

Continuing with our example, this code displays the action instance to dispatch the request and generate the HTML.

3.11 Front Controller: Initialization

Front Controller | Initialization

Magento\Framework\App\Http initializes the FrontController class
... and ...
delegates the request handling to the front controller

```
public function launch()
    {
    $areaCode = $this->_areaList->getCodeByFrontName($this->_request->getFrontName());
    $this->_state->setAreaCode($areaCode);
    $this->_objectManager->configure($this->_configLoader->load($areaCode));

// @var \Magento\Framework\App\FrontControllerInterface $frontController
    $frontController = $this->_objectManager->get('Magento\Framework\App
                                                  \FrontControllerInterface');
    $result = $frontController->dispatch($this->_request);

// Temp solution until all controllers returned, not ResultInterface (MAGETWO-28359)
    if ($result instanceof ResultInterface) {
         $this->registry->register('use_page_cache_plugin', true, true);
         $result->renderResult($this->_response);
    } elseif ($result instanceof HttpInterface) {
         $this->_response = $result;
    } else {
         throw new \InvalidArgumentException('Invalid return type');
    }
```

HOME

Magento U

Notes:

The Front Controller Execution Flow diagram, presented earlier, demonstrated that the front controller was initiated from the `App::launch()` method.

This code example provides more information on how that happens. `FrontController` is generated by the `app` class (`\Magento\Framework\App\FrontControllerInterface`). Above that is the bootstrap, above the bootstrap is the `index.php`, and above all this, the web server.

It is useful to keep this in mind as we examine details of that flow.

3.12 Front Controller: Initialization

Front Controller | Initialization

```
// This event allows launching something before sending output
// (allow cookie setting)
    $eventParams = ['request' => $this->_request,
                    'response' => $this->_response];
    $this->_eventManager->dispatch('controller_front_send_response_before',
                                   $eventParams);
    return $this->_response;
}
```

HOME

Magento U

Notes:

So, `app` creates the controller, calls `dispatch()` from the controller, launches the router, finds the right controller, calls `dispatch()` for the controller, calls `execute()` for the controller, and then comes back with the response.

This allows the system to launch the application before sending the response.

3.13 Front Controller: Obtain Generated HTML

Front Controller | Obtain Generated HTML

```
while (!$request->isDispatched() && $routingCycleCounter++ < 100) {
            /** @var \Magento\Framework\App\RouterInterface $router */

            foreach ($this->_routerList as $router) {
                try {

                    $actionInstance = $router->match($request);
                    if ($actionInstance) {
                        $request->setDispatched(true);
                        $actionInstance->getResponse()->setNoCacheHeaders();
                        $result = $actionInstance->dispatch($request);
                        break;
                    }

            } catch (Action\NotFoundException $e) {
                    $request->initForward();
                    $request->setActionName('noroute');
                    $request->setDispatched(false);
                    break;
                }
            }
        }
```

HOME

Magento U

Notes:

Here we see the result object containing the HTML being returned to the front controller.

3.14 Exercise 2.3.1

Reinforcement Exercise (2.3.1)

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

Create an extension that logs into a file a list of all available routers.

HOME

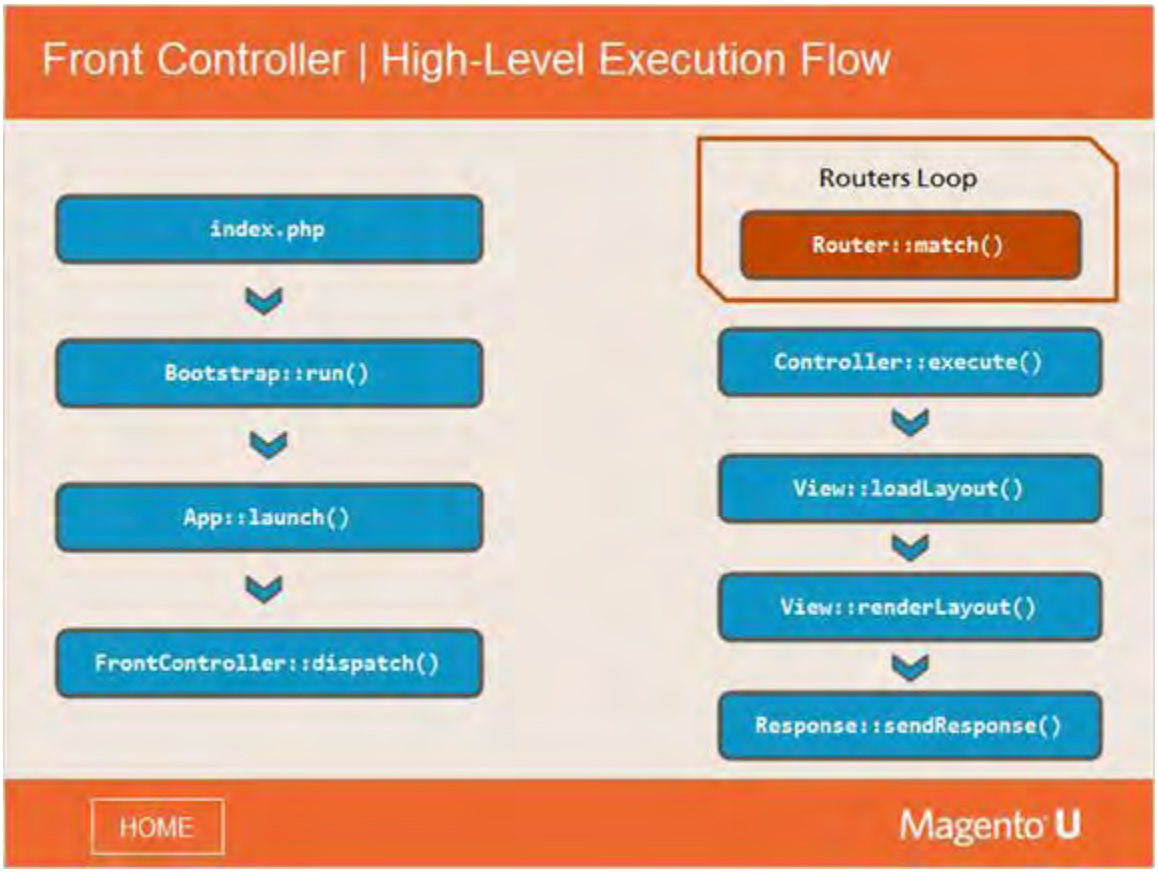
Magento U

3.15 Routing Mechanism

Notes:

Our next topic is the routing mechanism.

3.16 Front Controller: High-Level Execution Flow

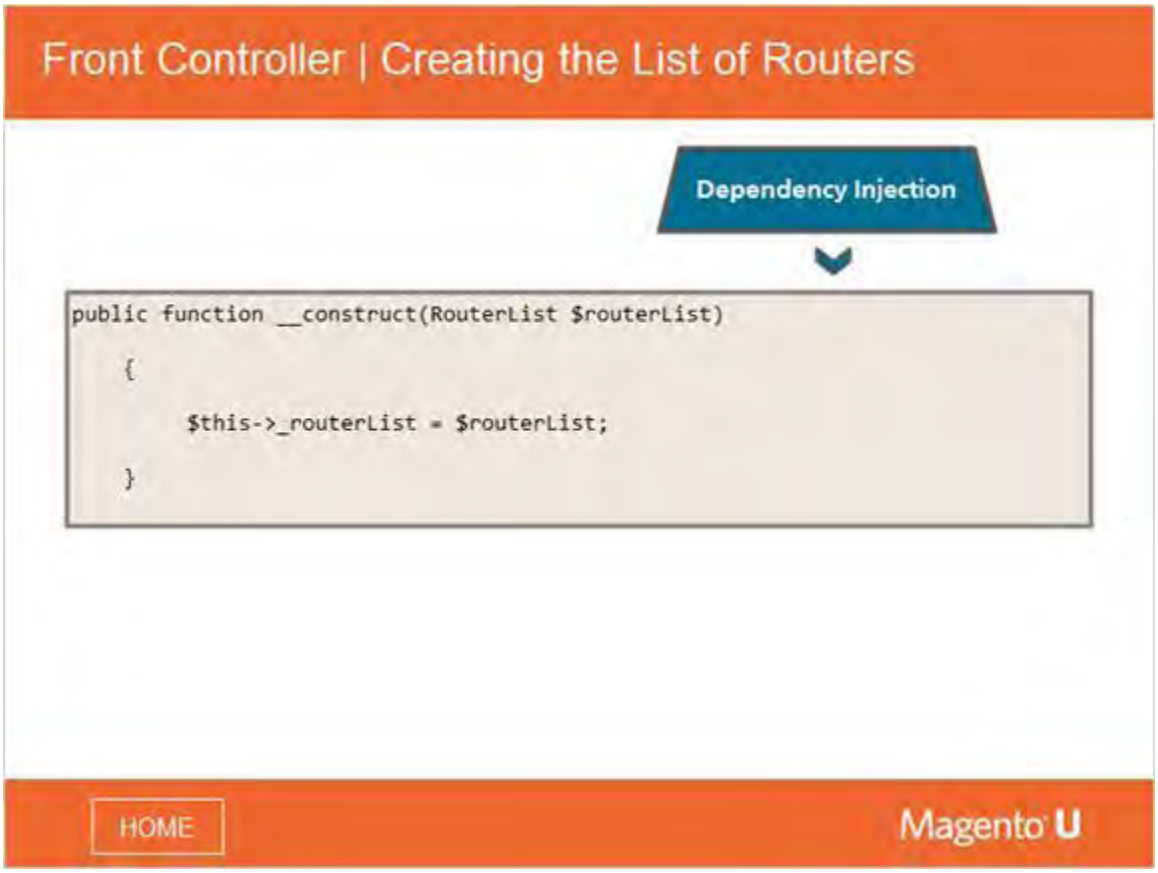


Notes:

We are now going to focus on routers in the overall execution flow, looking for the list of routers and matching them to requests.

Again referencing the Execution Flow diagram, we are now at the point of `Router::match()`, the most useful method when looking for the list of routers.

3.17 Front Controller: Creating the List of Routers



Notes:

This is how the list of routers is created, using Dependency Injection as a parameter.

Reference: Locate the following file in your Magento installation (to see all the available routers):

`<magento_installation_dir>/lib/internal/Magento/Framework/App/FrontController.php`

3.18 Routing Mechanism: List of Available Routers

Notes:

In the native Magento 2 installation, there are five routers, as shown on the slide. Each of these will be discussed in more detail in the following slides.

First, we have the base router; this is the analog of the standard router in Magento 1.

```
Magento\Core\App\Router\Base
```

Next is the default router in Magento 2.

```
Magento\Framework\App\Router\DefaultRouter
```

Third, the CMS router, which processes CMS pages in Magento 2.

```
Magento\Cms\Controller\Router
```

Fourth, the processing URL rewrite router in Magento 2. This is how URL rewrites work in Magento2.

```
Magento\UrlRewrite\Controller\Router
```

The last router is the design editor router in Magento 2.

```
Magento\DesignEditor\Controller\Varien\Router\Standard
```

The first three routers have equivalents in Magento 1; the last two, the URLRewrite and DesignEditor routers, are additions in Magento 2.

3.19 Code Demonstration: Magento\Core\App\Router\Base

Notes:

- The base router extends the `RouterInterface`. This is very important. The base router is an analog of the standard router in Magento 1.

How do we access it? Let's take a look at the base router.

It is located in Magento (not the library)….`Magento/Core/App/Router/Base.php`.

Remember in Magento 2 there is no strict separation between class groups. You don't have to declare them. This is an example of that.

The classes will work in the same way as other classes. The method follows the same steps as in Magento 1.

Here is the constructor – the method.
`match()` moves to `matchAction()`, goes to every module – it goes to factory, it checks for the appropriate controller, and if it doesn't exist, it builds it and then includes it.

3.20 Routing Mechanism: Router Interface

Routing Mechanism | Router Interface

```
<?php
// Router... matches action from request
namespace Magento\Framework\App;
interface RouterInterface
{
    /**
     * Match application action by request
     *
     * @param RequestInterface $request
     * @return ActionInterface
     */
    public function match(RequestInterface $request);
}
```

HOME

Magento U

Notes:

The router interface is simple in structure; it contains a single method, `Router::match()`.

3.21 Code Demonstration: Magento\Framework\App\Router\DefaultRouter

Notes:

Note that the default router is the only router located in the framework.

You can find the default router by going to:

```
Magento/Framework/App/Router/DefaultRouter.php
```

The Default Router process handles the “Not Found” page. So when there is no other router that can process a page, the request is sent to the default router to handle it (which means a "Not Found" page is returned).

Magento 2 allows you to create different handles for "Not Found" pages. For example, you may have a page that says: “Sorry, this product does not exist" or "This category does not exist”. The default router will handle all the types.

3.22 Code Demonstration: Magento\Cms\Controller\Router

Notes:

Here is the `CMS\Controlle\Router`. This is the CMS router in Magento 1. Looking at this router, the idea is to handle CMS pages and convert them. It checks if you have a redirect URL, and if you don't, it will generate a redirect or send it forward.

`setDispatched(false)` returns `$instance`, and does the tasks through the `Action/Forward`.

3.23 Code Demonstration: Magento\UrlRewrite\Controller\Router

Notes:

Let's look at the `UrlRewrite\Controller\Router`.

In Magento 1, the URL rewrite mechanism is implemented in the front controller, in the action class `frontController`.

In Magento 2 we have a separate router for processing URL rewrites.

We see here the parameter `RequestInterface` with a request object. It will perform a call and `getRewrite()` using this `$requestPath` and `$storeID`. The `getRewrite()` call uses a URL finder and performs a call `findOneByData()`. This call uses `performConnection::fetchRow()` and `prepareSelect($data)` to get the database, where `requestPath` and `storeID` are used to find the appropriate data from the database.

3.24 Code Demonstration: Magento\DesignEditor\Controller\Varien\Route\Standard

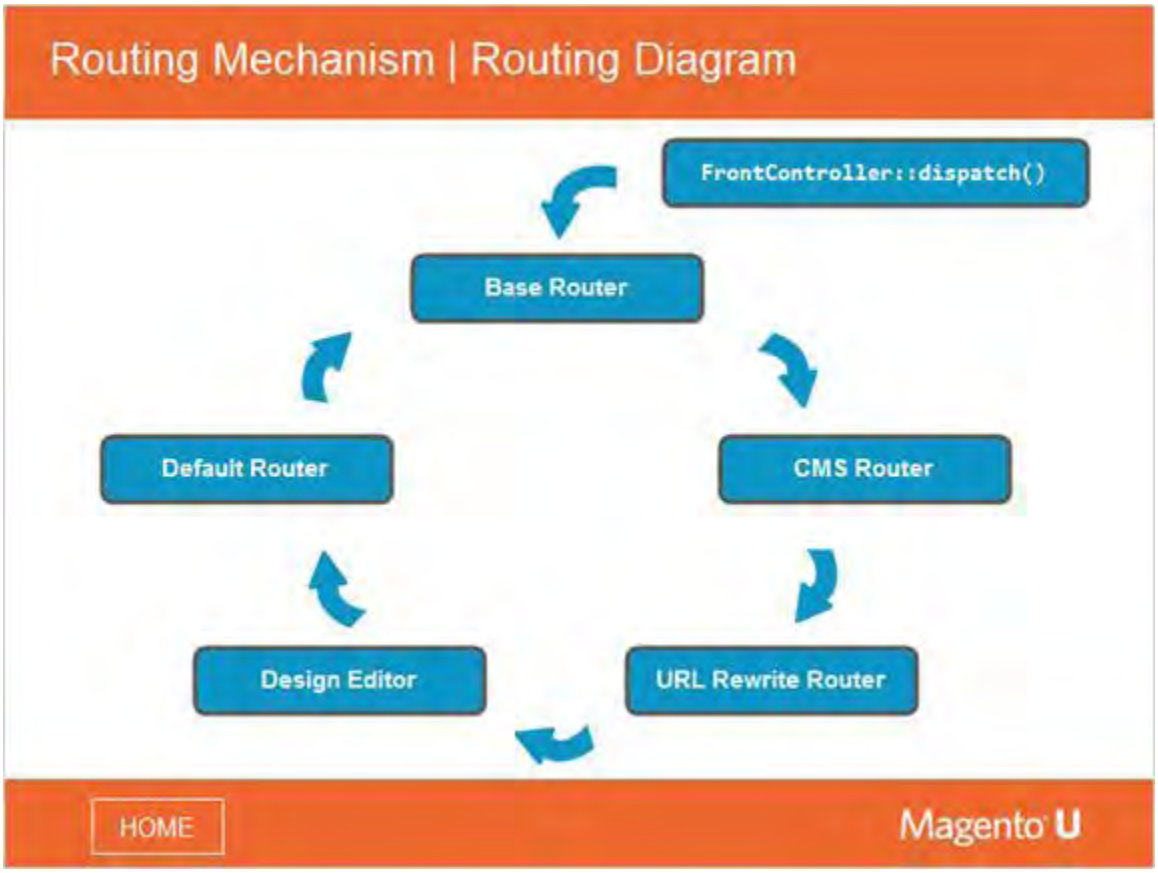
Notes:

Now we are looking at the `Magento\DesignEditor\Controller\Varien\Router\Standard`.

This is the design editor, something different from anything in Magento 1. It's called to process a specific request that might come from `DesignEditor`.

It uses this `match()` method and the `DesignEditor` helper to match routers and process a specific request.

3.25 Routing Mechanism: Routing Diagram



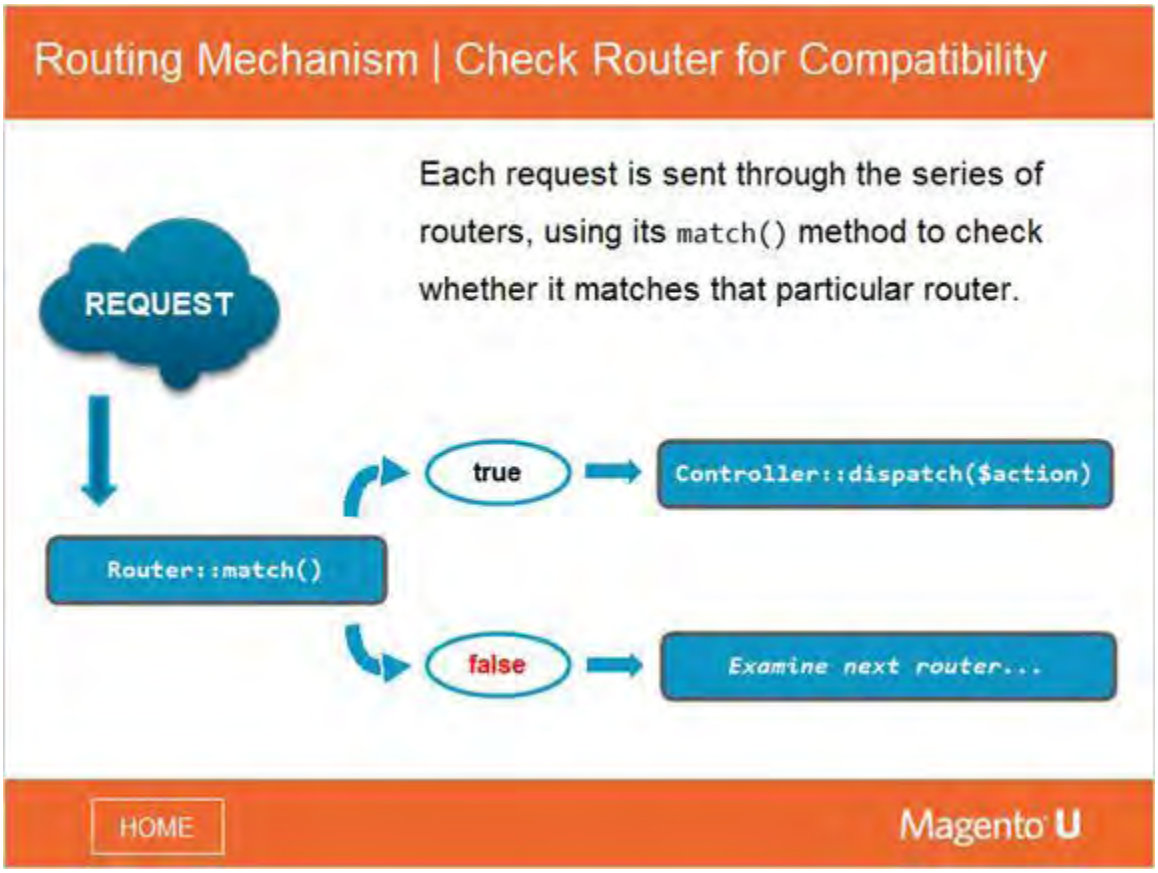
Notes:

Here is a diagram of the order in which the five native routers are processed.

The front controller starts with the base router to check if it can handle the request or not.

If not, it goes to the next router, CMS, and so on. The last router to take over is the default router when there is no other router that can take the request.

3.26 Routing Mechanism: Check Router for Compatibility



Notes:

Each request is sent through the series of routers, using its `match()` method to check whether it matches that particular router.

If there is a match (true), then an action instance is returned.

If false, the request passes to the next router in the sequence.

The process ends with the default router as the "catch-all."

3.27 Routing Mechanism: Register a New Router

Notes:

You can create a custom router with its own action instance and dispatch method.

To create a custom router, you need to add it as a parameter to the class:

```
Magento\Framework\App\RouterList
```

An example of that can be seen in `Cms/etc/frontend/di.xml`.

In Magento 2, the true argument returns an instance and takes charge of the false argument.

3.28 Exercise 2.3.2

Reinforcement Exercise (2.3.2)

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

Create a new router which "understands" URLs like

`/frontName-actionPath-action`

(and converts them to:)

`/frontName/actionPath/action`

HOME

Magento U

3.29 URL Processing

Notes:

This section focuses on URL processing.

3.30 URL Processing: Overview

URL Processing | Overview

URL Processing

In Magento 2, there are a number of native pages, each with a distinct URL.

Sometimes individual pages can also have multiple URLs (for example, as part of search engine optimization).

HOME

Magento U

Notes:

In Magento 2, the same product can have different URLs.

There are several reasons why you would assign multiple URLs: search engine optimization, human reading optimization, and so on.

3.31 URL Processing: Base Router URL Composition

URL Processing | Base Router URL Composition

The standard router is implemented by the class:
`Magento\Core\App\Router\Base`

The structure of the URL, as accepted by the router, is:
`$baseUrl/$frontName/$controllerName/$actionName/$otherParams`

Example:

http://magento-installation.com/catalog/product/view/id/1/

Front name:	catalog	Action name:	view
Controller name:	product	Parameters:	id=1

HOME

Magento U

Notes:

This slide shows you the base router URL composition. The base router is implemented by the class `magento\core\app\router\base`.

The structure of the URL, as accepted by the router, is:

`$baseUrl/$frontName/$controllerName/$actionName/$otherParams`

For example, the URL: http://magento-installation.com/catalog/product/view/id/1

3.32 Code Demonstration: Base Match

Notes:

In this slide, we will explain the process on how the base router will match the standard URL.

To start, we will go to the base router and we can see the `matchAction()` method. Basically, the Magento URL consists of three chunks. First is `FrontName`, second is `ActionPath`, and the third is `Action`.

So, this method extracts three chunks of the URL and tries to locate the `actionClass`. If the class is located, then it runs.

First, the module `FrontName`, usually defined in the `routes.xml` file. Here we define the route `FrontName, Catalog`, and this corresponds to the module name `Magento_Catalog` – that's the first chunk.

So it tries to match `FrontName` to the module. Then it extracts the second chunk – `ActionPath`and the third chunk, `Action`.

Next, it goes through all module names because there might be additional directories that would add routes to the `FrontName`. Based on all the modules found, it tries to define `actionClassName`, which it composes based on various rules and then creates actionInstance. The rule it uses is simple. Starting with module name like `Magento_Catalog` (it's `_Controller` with a capital C).

Then `actionPath`, and the class name should be the third chunk, $action.

So, Catalog is connected to the module `Magento, product is connected to Controller/Product` and view is connected to `view.php.`

This results in the class name `Magento\Catalog\Controller\Product\View.`

3.33 URL Processing: Controller Execution

URL Processing | Controller Execution

Base Router

```
foreach ($modules as $moduleName) {
        $currentModuleName = $moduleName;
        $actionClassName = $this->actionList
             ->get($moduleName, $this->pathPrefix, $actionPath, $action);
        if (!$actionClassName || !is_subclass_of($actionClassName,
                                          $this->actionInterface)) {
             continue;
        }
        $actionInstance = $this->actionFactory->create($actionClassName,
                                          ['request' => $request]);
        break;
}
```

Front Controller

```
foreach ($this->_routerList as $router) {
     try {
          $actionInstance = $router->match($request);
          if ($actionInstance) {
               $request->setDispatched(true);
               $actionInstance->getResponse()->setNoCacheHeaders();
               $result = $actionInstance->dispatch($request);
               break;
          }
}
```

HOME

Magento U

Notes:

In the first example, the action instance is created in the base router with the code:

```
$actionInstance = $this->actionFactory->create($actionClassname,
                                          ['request' => $request]);
```

In the second example, the action instance is executed in the front controller.

This is the action instance that the base router returns:

```
$actionInstance = $router->match($request);
```

And this is how the action instance is executed:

```
$result = $actionInstance->dispatch($request)
```


3.34 URL Processing: Non-standard Router (CMS Example)

URL Processing | Non-standard Router (CMS Example)

```
public function match(\Magento\Framework\App\RequestInterface $request)
    {
        $identifier = trim($request->getPathInfo(), '/');
        $condition = new \Magento\Framework\Object(['identifier' => $identifier,
                                                    'continue' => true]);
        $this->_eventManager->dispatch(
            'cms_controller_router_match_before',
            ['router' => $this, 'condition' => $condition]
        );
        $identifier = $condition->getIdentifier();

        if ($condition->getRedirectUrl()) {
            $this->_response->setRedirect($condition->getRedirectUrl());
            $request->setDispatched(true);
            return $this->actionFactory->create(
                'Magento\Framework\App\Action\Redirect',
                ['request' => $request]
            );
        }
…
```

HOME

Magento U

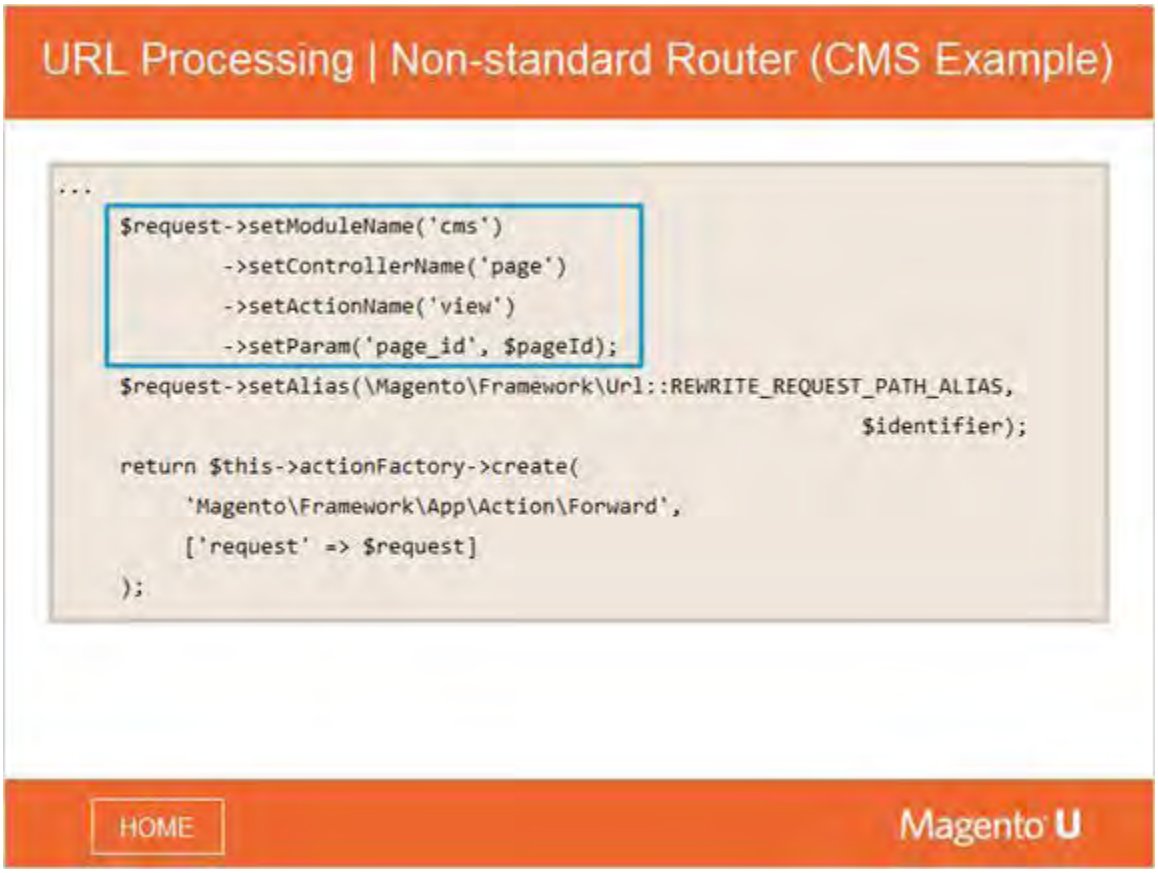
Notes:

Earlier, we looked in detail at the base router. Now, we will examine the CMS router to see an example of a non-standard router.

A non-standard router takes the URL, parses it, and converts it to a standard URL.

The most important part of the code in handling non-standard routers appears on the next slide.

3.35 URL Processing: Non-standard Router (CMS Example)



Notes:

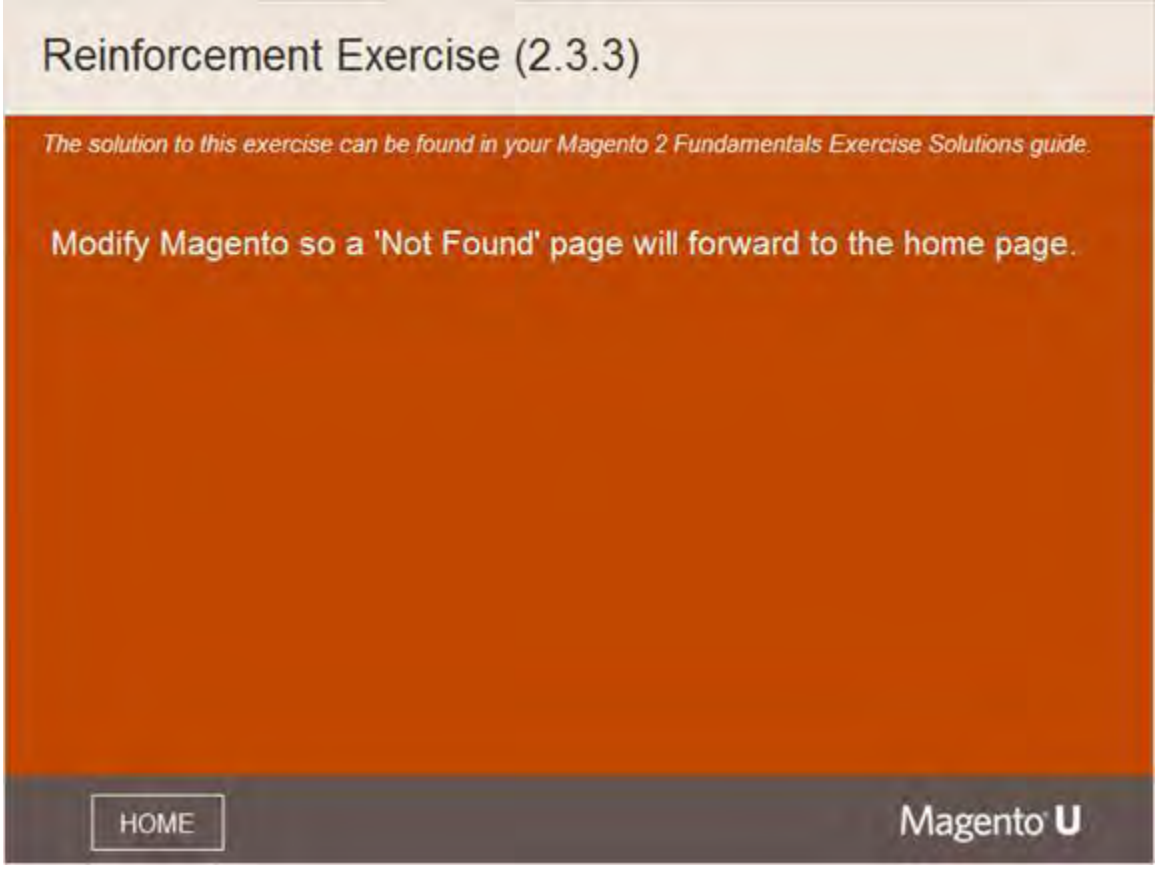
Continuing with the CMS example, the code to focus on is:

```
$request->setModuleName('cms')
         ->setControllerName('page')
         ->setActioName('view')
         ->setParam('page_id';$pageid);
```

This code demonstrates how a URL that does not match a standard router is parsed by a new router, which then converts it to a standard format.

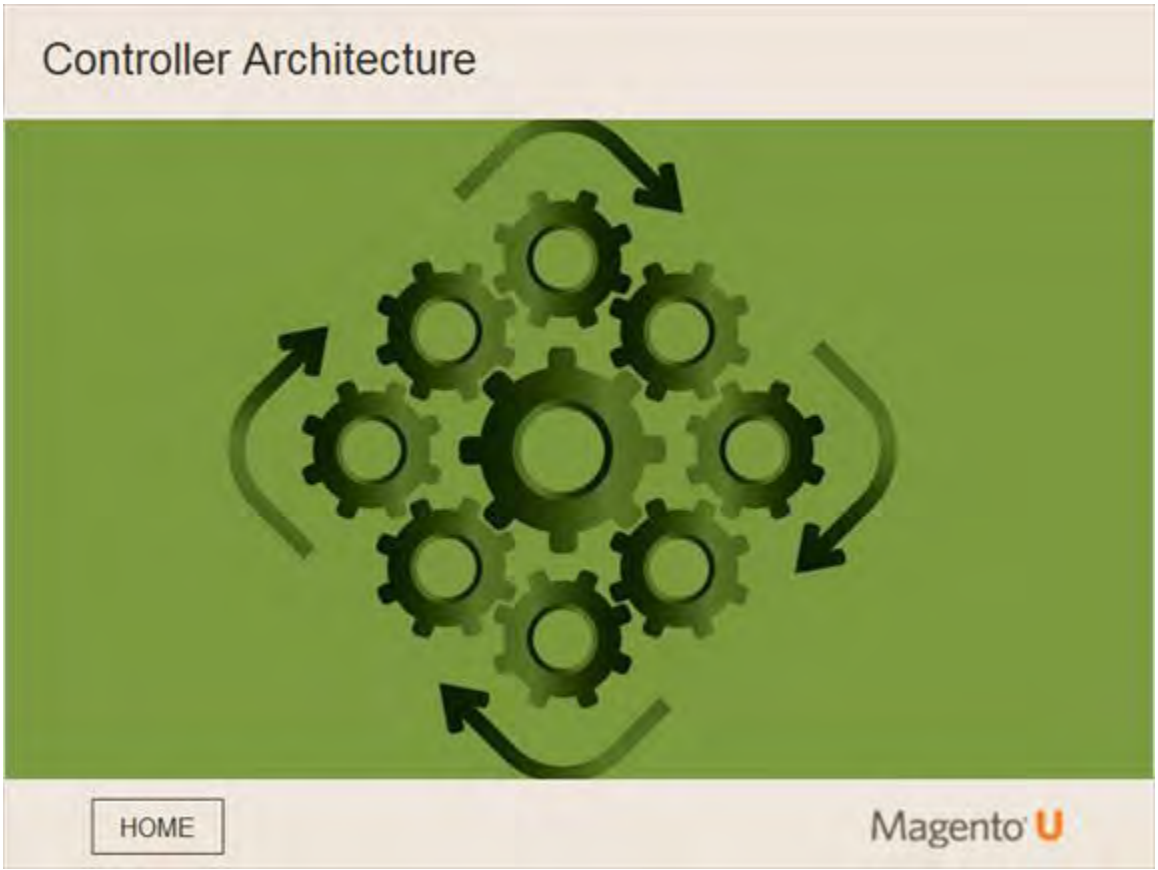
Other than the base router, most other routers are used only to convert a URL in non-standard format to a standard format.

3.36 Exercise 2.3.3



4. Controller Architecture

4.1 Controller Architecture



Notes:

This module discusses controller architecture.

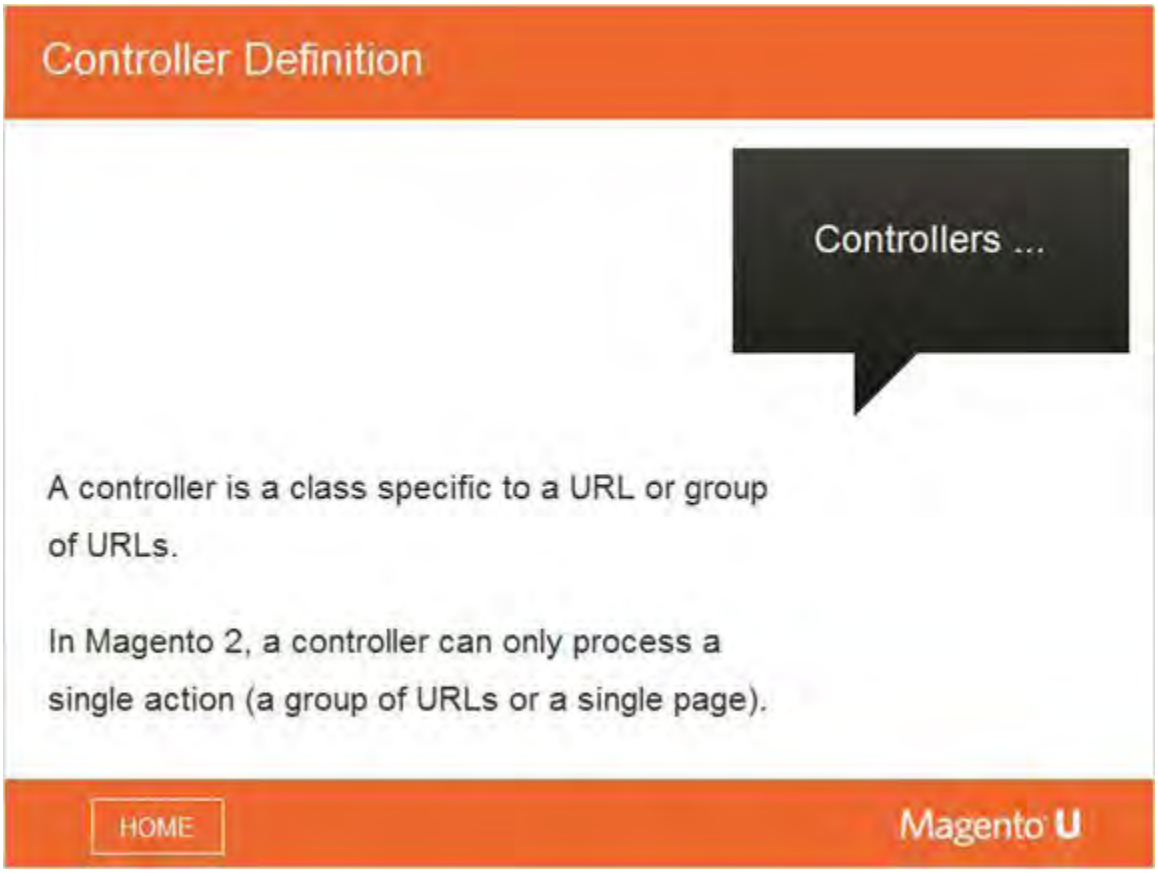
4.2 Module Topics

Notes:

Controller architecture describes which methods and classes are being executed by the controllers -- how they connect together, which classes the controller depends on, and so on.

The section on admin and frontend controllers will describe the workings of the front and backend controllers.

4.3 Controller Definition



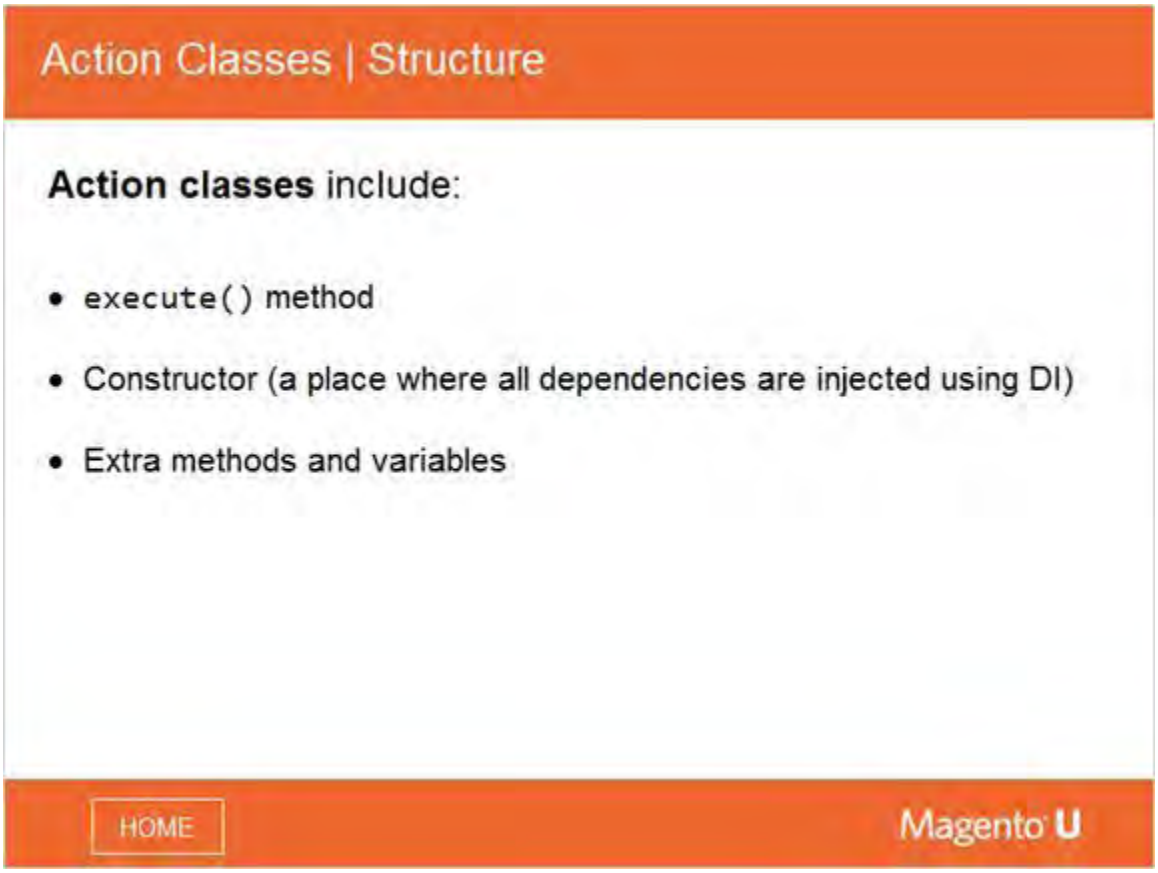
Notes:

Recall that a controller is a class specific to a URL or group of URLs (for example, all product pages). In Magento 2, a controller can only process a single action. It may have different parameters but it will be the same type of page. We have only one class per action.

Example of action class: `Magento_Catalog_Controller_Product_View`

In contrast, Magento 1 is able to process multiple types of pages in one class.

4.4 Action Classes: Structure

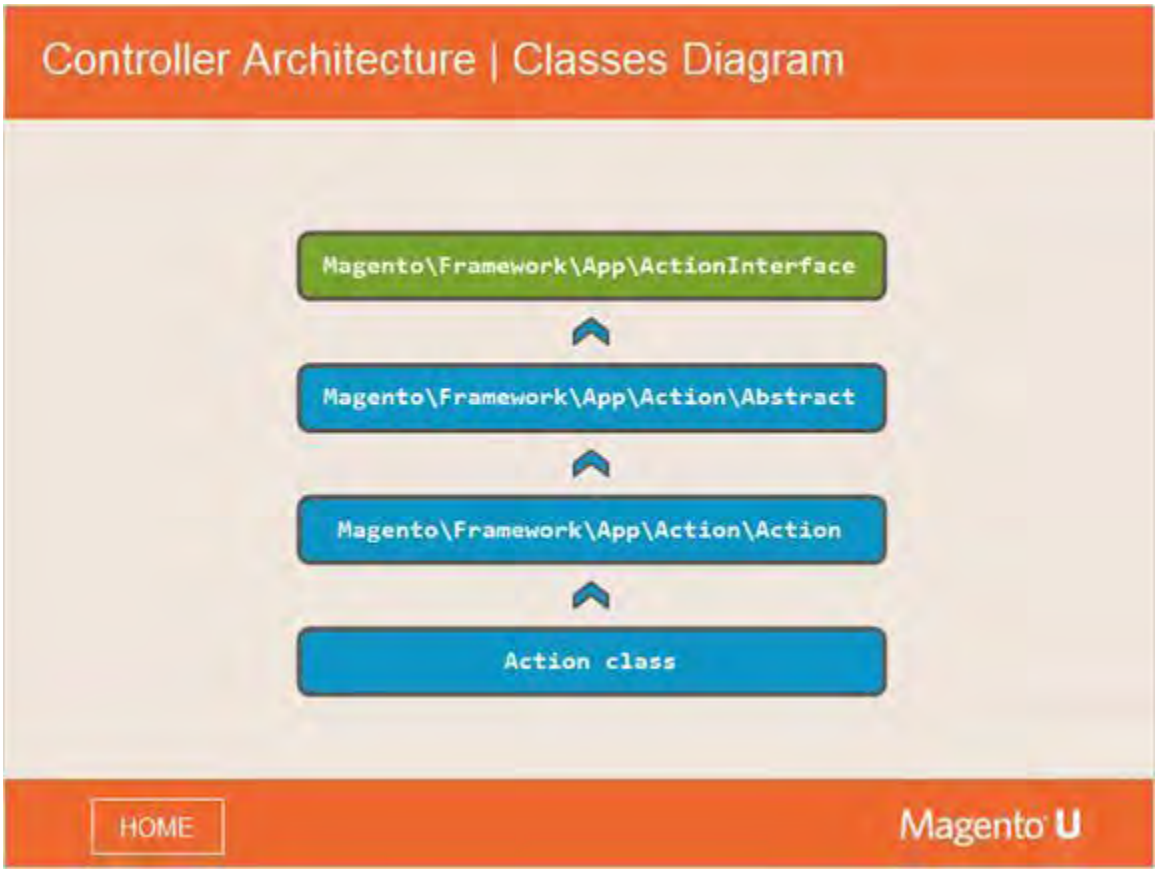


Notes:

If you look at the actual Magento 2 code – for example, the Category View Page – you will see it has a constructor, the required parameters, and an `execute()` method. Every controller has to have an `execute()` method. Every time a user hits a link, the `execute()` method will take over.

- The `execute()` method actually processes the URL. When someone enters a URL, it winds up in the action class (of a controller).
- Constructor: The action class holds dependencies to be injected, using dependency injection (DI). If a class needs to be inserted into the controller, it must be required in the constructor, and then injected by DI.
- Extra methods and variables: In addition to the `execute()` method, there may be other methods and other variables in a class. Controllers may contain additional methods that aid the `execute()` method. They may be broken up into smaller methods for efficiency, as one action class can extend another.

4.5 Controller Architecture: Classes Diagram



Notes:

An action class usually extends `Magento\Framework\App\Action`, but it is not always that simple.

Example:

```
Magento\Framework\App\Action
Magento\Catalog\Controller\Product\View
```

`Magento\Framework\App\Action\Action` extends `\AbstractAction`, which implements the action interface.

`ActionInterface` has only two methods: `dispatch()` and `response()`.

Note the interface is highlighted in green to distinguish it from the classes.

The diagram shows a hierarchy, which extends a class one at a time.

4.6 Front Controller: FrontControllerInterface

Front Controller | ActionInterface

```php
<?php
namespace Magento\Framework\App;

interface ActionInterface
{
    const FLAG_NO_DISPATCH = 'no-dispatch';
    const FLAG_NO_POST_DISPATCH = 'no-postDispatch';
    const FLAG_NO_DISPATCH_BLOCK_EVENT = 'no-beforeGenerateLayoutBlocksDispatch';
    const PARAM_NAME_BASE64_URL = 'r64';
    const PARAM_NAME_URL_ENCODED = 'uenc';
      // Dispatch request**
      //  @param RequestInterface $request
      //  @return \Magento\Framework\Controller\ResultInterface
      //                                      \ResponseInterface
      //  @throws Action\NotFoundException

    public function dispatch(RequestInterface $request);

      // Get Response object
      // @return ResponseInterface
    public function getResponse();
}
```

HOME

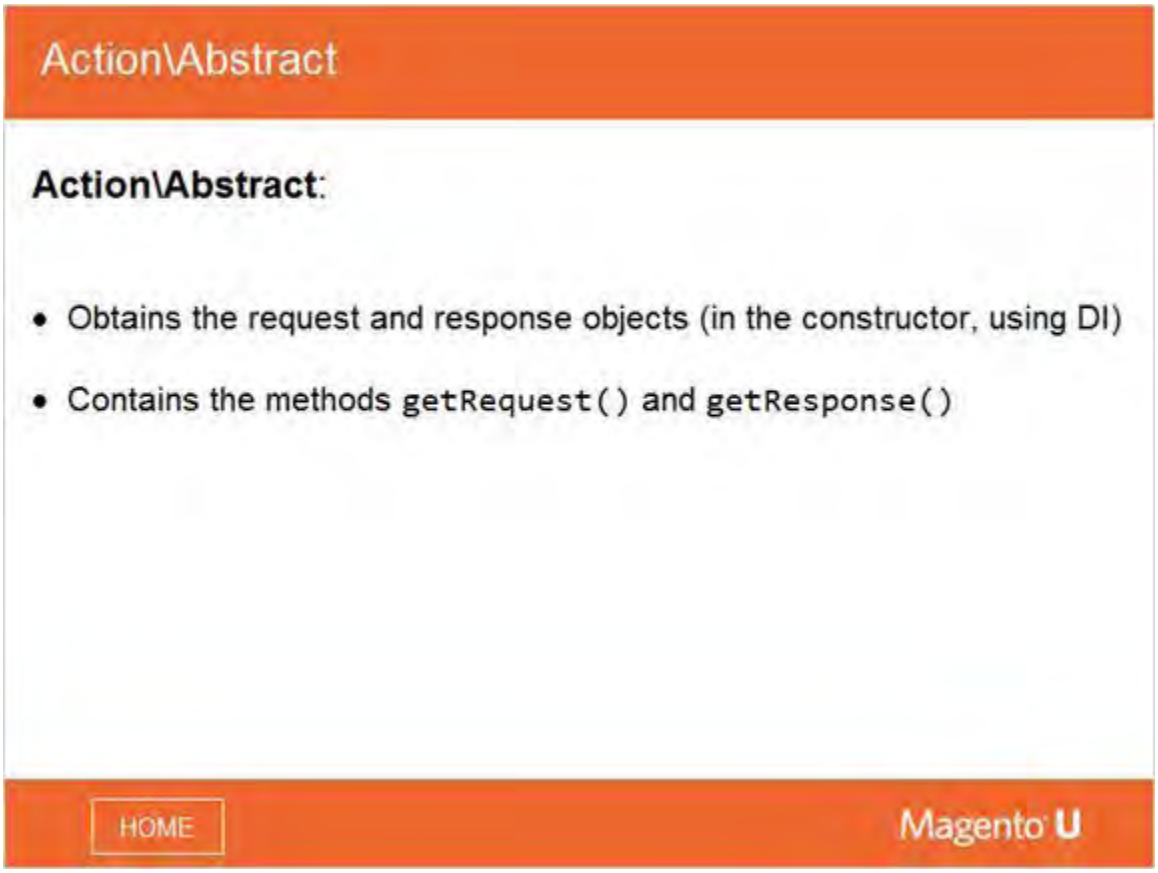
Magento U

Notes:

`ActionInterface` has only two methods: `dispatch()` and `getResponse()`.

This is not the `execute()` method.

4.7 Action\Abstract

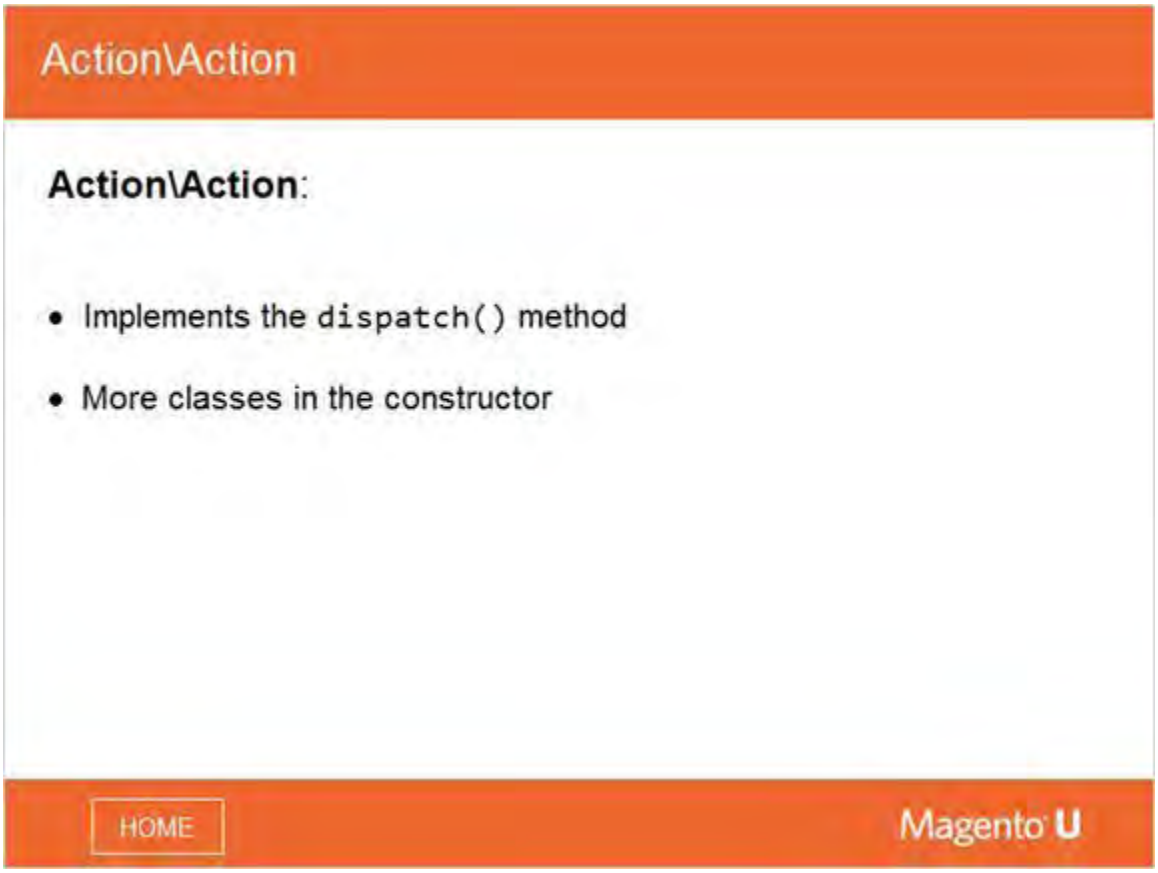


Notes:

`Action\Abstract` contains methods for obtaining request and response objects in the constructor using DI: `getRequest()` and `getResponse()`.

You must call the parent class when creating the action. Otherwise, the request and response objects will not be available.

4.8 Action\Action

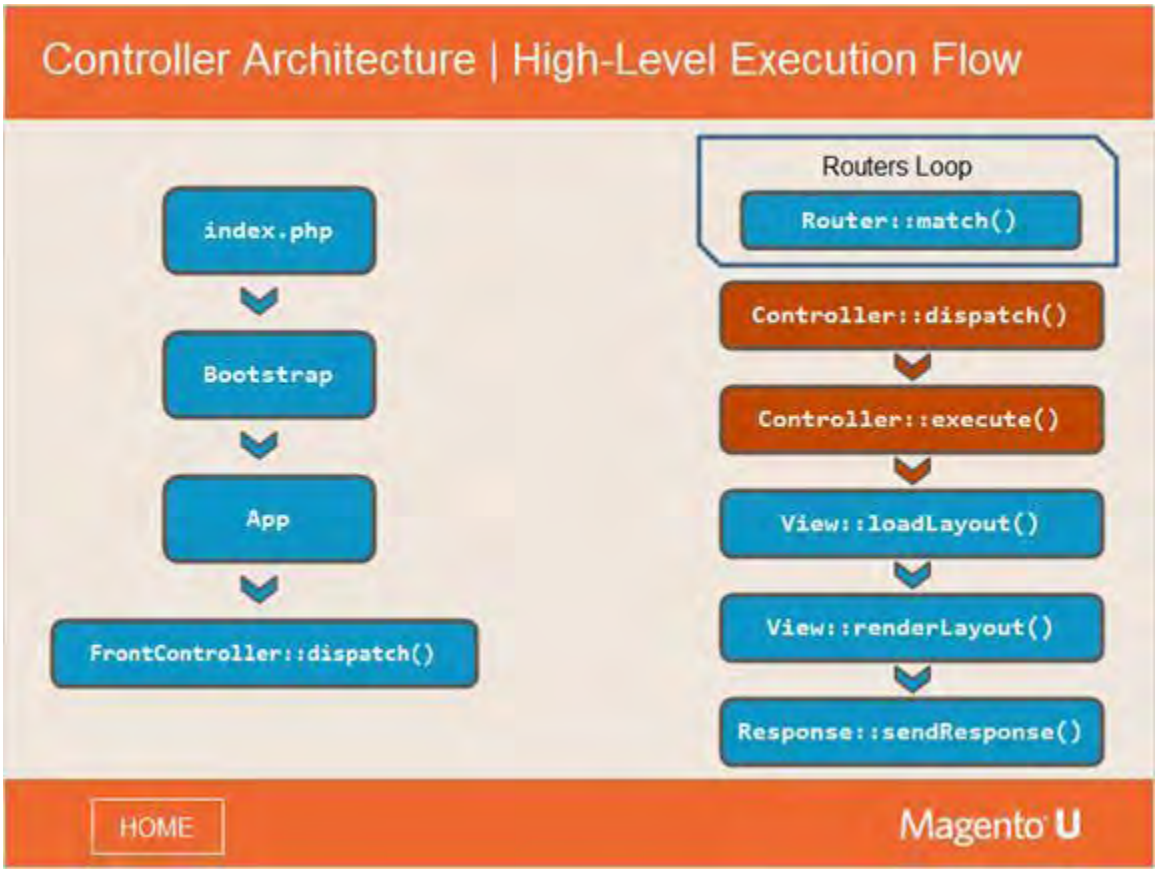


Notes:

- It is very important to extend every action class from `Action\Action`, not from `Action\Abstract` or `ActionInterface` because the `dispatch()` method is implemented here.

`Action\Action` includes different objects, like an object manager, a URL flag, and a redirect.

4.9 Controller Architecture: High-Level Execution Flow



Notes:

The router calls `Controller::dispatch()`, not `Controller::execute()`. The router will return an action instance from the controller.

The calls that the controller executes are inside the `dispatch()` method.

Summary of the Flow:

1. The `FrontController::dispatch()` is called from the `app::launch()`.
2. It finds the correct router, which then finds the correct action class.
3. The front controller calls an action class that then calls `dispatch()`. `Magento\Framework\App\Action\Action` is where the `dispatch()` method is implemented; it then calls the `execute()` method.

4.10 Action Wrappers

Action Wrappers

Action wrappers:

`Action/Action` is a class that almost every controller extends. If a controller doesn't extend this class but, for example, directly implements `ActionInterface`, it must implement the `dispatch()` method with specific logic.

As such logic is the same for all (native) controllers, they extend the `Action/Action` class and only implement the `execute()` method, not `dispatch()`.

- Examples: Catalog, Sales modules

HOME

Magento U

Notes:

Sometimes, specific action classes are used to extend the Action class. The reason is that a wrapper or intermediate class may have some logic that is needed for all action classes in the module - for example, the `goBack()` method.

In these modules, the controllers do not extend the `Action\Action` class directly.

The Product View controller extends the `Magento\Catalog\Controller\Product` class, which then extends the `Magento\Framework\Action\Action` class.

Note: The terms "action class" and "controller" are interchangeable.

4.11 Forward Function

Forward Function

```
protected function _forward($action, $controller = null, $module = null,
                            array $params = null)
    {
    $request = $this->getRequest();
    $request->initForward();

         if (isset($params)) {
              $request->setParams($params);
         }

         if (isset($controller)) {
              $request->setControllerName($controller);

        // Module should only be reset if controller has been specified
         if (isset($module)) {
              $request->setModuleName($module);
              }
         }

    $request->setActionName($action);
    $request->setDispatched(false);
    }
```

HOME

Magento U

Notes:

Magento 2 offers more structure around the forward function and redirects than Magento 1.

It is important to have both forward and redirect actions, and forward and redirect methods. They are not the same.

Why do we have both actions and methods for these functions?

Due to the way the routing loop is performed, in which the controller returns an action instance and requires a true argument, the action sets it to false within the loop.

Example:

Action Forward works with the request as it redirects the request to the next router.

Method Forward forwards directly. It takes the request, sets parameters, and sets a new controller name. This code is inside the action itself.

4.12 Redirect Function

Redirect Function

```
protected function _redirect($path, $arguments = [])
    {
        $this->_redirect->redirect($this->getResponse(), $path, $arguments);
        return $this->getResponse();
    }
```

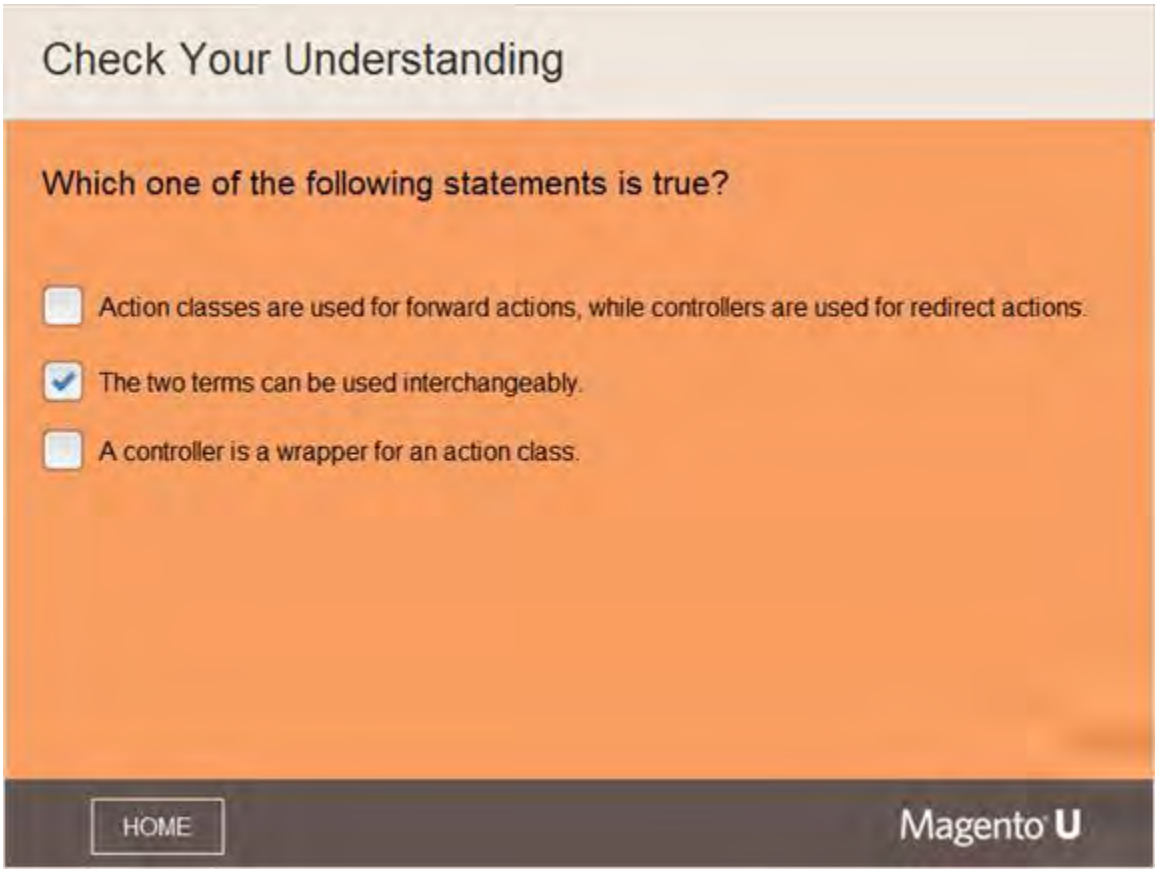
HOME

Magento U

Notes:

Redirect works with the response.

4.13 Check Your Understanding

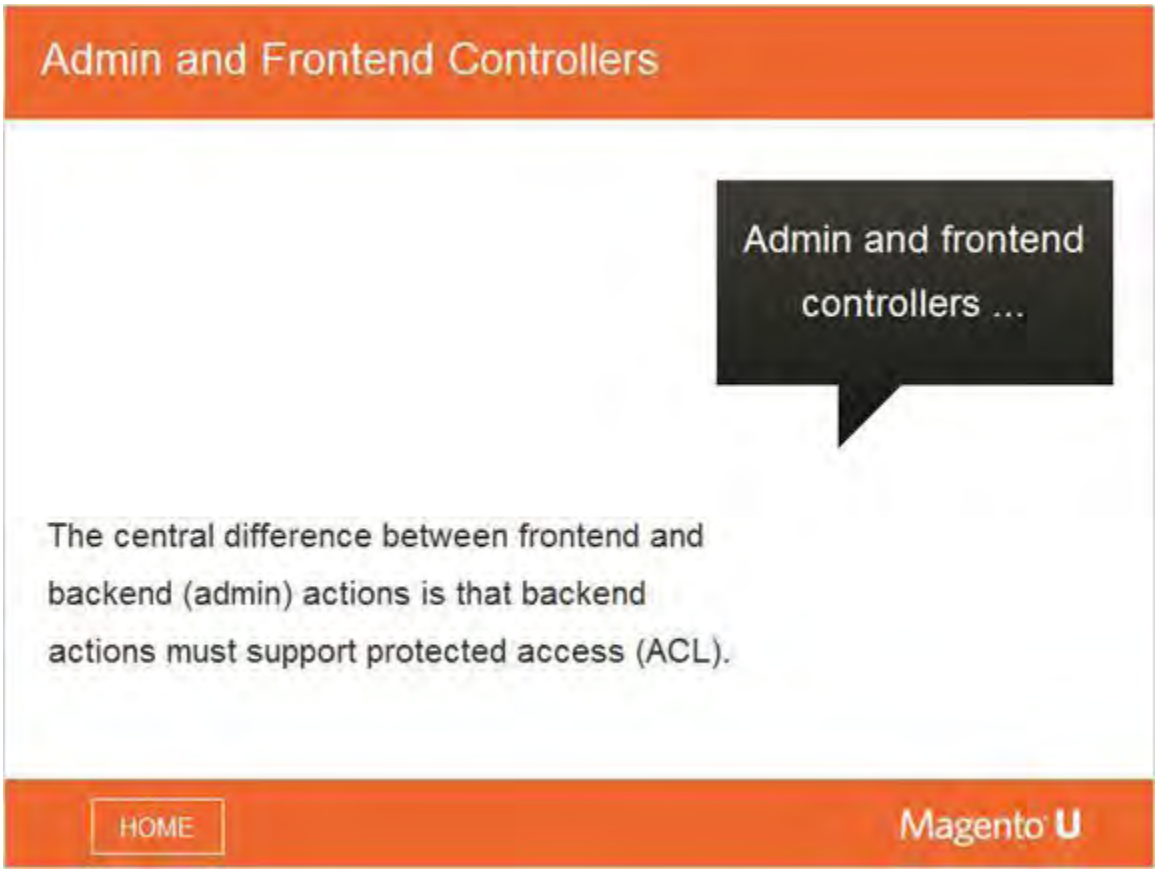


Notes:

The correct answer is only:

"The two terms can be used interchangeably".

4.14 Admin and Frontend Controllers

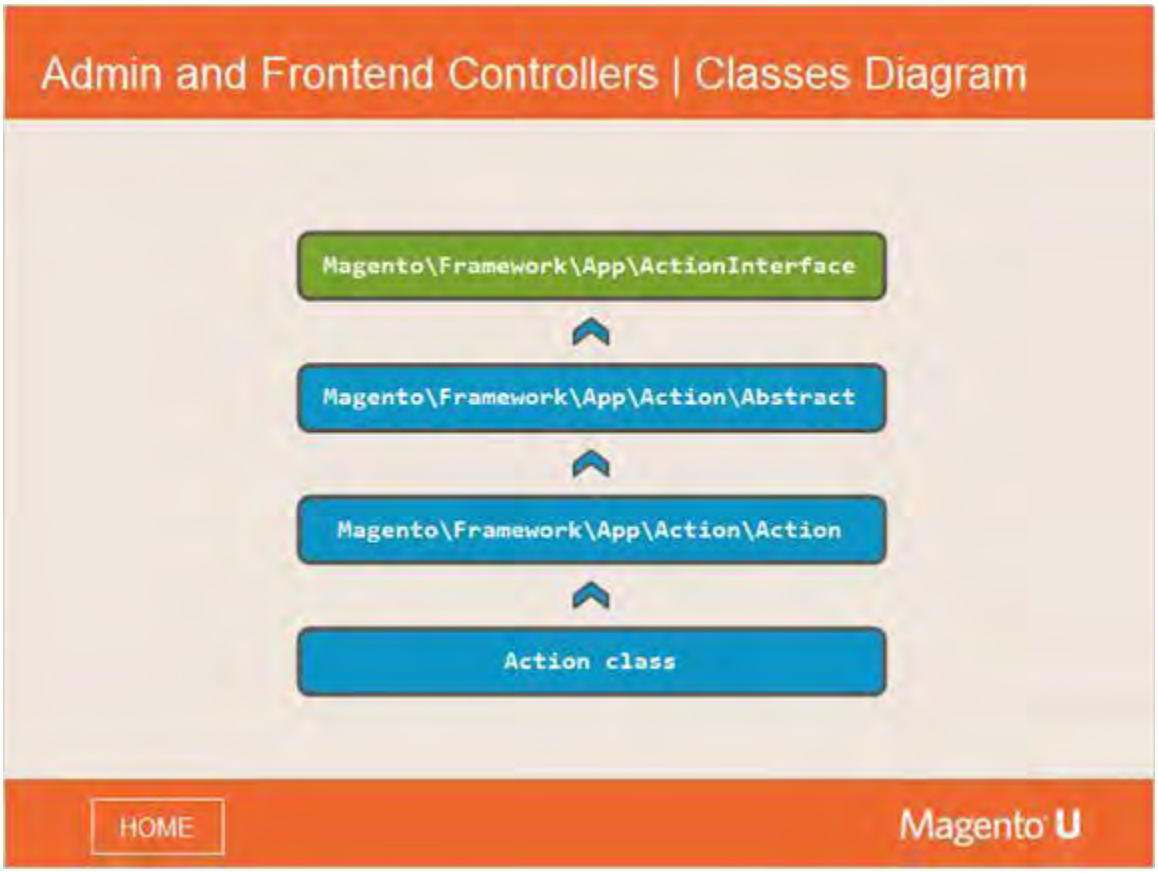


Notes:

To access the backend, you must log in, so these actions must support ACL, to be more protected.

Every controller has to facilitate this protection.

4.15 Admin and Frontend Controllers: Classes Diagram



Notes:

Recall the methods and actions from our controller classes diagram.

4.16 Admin and Frontend Controllers: Example 1

Admin and Frontend Controllers | Example 1

```php
<?php

namespace Magento\Catalog\Controller\Adminhtml\Product;

use Magento\Backend\App\Action;
use Magento\Catalog\Controller\Adminhtml\Product;

class Save extends \Magento\Catalog\Controller\Adminhtml\Product
{
    /**
     * @var Initialization\Helper
     */
    protected $initializationHelper;
```

HOME

Magento U

Notes:

In this code example, the Product Save controller extends `Magento\Catalog\Controller\Adminhtml\Product`, using `Magento\Backend\App\Action`.

We are going to examine how frontend controllers work - what they extend, what methods they contain, and what capabilities they have.

First, let's look at how admin controllers work.

4.17 Code Demonstration: Backend Action Class

Notes:

Let's take a look at: `Magento/Catalog/Controller/Adminhtml/Product/Save.php.` As you can see, it extends `adminhtml\product` and extends `Magento\backend\app\action.`

In Magento 2, `admin\module\adminhtml\module` has changed to backend, but in the controller folder the backend unit still starts with `adminhtml`.

As you can see, it extends `Magento\Backend\Application` but note it does not extend `Magento\Catalog\Controller\Product`, as other frontend actions might.

Let's go deeper into the folder `Magento/Backend/app/action.` You will notice that `Magento/Backend/App/Action.php` folder extends `Magento/Backend/App/AbstractAction` folder and now this method will extend `Magento/Framework/App/Action/Action` folder.

Let's look at that `App/Action` folder now. We can see the new `dispatch()` method, rewritten in this class. This is where the action happens. Going through this method, it goes to the parent dispatch. This is where the check for allowed access happens. In other words, it will check if it's allowed, and then you will not be allowed to go to the page if you are not authorized – you will be redirected to the login page. Located here is a new `dispatch()` method that goes to the parent. This is where the check for allowed access happens.

So, every action class doesn't have to extend the `Magento\Framework\App\Action\Action,` but rather the `Magento\Backend\App\AbstractAction` class. This class contains another `dispatch()` method rewritten from the `action\action,` and this dispatch method checks for multiple conditions. One of them is the `_isAllowed()` method, which checks if you're authorized to visit a specific page. If it fails, then you 'll be redirected to the login page, or an exception may happen. Magento 1 and 2 have a similar process in that they use the `_isAllowed()`method. Every backend controller, in order to make it work successfully, has to satisfy two conditions. It must extend `Magento\Backend\App\AbstractAction` class, and it must implement the `_isAllowed()`method.

4.18 Admin and Frontend Controllers: Example 2

Admin and Frontend Controllers | Example 2

```
public function dispatch(\Magento\Framework\App\RequestInterface $request)
    {
        if (!$this->_processUrlKeys()) {
            return parent::dispatch($request);
        }
        if ($request->isDispatched() && $request->getActionName() !== 'denied' &&
                !$this->_isAllowed() ) {
            $this->_response->setHeader('HTTP/1.1', '403 Forbidden');
            $this->_response->setHttpResponseCode(403);
            if (!$this->_auth->isLoggedIn()) {
                return $this->_redirect('*/auth/login');
            }
            $this->_view->loadLayout(['default', 'adminhtml_denied'], true, true,false);
            $this->_view->renderLayout();
            $this->_request->setDispatched(true);
            return $this->_response;
        }
        if ($this->_isUrlChecked()) {
            $this->_actionFlag->set('', self::FLAG_IS_URLS_CHECKED, true);
        }
        $this->_processLocaleSettings();
        return parent::dispatch($request);
    }
```

HOME

Magento U

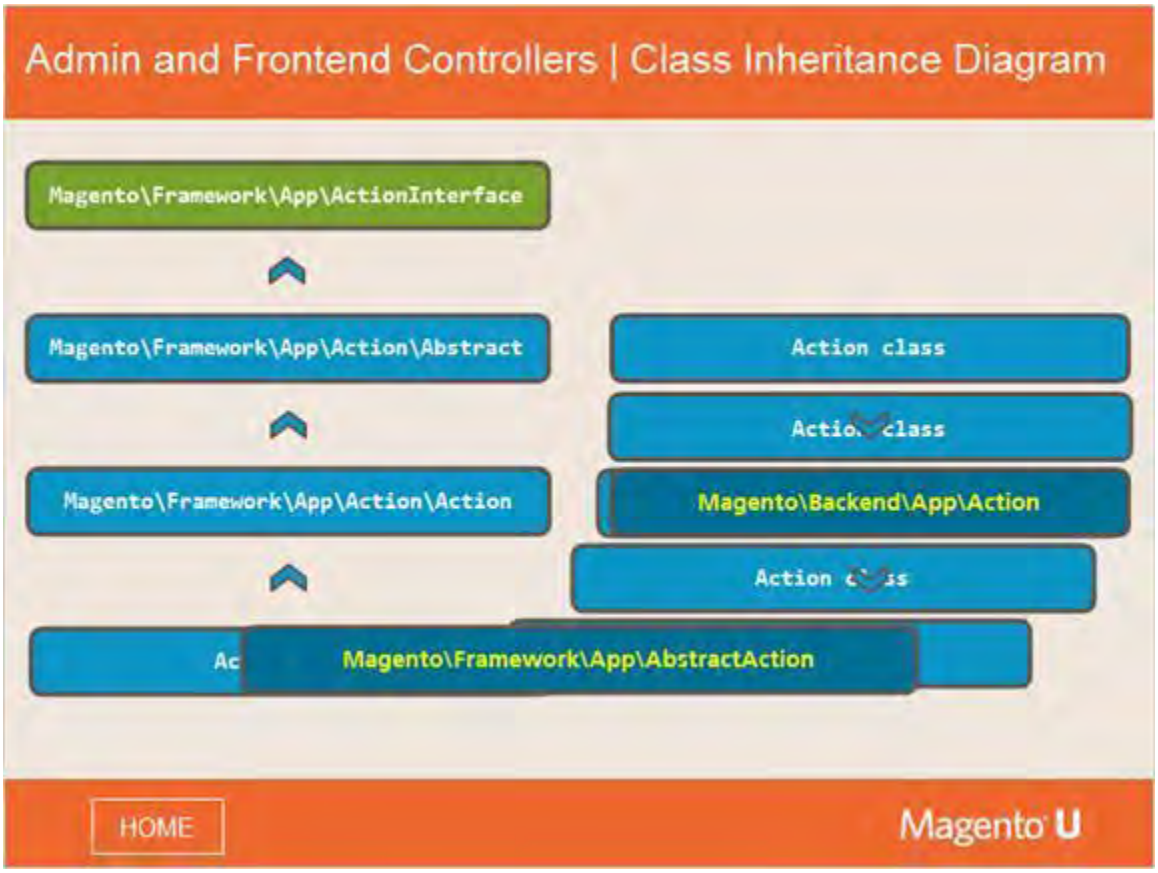
Notes:

In this code example, there is an extra check as to whether access is allowed or not. If the action is allowed, it will go to the parent `dispatch()` method, just as with the frontend.

Example:

```
Magento\Backend\App\AbstractAction
```


4.19 Admin and Frontend Controllers: Flow Diagram

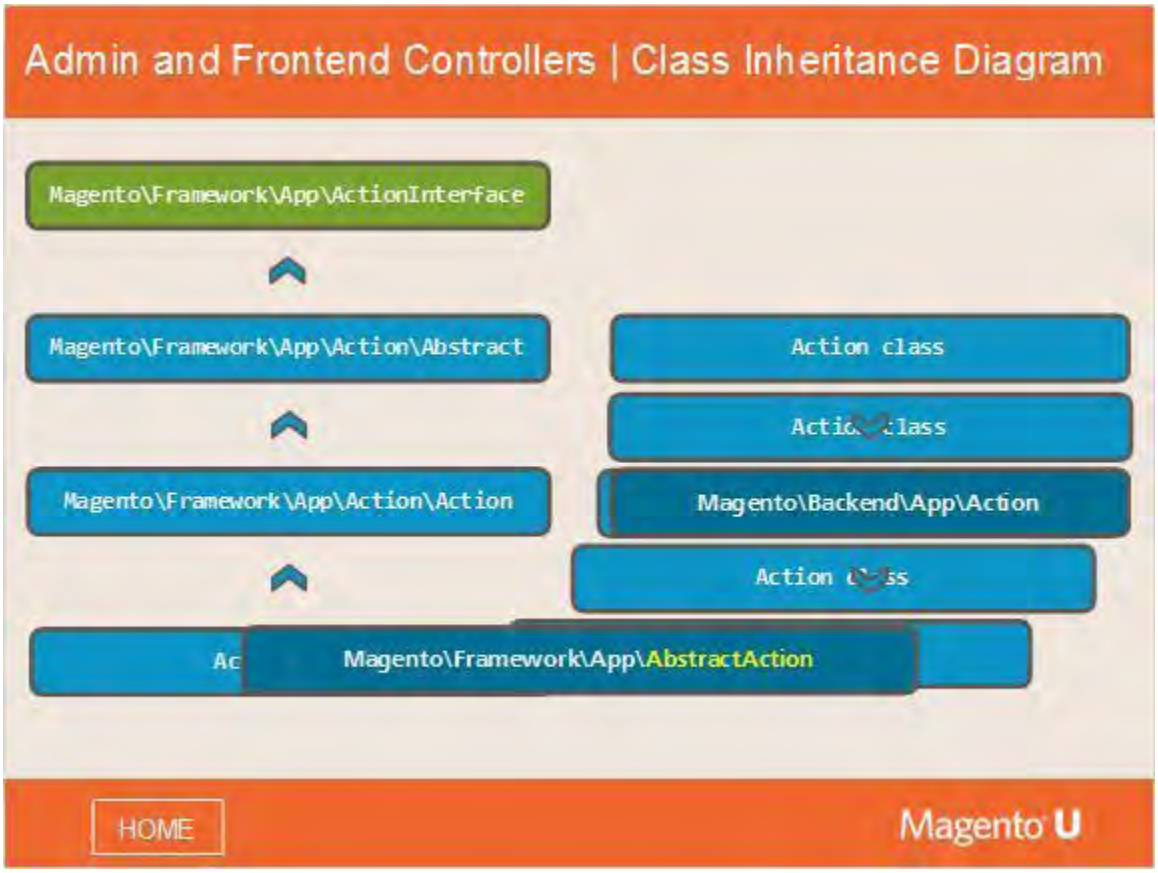


Notes:

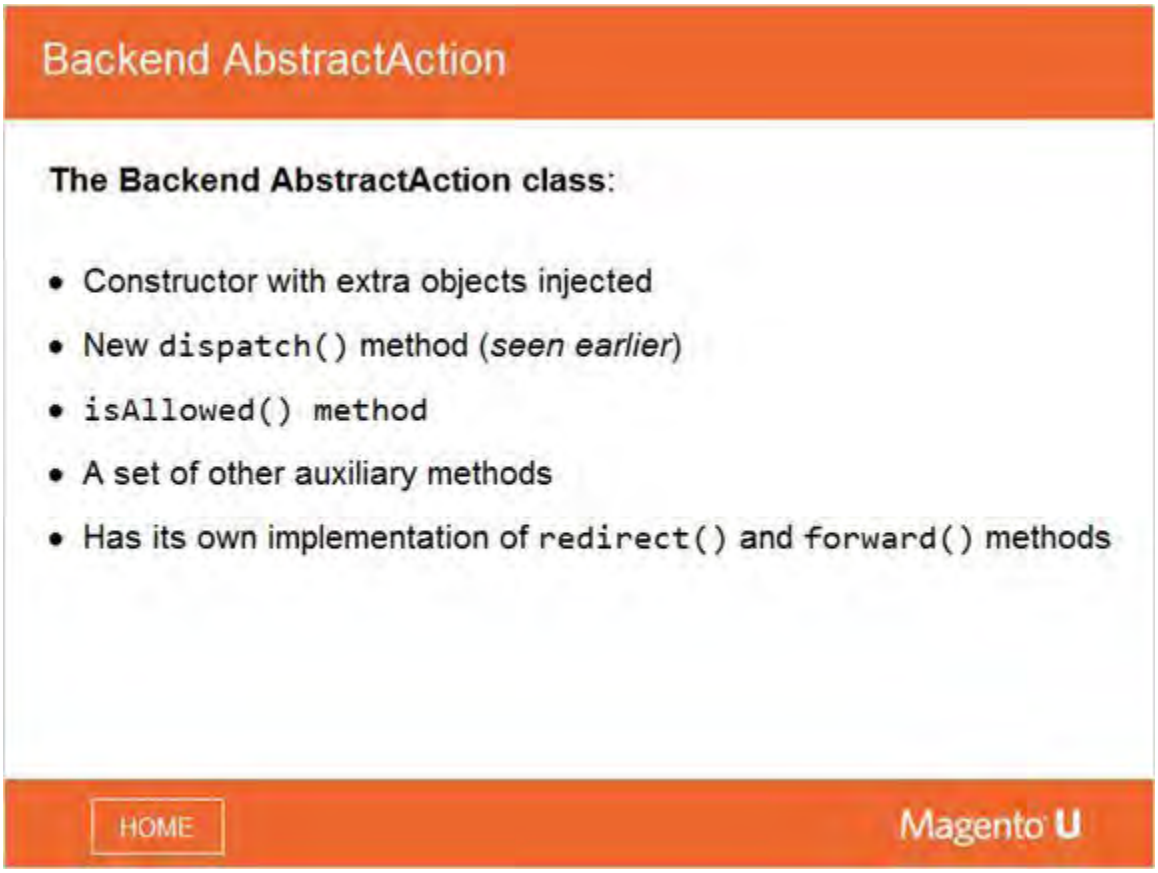
Looking at this updated flow diagram, you will notice that there is now a `Magento\Framework\App\AbstractAction` and the `Magento\Backend\App\Action`.

This is the difference between the Backend and the Frontend action flow.

Updated Diagram (Slide Layer)



4.20 Backend AbstractAction



Notes:

The backend `AbstractAction` class is a constructor with extra objects injected. The class contains the new `dispatch()` method, the `isAllowed()` method, and a set of other auxiliary methods.

It also owns the implementation of the backend `redirect()` and `forward()` methods.

4.21 Admin and Frontend Controllers: Constructor

Admin and Frontend Controllers | Constructor

```
public function __construct(Action\Context $context)
    {
        parent::__construct($context);
        $this->_authorization = $context->getAuthorization();
        $this->_auth = $context->getAuth();
        $this->_helper = $context->getHelper();
        $this->_backendUrl = $context->getBackendUrl();
        $this->_formKeyValidator = $context->getFormKeyValidator();
        $this->_localeResolver = $context->getLocaleResolver();
        $this->_canUseBaseUrl = $context->getCanUseBaseUrl();
        $this->_session = $context->getSession();
    }
```

HOME

Magento U

Notes:

This slide shows the constructor of the `Magento\Backend\App\ActionAbstract` class. Following common Magento 2 practice, the constructor defines key objects that will be used by the controller. However, we can see a slightly different approach with other classes.

Usually, there are many classes declared in the constructor, which are then instantiated by `ObjectManager` and injected using DI. Here we can see the context class, which provides access to required classes.

4.22 Code Demonstration: _isAllowed() Method

Notes:

How does the `_isAllowed()` method work?

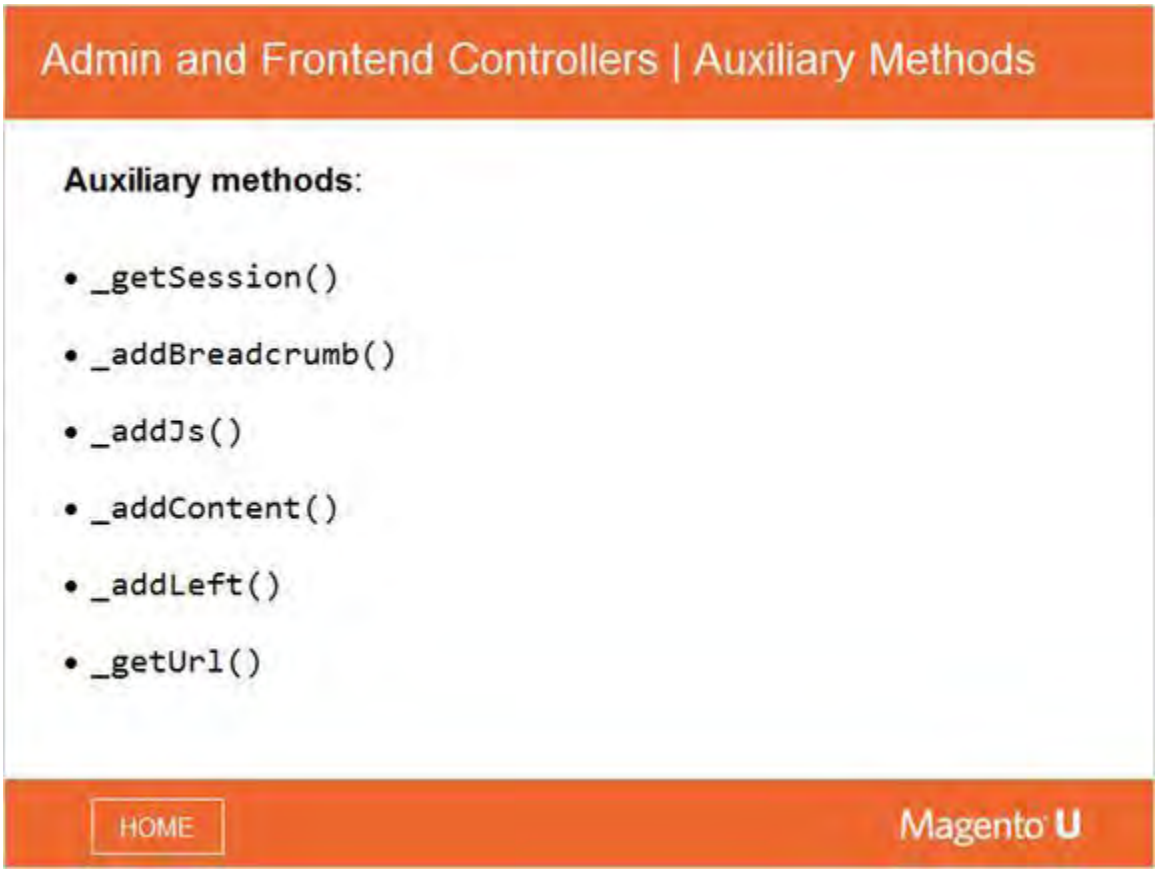
Before calling `Controller\Admin\Action`, the process will check this method. If the return value is `false`, then the `Controller\Admin\Action` will not be executed.

The _isAllowed() method is where you can implement restrictions. It can contain custom controllers and `_isAllowed()` methods to implement what you want.

The _isAllowed() method prevents the `execute()` method from performing, and checks the ACL. By default, it returns `true`.

If you don't implement this method in your action class, then it means that the page will be available through the Admin with no restrictions.

4.23 Admin and Frontend Controllers: Auxiliary Methods



Notes:

These are the auxiliary methods that are useful in the admin controllers.

4.24 Admin and Frontend Controllers: Forward and Redirect

Admin and Frontend Controllers | Forward and Redirect

```
protected function _redirect($path, $arguments = [])
    {
        $this->_getSession()->setIsUrlNotice(
            $this->_actionFlag->get('', self::FLAG_IS_URLS_CHECKED)
        );
        $this->getResponse()->setRedirect($this->getUrl($path, $arguments));
        return $this->getResponse();
    }

protected function _forward($action, $controller = null, $module = null,
                            array $params = null)
    {
        $this->_getSession()->setIsUrlNotice(
            $this->_actionFlag->get('', self::FLAG_IS_URLS_CHECKED)
        );
        return parent::_forward($action, $controller, $module, $params);
    }
```

HOME

Magento U

Notes:

Often an action class has to delegate execution of a specific URL to some other class.

There are two ways to do it: forward and redirect.

Forward: Processes an action inside the same request:

- Calls the `forward()` method, which will call the controller you want to process it (ex: specific URL).
- Invisible to the customer; sets new request and controller names.
- Calls the `dispatch()` method, goes back to loop with the new controller action and front name.

Redirect: Sends redirect header to the browser, which then reloads;

- URL will change to a new page.

4.25 Check Your Understanding

Check Your Understanding

Which of the following statements are true?

- [ ] When the `_isAllowed()` method checks the ACL, the default value it returns is "`false`", for safety.
- [x] The backend `AbstractAction` class contains the `_isAllowed()` method.
- [ ] The `_isAllowed` method controls the implementation of the backend `redirect()` and `forward()` methods.

HOME

Magento U

The correct statement is:

"The backend `AbstractAction` class contains the `_isAllowed()` method".

5. Working with Controllers

5.1 Working with Controllers

Notes:

This module will go into depth on how controllers work.

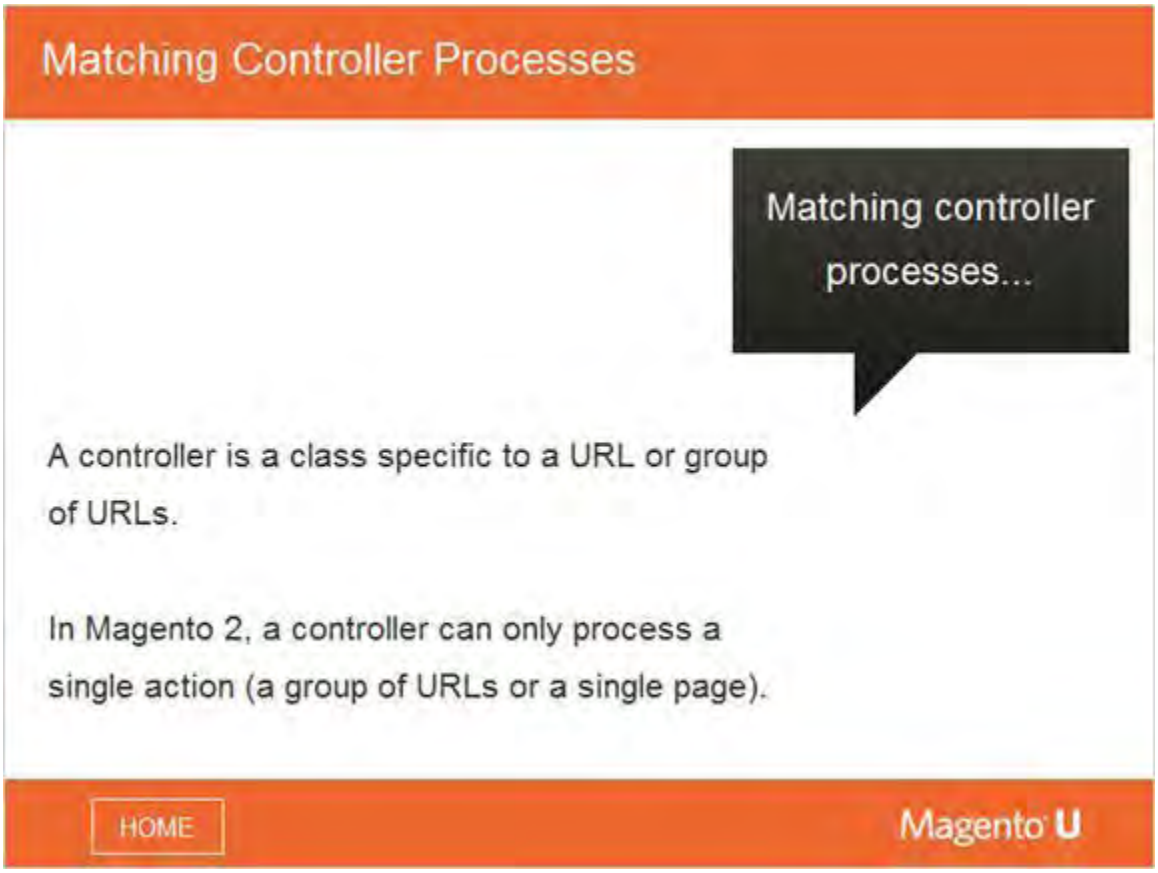
5.2 Module Topics

Notes:

Matching controller processes describes how Magento finds exactly the right controller for a URL. This is an important process for developers to understand, especially for debugging code.

Creating controllers explains how to create and register a new controller.

5.3 Matching Controller Processes

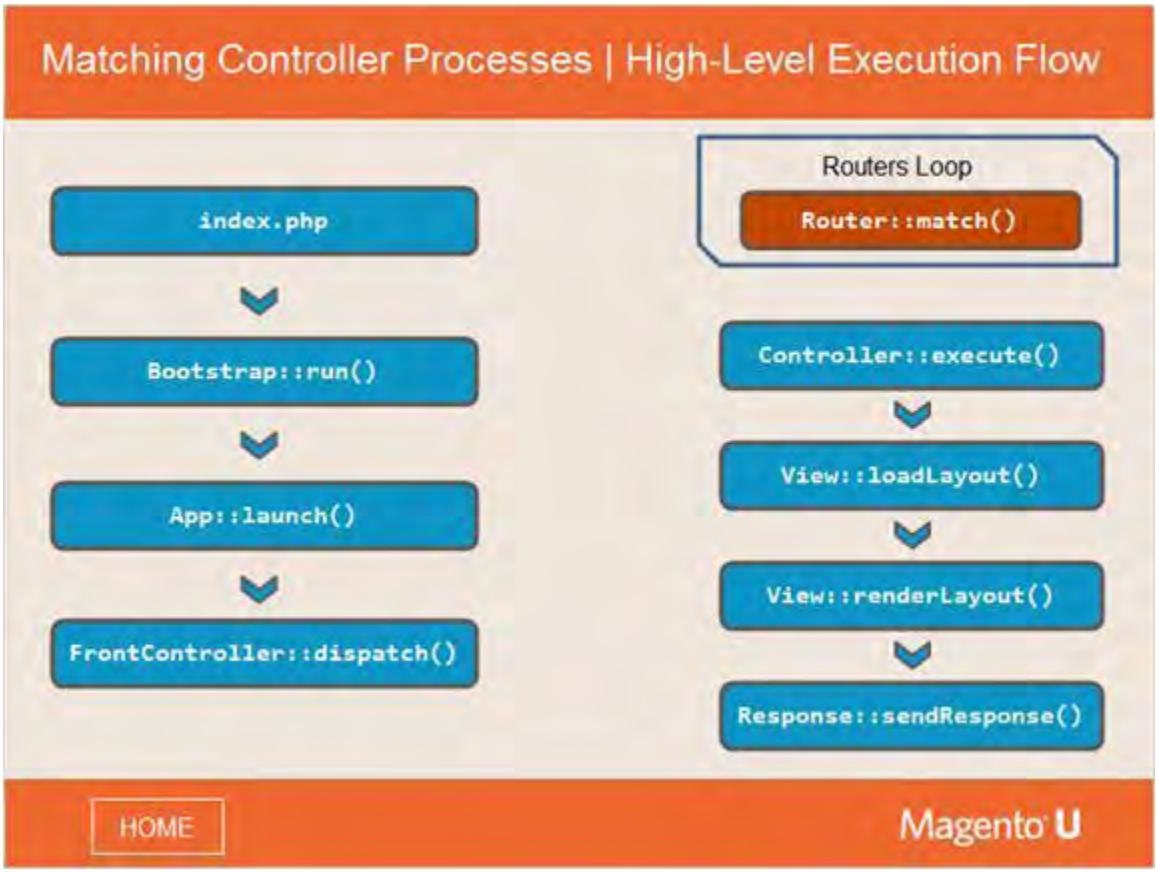


Notes:

Recall that a controller is a class specific to a URL or group of URLs, and that in Magento 2, a controller can only process a single action.

Example of action class: `Magento\Catalog\Controller\Product\View`

5.4 Matching Controller Processes: High-Level Execution Flow



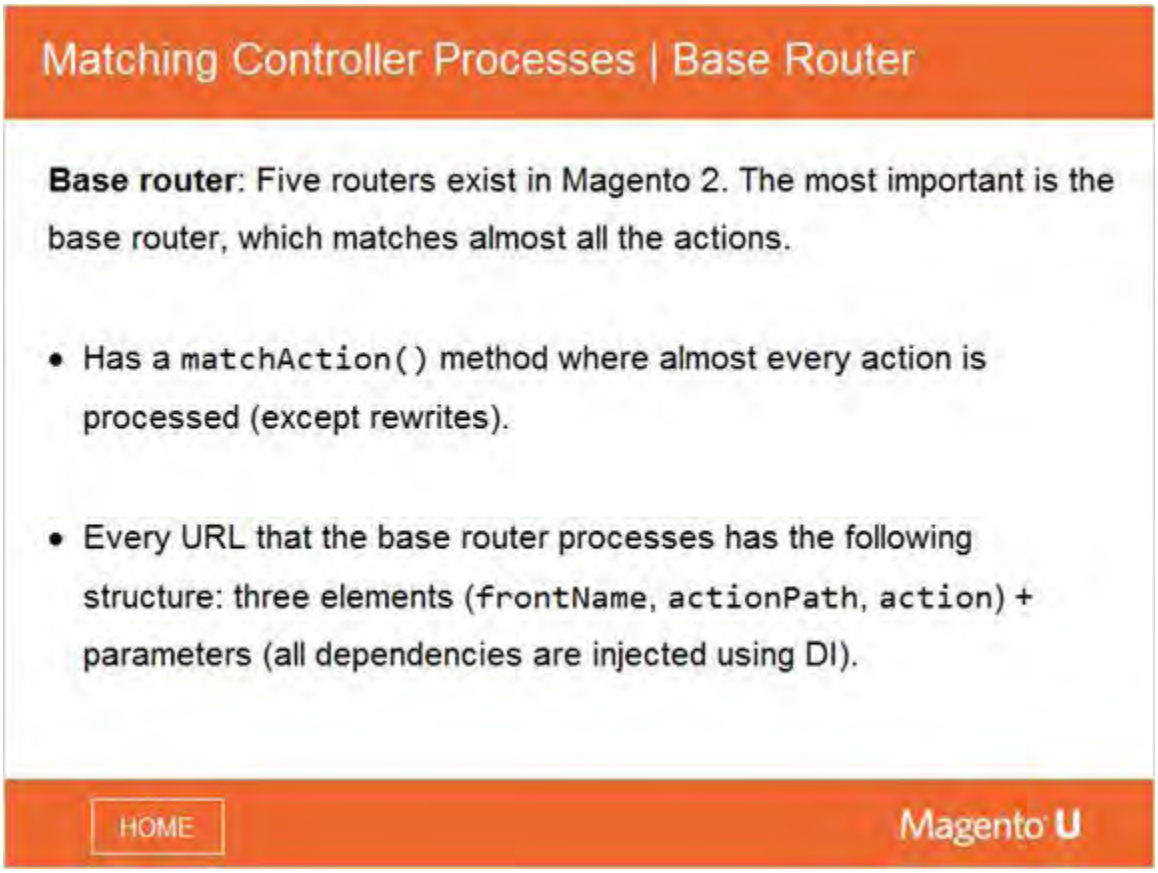
Notes:

We once again use the Execution Flow diagram to highlight where we are in the process. We are focused on the point where a customer enters a URL, and how the `Router::match()` responds. Inside the `dispatch()` method are the calls that the controller executes.

Summary of the Flow:

- `FrontController::dispatch()` is called from the app launch.
- It finds the correct router, which then finds the correct action class.
- The front controller will call an action class that calls the `dispatch()`method. The `Action\Action()` method is where the `dispatch()` method is implemented; it then calls the `execute()` method.

5.5 Matching Controller Processes: Base Router



Notes:

The base router has a `matchAction()` method where almost every action is processed (except rewrites - they are processed by a different router).

Every URL the base router processes has the following structure:

- 3 elements (frontName, actionPath, action)
- Parameters (all dependencies are injected using DI)

5.6 Code Demonstration: Base Router

Notes:

The first thing that happens is the front name is defined.

The modules come from the `routeConfig()`, which is injected by DI. The module then defines the action class and action.

Let's analyze what happens inside the `router::match()`. We start by going to `app/code/Magento/Core/App/Router/Base.php` In this version, this is how we access the base router, but in later installations, you may have to access it through the framework.

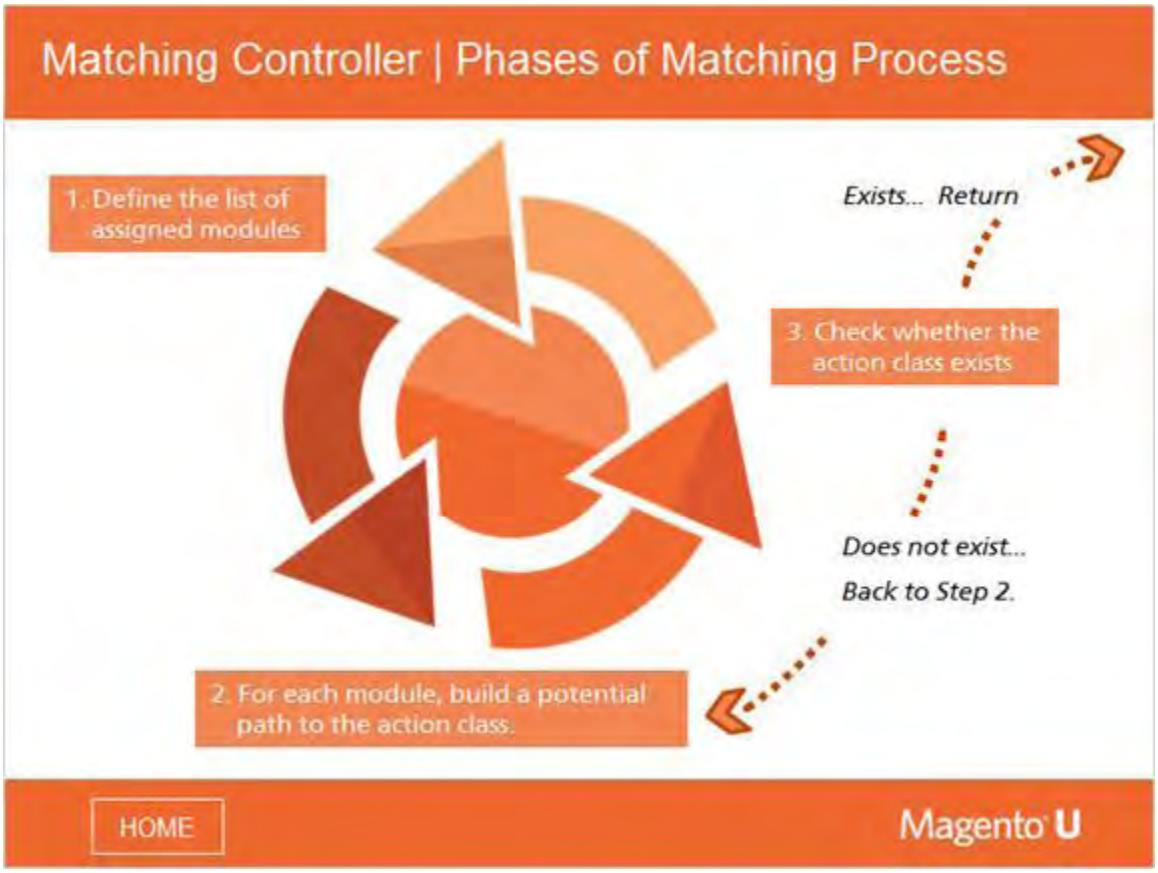
The `match()` method goes to `matchAction()`. `matchAction()` follows several steps.

First, it defines the front name and then defines the list of modules that are attached to this front name.

For each module method, it creates a path to the action class and then it defines what should be the class name.

After, if the action class exists, then it returns the class name; if does not exist, then it creates the action class name using the action factory.

5.7 Matching Controller: Phases of Matching Process



Notes:

The phases of the controller matching process:

1. Define the list of assigned modules.
2. For each module, build a potential path to the action class.
3. Check whether the action class exists.
4. If an action class exists, return – stop execution; if not, repeat step 2.

5.8 Matching Controller: Define List of Assigned Modules

Matching Controller | Define List of Assigned Modules

```
$moduleFrontName = $this->matchModuleFrontName($request, $params['moduleFrontName']);
    if (empty($moduleFrontName)) {
        return null;
    }
/**
 * Searching router args by module name from route using it as key
 */

$modules = $this->_routeConfig->getModulesByFrontName($moduleFrontName);
    if (empty($modules) === true) {
        return null;
    }
```

HOME

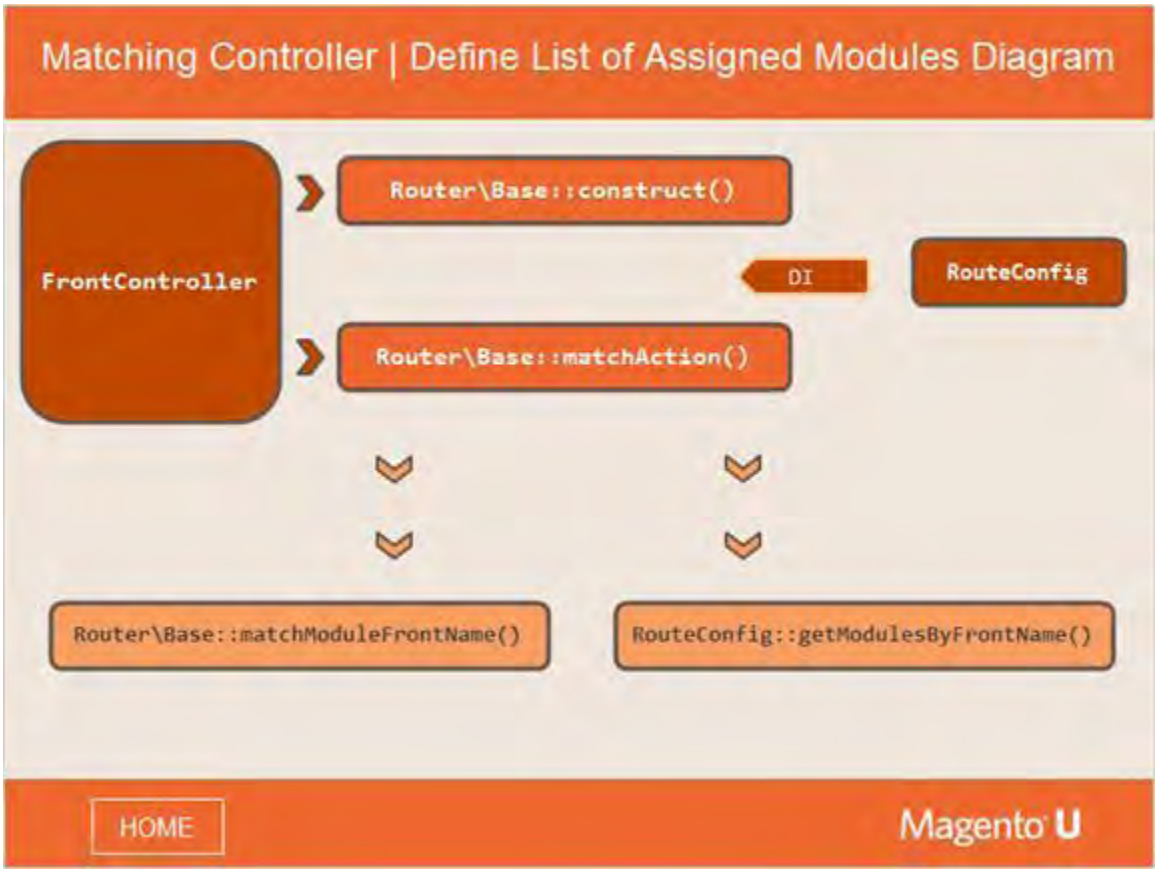
Magento U

Notes:

The first line shows the `moduleFrontName` and the line with the `routeConfig` class defines the list of assigned modules.

We will examine how the `routeConfig` class works in a diagram on the next slide.

5.9 Matching Controller: Define List of Assigned Modules Diagram



Notes:

The controller initiates the process of defining the list of assigned modules. The `routerConfig` is injected using DI.

`matchAction()` will perform two calls:

`Router\Base::matchModuleFrontName()` and `RouteConfig::getModulesByFrontName()`.

5.10 Matching Controller: Define List of Assigned Modules

Matching Controller | Define List of Assigned Modules

```
// Going through modules to find appropriate controller

$currentModuleName = null;
$actionPath = null;
$action = null;
$actionInstance = null;

$actionPath = $this->matchActionPath($request, $params['actionPath']);
$action = $request->getActionName() ? :
                    ($params['actionName'] ?: $this->_defaultPath->getPart('action'));
$this->_checkShouldBeSecure($request,
                    /' . $moduleFrontName . '/' . $actionPath . '/' . $action );
foreach ($modules as $moduleName) {
  $currentModuleName = $moduleName;
  $actionClassName = $this->actionList->get($moduleName, $this->pathPrefix,
                                            $actionPath, $action);
  if (!$actionClassName || !is_subclass_of($actionClassName,
                                           $this->actionInterface)) {
        continue;
  }
  $actionInstance = $this->actionFactory->create($actionClassName,
                                                 ['request' => $request]);
  break;
}
```

HOME

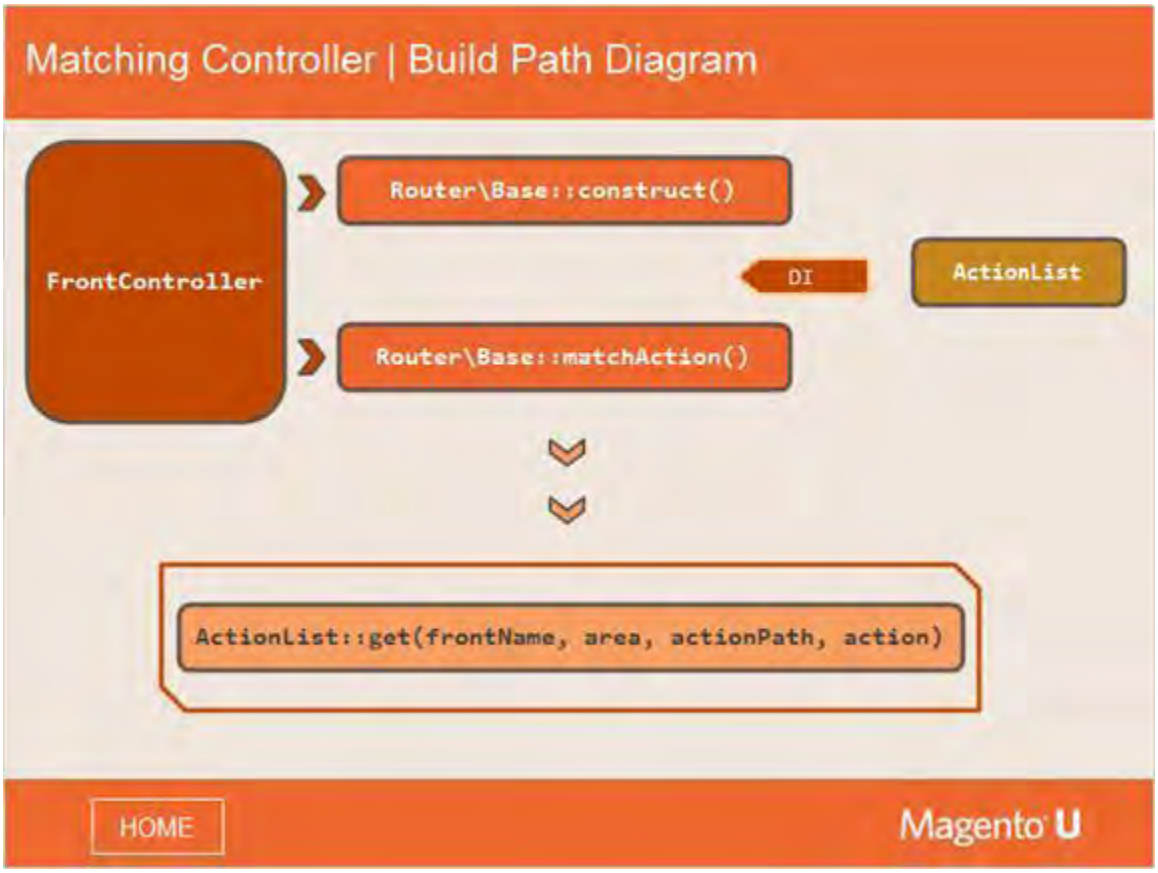
Magento U

Notes:

This code example demonstrates how the program goes through all the modules to find the appropriate controller.

It checks if there is an action name that exists for the current module. If not, then it defines the `actionClassName`.

5.11 Matching Controller: Build Path Diagram



Notes:

Returning to the Build Path diagram, you now see the `ActionList` (highlighted in gold) inserted into the process.

There is a loop in the `Router\Base::matchAction()` class.

In this loop, every module router calls `ActionList::get()` methods, with parameters: `frontName, area, actionPath, action`.

These correspond to 3 chunks of a URL: (`frontName/actionPath/action`) + area.

5.12 Matching Controller: Build Path Action List

Matching Controller | Build Path Action List

```
public function get($module, $area, $namespace, $action)
    {
        if ($area) {
            $area = '\\' . $area;
        }
        if (in_array(strtolower($action), $this->reservedWords)) {
            $action .= 'action';
        }
        $fullPath = str_replace(
            '_',
            '\\',
            strtolower(
            $module . '\\controller' . $area . '\\' . $namespace . '\\' . $action
            )
        );
        if (isset($this->actions[$fullPath])) {
            return is_subclass_of($this->actions[$fullPath], $this->actionInterface) ?
                $this->actions[$fullPath] : null;
        }

        return null;
    }
```

HOME

Magento U

Notes:

This code example shows how to build the path for the action list.

Reference:

`Magento/Catalog/Controller/Product/View.php`

5.13 Matching Controller: Check Action Class

Matching Controller | Check Action Class

```
public function get($module, $area, $namespace, $action)
    {
        if ($area) {
            $area = '\\' . $area;
        }
        if (in_array(strtolower($action), $this->reservedWords)) {
            $action .= 'action';
        }
        $fullPath = str_replace(
            '_',
            '\\',
            strtolower(
            $module . '\\controller' . $area . '\\' . $namespace . '\\' . $action
            )
        );
        if (isset($this->actions[$fullPath])) {
            return is_subclass_of($this->actions[$fullPath], $this->actionInterface) ?
                   $this->actions[$fullPath] : null;
        }

        return null;
    }
```

HOME

Magento U

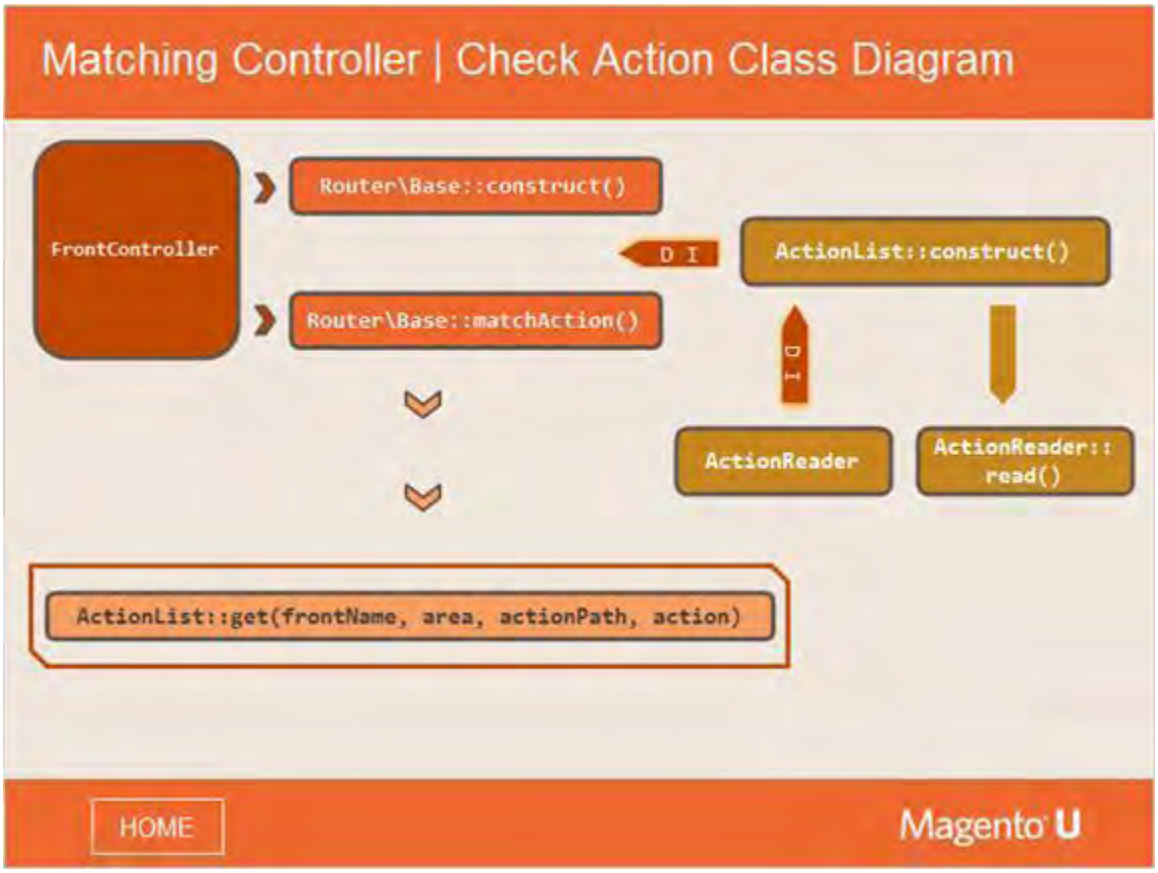
Notes:

Continuing with the action class example, this code is checking whether the action class exists.

Reference:

`Magento\Framework\App\Router\ActionList`

5.14 Matching Controller: Check Action Class Diagram



Notes:

Now, in the Action Class diagram, the next step is depicted. At the same time DI creates an `ActionList` class, an `ActionReader` class is also created, which is then passed to `ActionList`.

The `ActionList` constructor is executed, and the constructor will call `ActionReader::read()`, and then goes back to the `Router\Base` constructor.

The FrontController then calls `Router\Base::matchAction()`, which loops through all the modules and performs a `get()` call to the action list.

Because `ActionReader::read()` has already been called, an internal array of available actions has already been populated, so this `get()` will only check that array.

5.15 Matching Controller: Create Action Instance

Matching Controller | Create Action Instance

```
// Going through modules to find appropriate controller

$currentModuleName = null;
$actionPath = null;
$action = null;
$actionInstance = null;

$actionPath = $this->matchActionPath($request, $params['actionPath']);
$action = $request->getActionName() ? :
                    ($params['actionName'] ?: $this->_defaultPath->getPart('action'));
$this->_checkShouldBeSecure($request,
                        /' . $moduleFrontName . '/' . $actionPath . '/' . $action
 );
foreach ($modules as $moduleName) {
    $currentModuleName = $moduleName;
    $actionClassName = $this->actionList->get($moduleName, $this->pathPrefix,
                                                        $actionPath, $action);
  if (!$actionClassName || !is_subclass_of($actionClassName, $this->actionInterface)) {
      continue;
  }
  $actionInstance = $this->actionFactory->create($actionClassName, ['request' =>
                                                          $request]);
  break;
}
```

HOME

Magento U

Notes:

Continuing the example, this code shows the creation of the action instance.

Reference:

`Magento\Framework\App\Router\ActionList`

5.16 Matching Controller: Noroute

Matching Controller | Noroute

```
if (null == $actionInstance) {
    $actionInstance = $this->getNotFoundAction($currentModuleName, $request);
    if (is_null($actionInstance)) {
            return null;
    }
    $action = 'noroute';
}

// set values only after all the checks are done

$request->setModuleName($moduleFrontName);
$request->setControllerName($actionPath);
$request->setActionName($action);
$request->setControllerModule($currentModuleName);
$request->setRouteName($this->_routeConfig->getRouteByFrontName($moduleFrontName));
if (isset($params['variables'])) {
      $request->setParams($params['variables']);
}
return $actionInstance;
```

HOME

Magento U

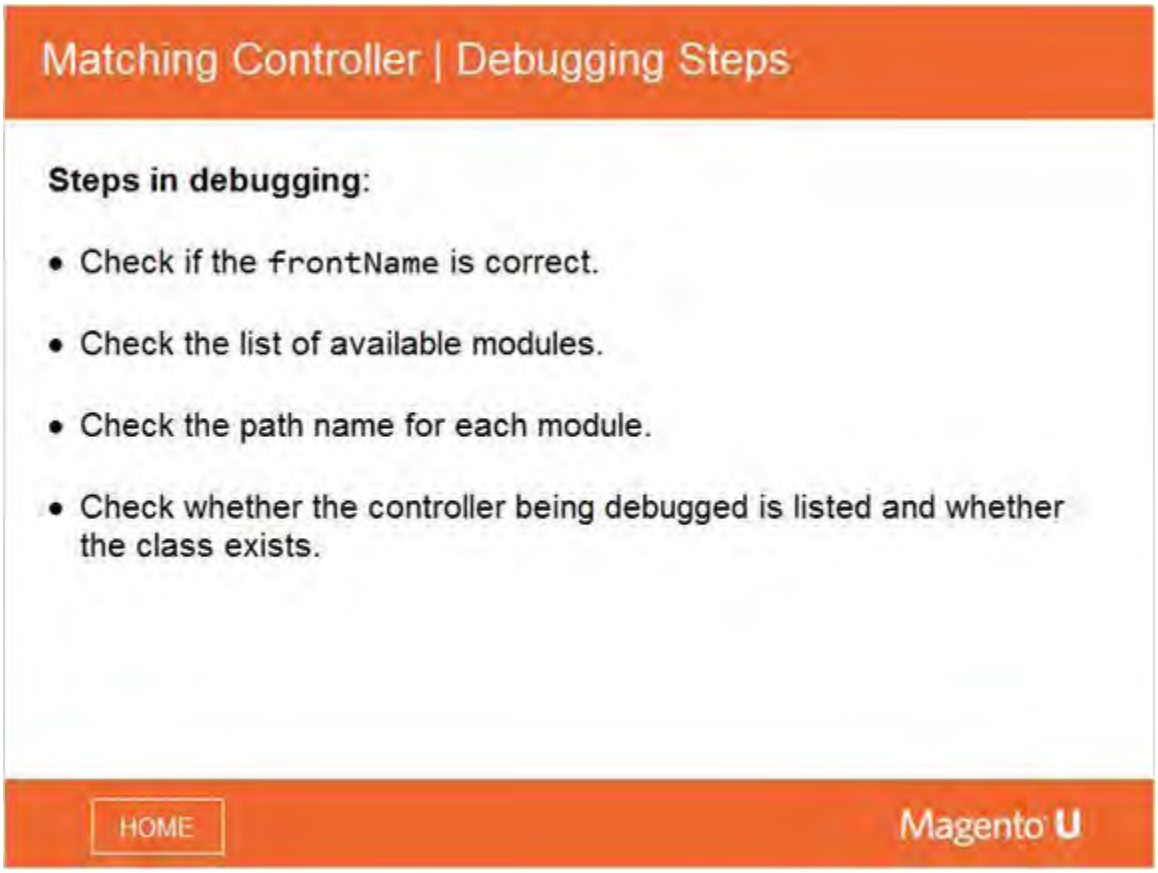
Notes:

The code displays what happens if nothing has been matched.

In this case, the `actionInstance` will be changed to `NotFoundAction`, and the action changed to `noroute`.

Then, the request objects are populated with actions, wherever they appear.

5.17 Matching Controller: Debugging Steps

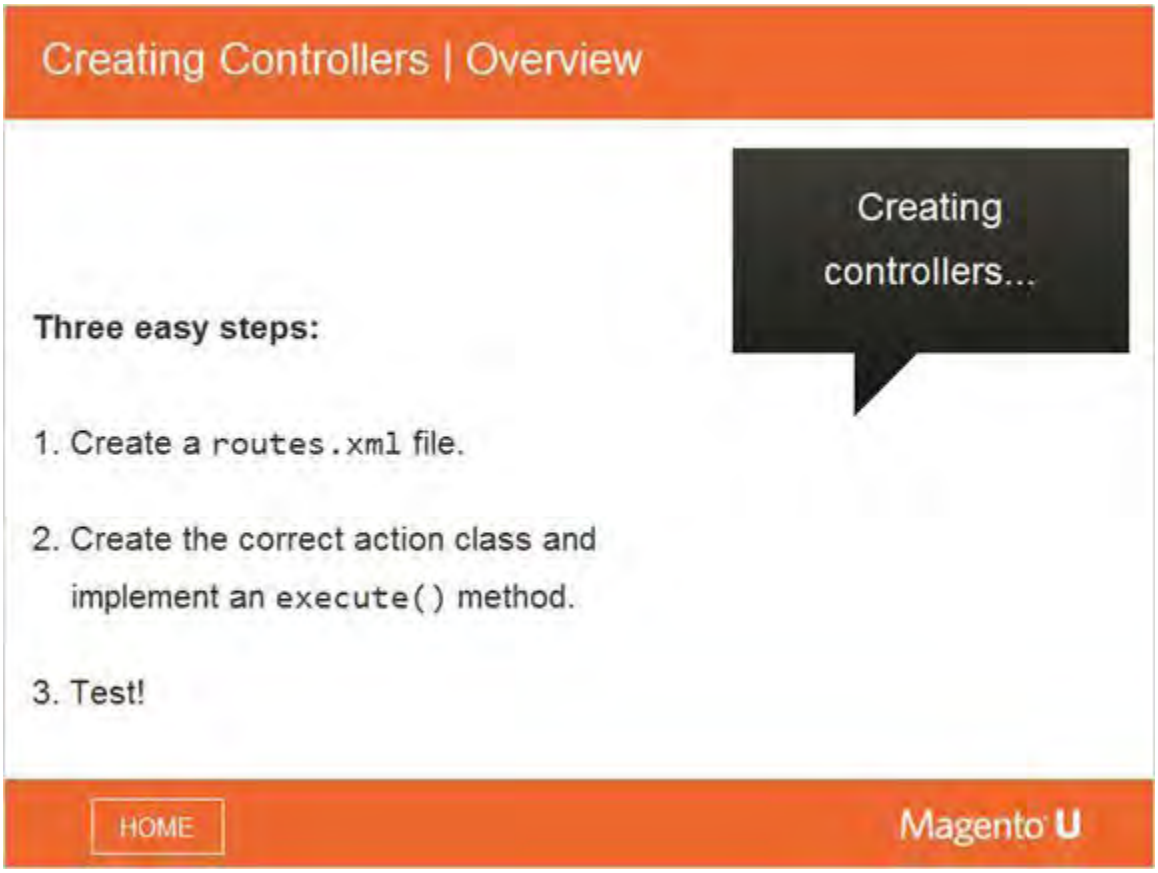


Notes:

There are recommended steps to take, in a specific order, when debugging the matching controller path.

For example, if you want to know the `frontName` is correct, then you should check the `Router\Base::matchModuleFrontName()` and `routeConfig::getModulesByFrontName()`.

5.18 Creating Controllers: Overview



Notes:

In order to create a controller, we have to follow three steps:

1. Create a `routes.xml` file.
2. Create the correct action class and implement an `execute()` method.
3. Test the new controller.

5.19 Creating Controllers: Routes.xml

Notes:

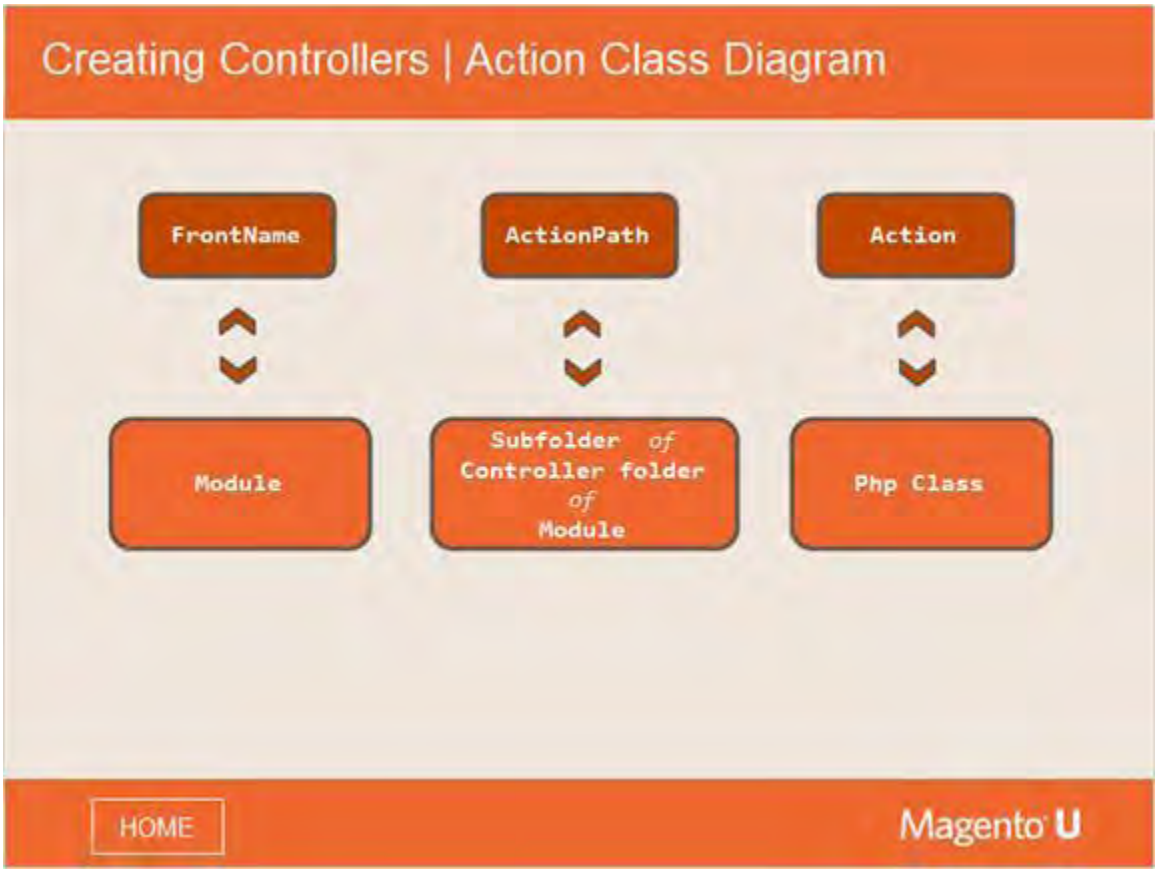
Let's take a look at the native example in `Magento/Catalog/etc/frontend/routes.xml`.

The code example demonstrates defining the ID of the router, the `FrontName`, and the module name.

You define the Id of the router (in this case, `standard` means `base`) in the same code in which you define a `frontName` and module name.

This example tells Magento that this module is registered to process the URL that starts with "catalog".

5.20 Creating Controllers: Action Class Diagram



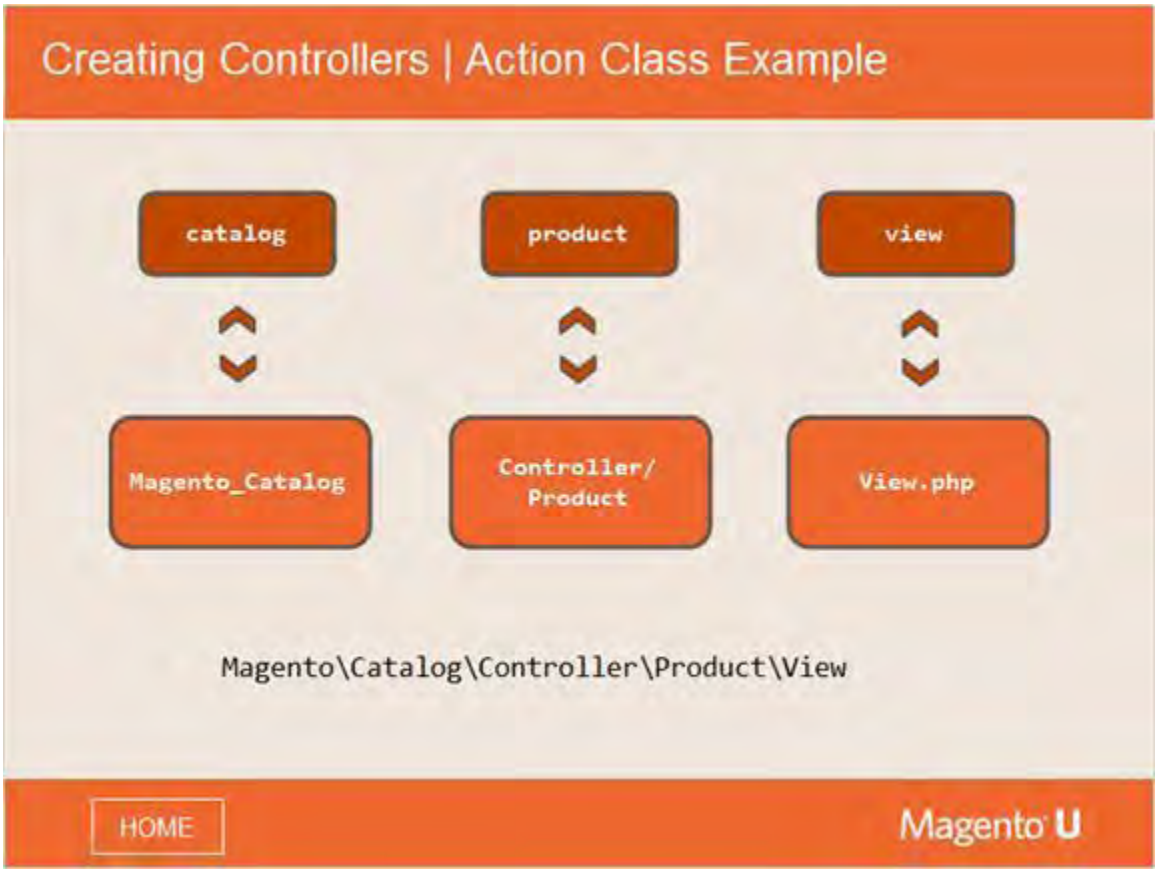
Notes:

Recall that a URL consists of 3 chunks: frontName, actionPath, actionName.

The rule on how a URL will be converted to an action class name is as follows:

- the `FrontName` is connected to the module.
- the `ActionPath` is connected to the folder.
- the `action` is connected to the PHP class.

5.21 Creating Controllers: Action Class Example



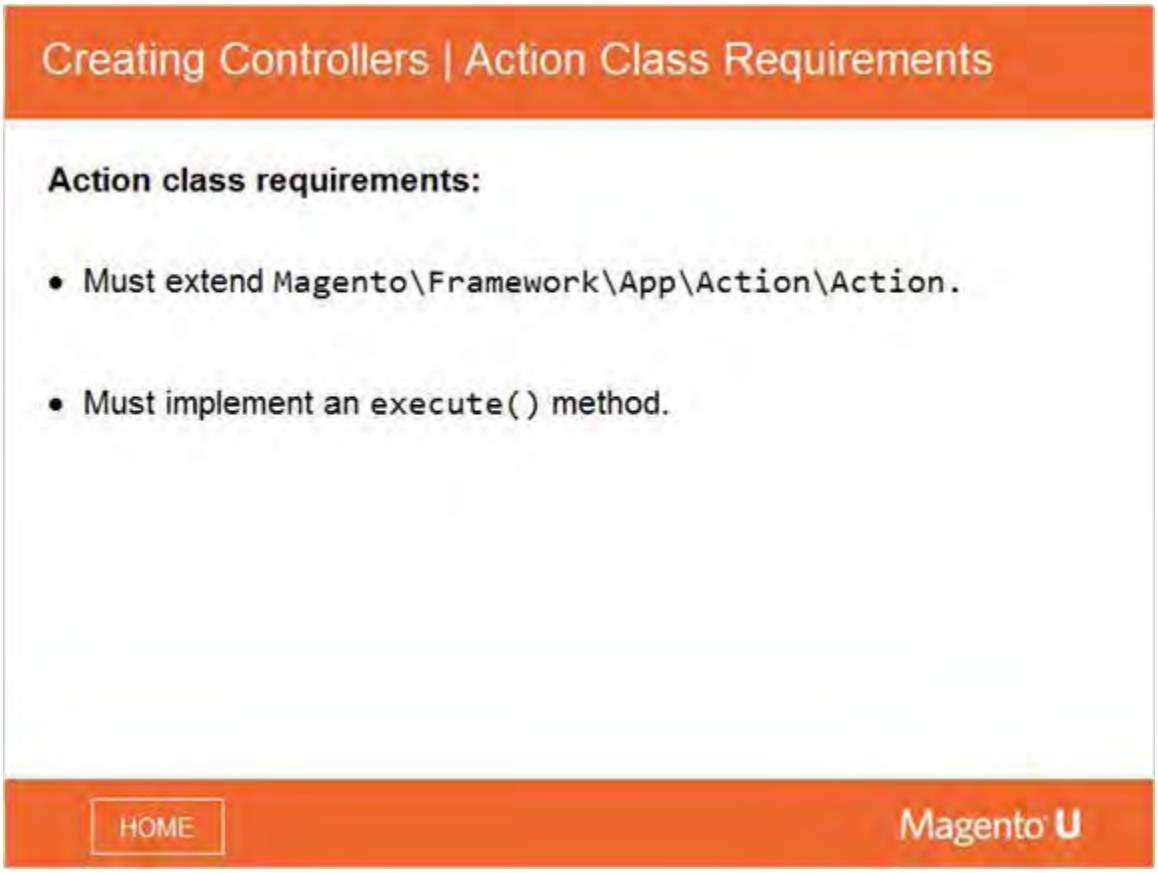
Notes:

Let's take a look at an example...

Catalog is connected to the module `Magento`, product is connected to `Controller/Product` and view is connected to `view.php`.

This results in the class name `Magento\Catalog\Controller\Product\View`.

5.22 Creating Controllers: Action Class Requirements



Notes:

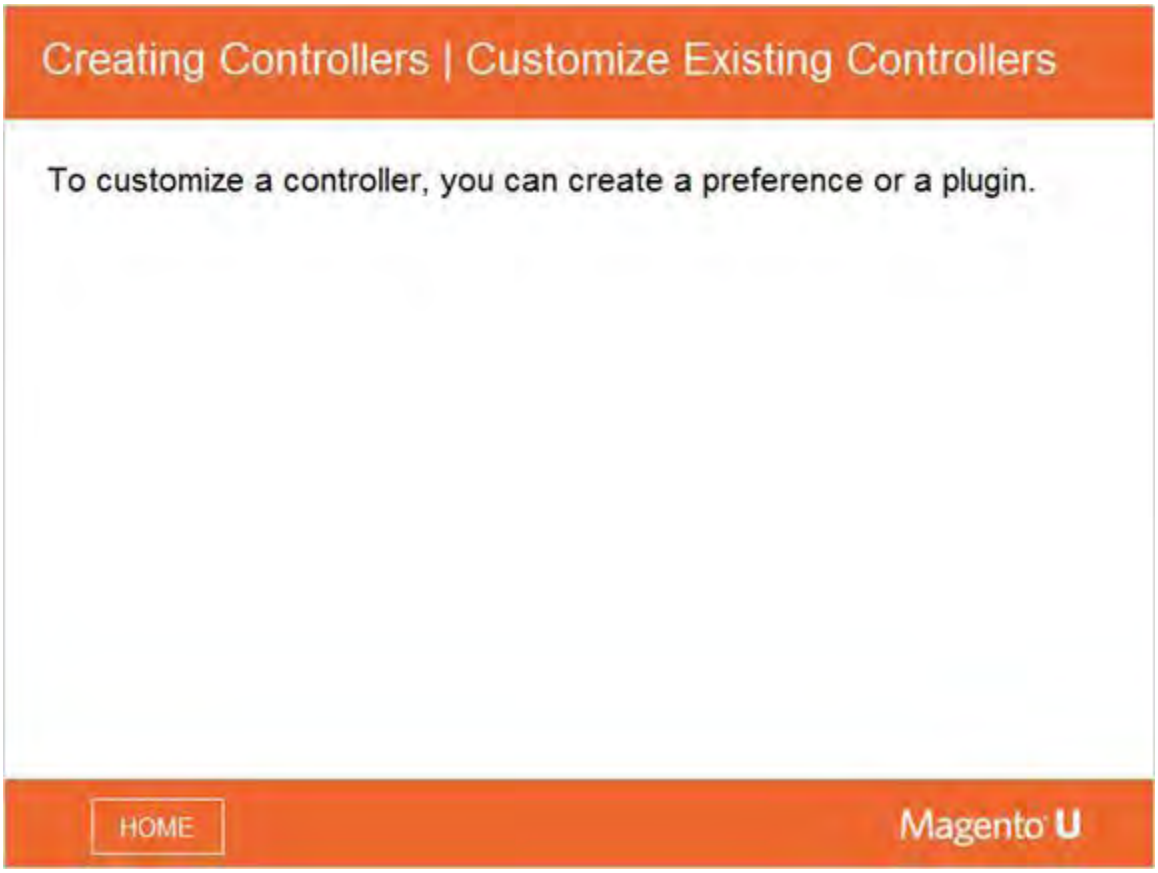
A short reminder that the action class has to extend `Magento\Framework\App\Action\Action` and must implement an `execute()` method.

5.23 Creating Controllers: Testing

Notes:

A simple way to test controllers: enter or hit the URL `frontName/actionPath/action`.

5.24 Creating Controllers: Customize Existing Controllers



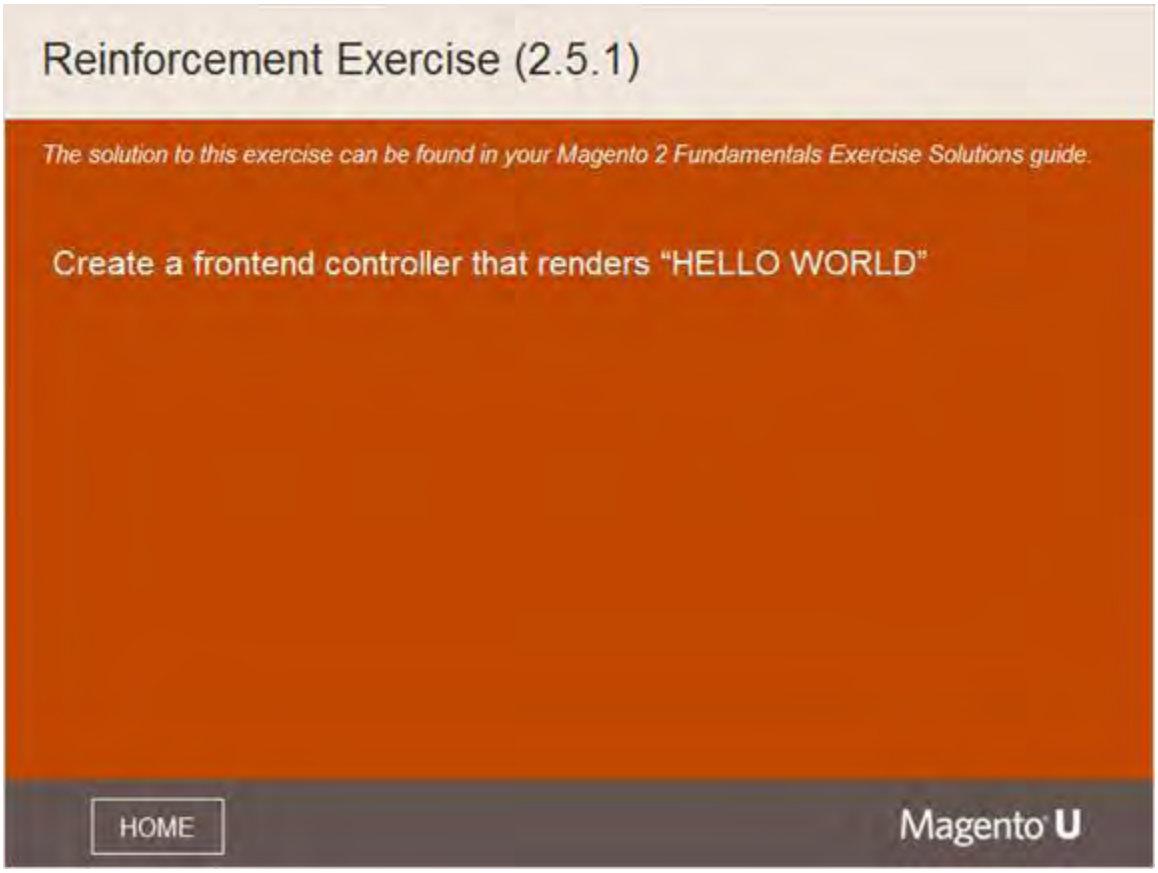
Notes:

To customize a controller, you can either create a preference or a plugin. The key is that the controller is the same class as the model.

In Magento 1, controllers are specific classes, so there was a specific way to customize them, but in Magento 2, controllers are the same as other classes, so they can be customized in the same way as other classes - using preferences or plugins.

This is an advantage in Magento 2.

5.25 Exercise 2.5.1



5.26 Exercise 2.5.2

Reinforcement Exercise (2.5.2)

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

Customize the catalog product view controller using plugins and preferences.

HOME

Magento U

5.27 Exercise 2.5.3

Reinforcement Exercise (2.5.3)

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

Create an adminhtml controller that allows access only if the GET parameter "secret" is set.

HOME

Magento U

5.28 Exercise 2.5.4

Reinforcement Exercise (2.5.4)

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

Make the "HELLO WORLD" controller you just created redirect to a specific category page.

HOME

Magento U

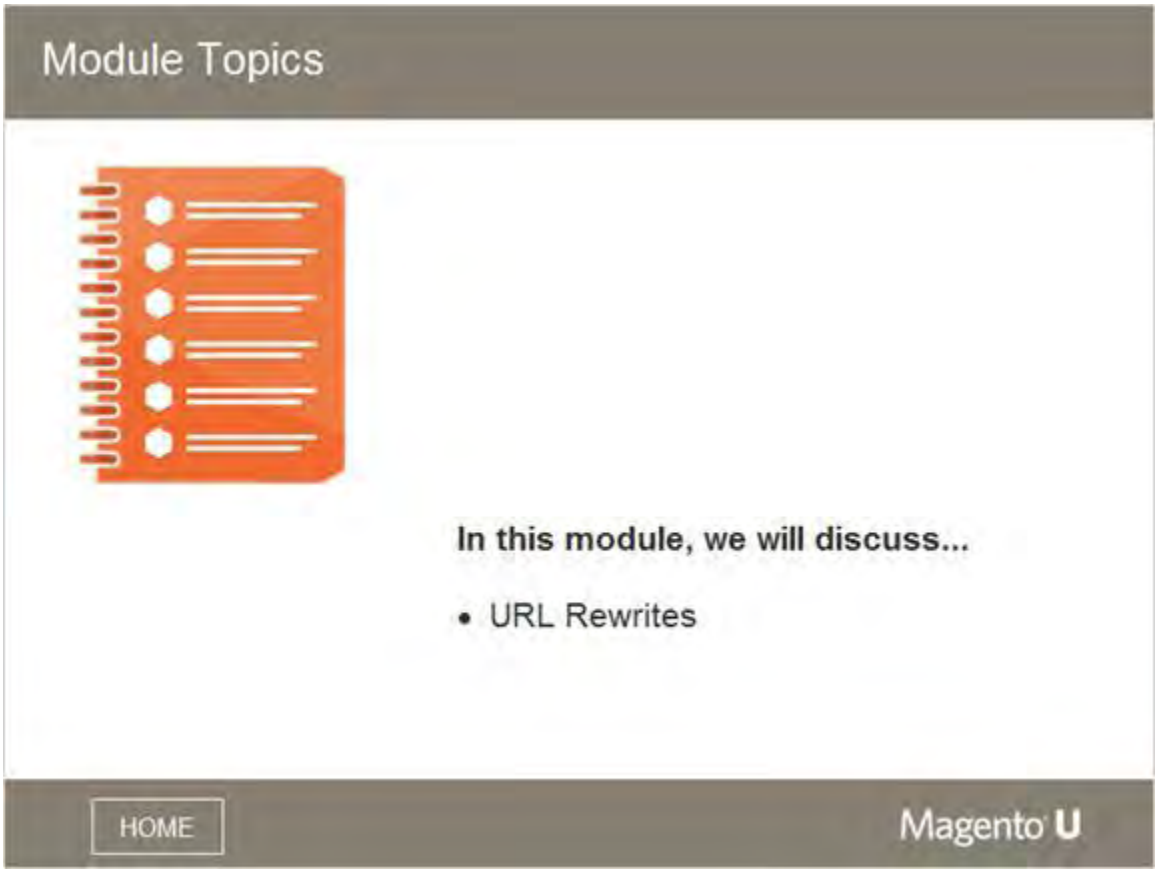
6. URL Rewrites

6.1 URL Rewrites

Notes:

This module will cover the process of URL rewrites.

6.2 Module Topics



Notes:

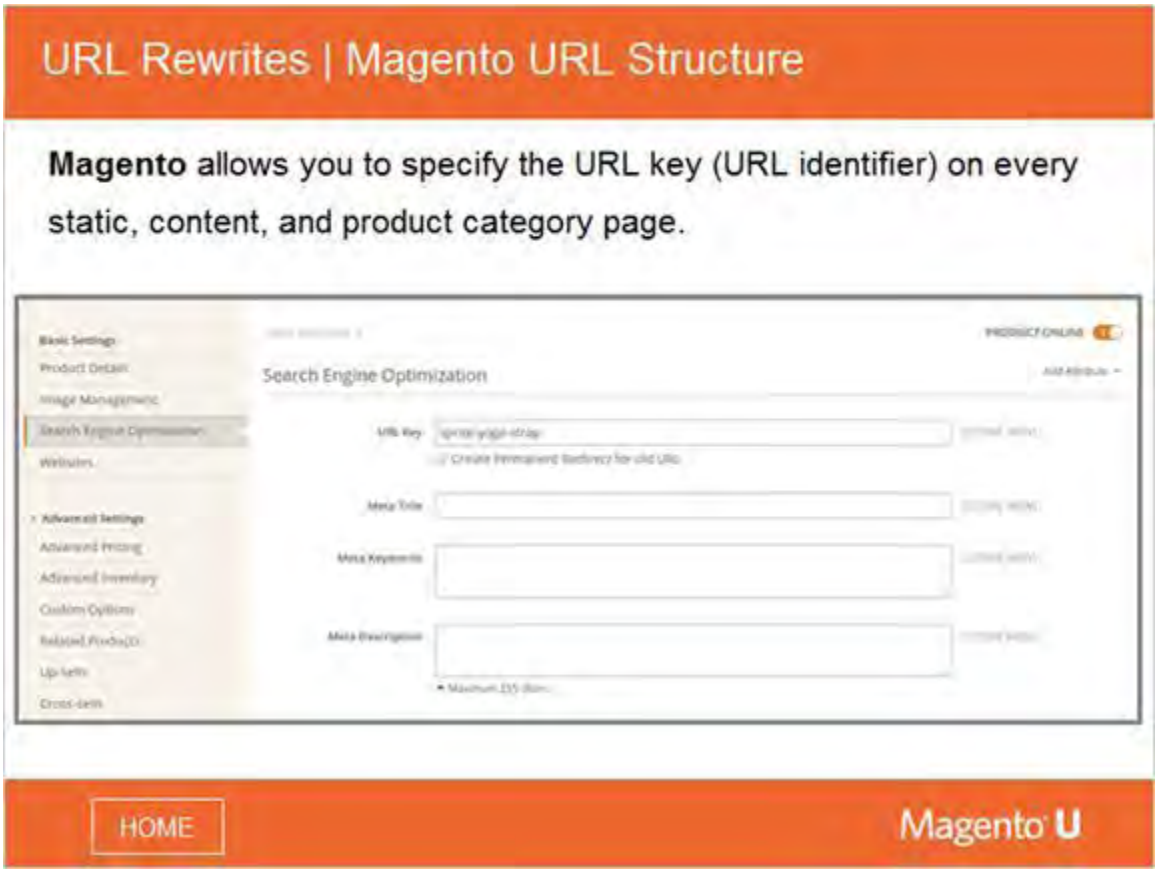
This module focuses solely on URL rewrites.

6.3 URL Rewrites

Notes:

The process of URL rewriting is used to make complicated URL addresses more "user-friendly" by making them shorter, more descriptive, and/or easier to remember.

6.4 URL Rewrites: Magento URL Structure



Notes:

Magento allows you to specify the URL key (URL identifier) on every static, content, and product category page.

For example, you can choose the keyword you want and add it to a particular page's URL, independent of the Magento page name.

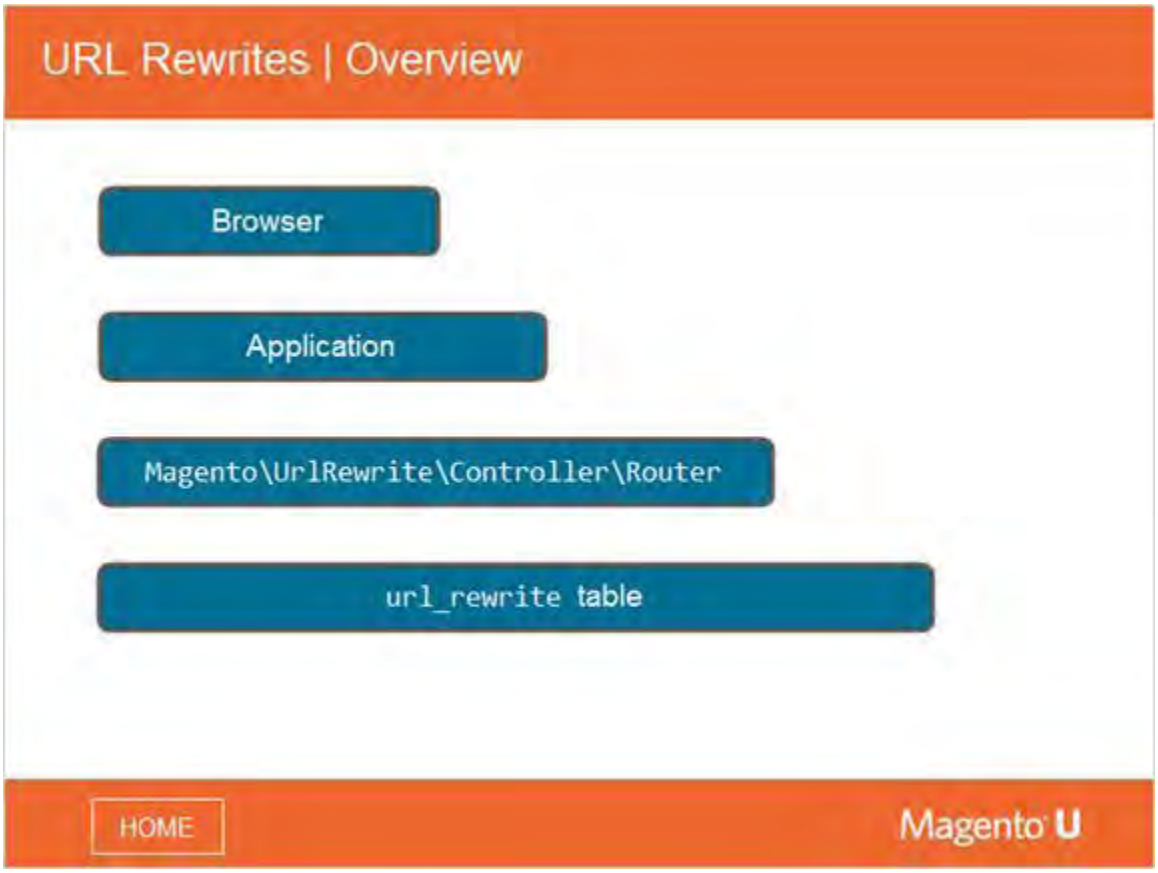
6.5 URL Rewrites: Overview

Notes:

An example of a URL rewrite is displayed on the slide.

While the URL on the left is more easily read by people, Magento will use the second. So, in the browser, the customer would enter the first URL but Magento will work with the second URL.

6.6 URL Rewrites: Overview



Notes:

How do rewrites work?

The browser sends a request to the server or application, which will require the use of `Magento\UrlRewrite\Controller\Router`.

Next, it will access the database - specifically, the `url_rewrite` table. In other words, a customer sends a request using a browser, which then goes to PHP and then to Magento. Within Magento, the request goes to the `UrlRewrite` router.

There are a couple of routers in the matching controller phase, triggered by the `UrlRewrite` router, which will go to the database. It will simply check and substitute one filename path for the other one.

6.7 URL Rewrites: Router Example

URL Rewrites | Router Example

```
public function match(\Magento\Framework\App\RequestInterface $request)
{
    …
    $rewrite = $this->getRewrite($request->getPathInfo(),
        $this->storeManager->getStore()->getId()
    );
    if ($rewrite === null) {
        return null;
    }

    if ($rewrite->getRedirectType()) {
        return $this->processRedirect($request, $rewrite);
    }

    $request->setPathInfo('/' . $rewrite->getTargetPath());
    return $this->actionFactory
       ->create('Magento\Framework\App\Action\Forward', ['request' => $request]);
}
```

HOME

Magento U

Notes:

This code describes how URL rewrites work.

Note that Magento can use the redirect, which results in the customer being redirected to another page. Otherwise, it will simply forward the request.

6.8 URL Rewrites: getRewrite( )

URL Rewrites | getRewrite( )

```
protected function getRewrite($requestPath, $storeId)
    {

/**
  * $this->urlFinder is an
  * instance of Magento\UrlRewrite\Model\Storage\DbStorage class.
 /*
        return $this->urlFinder->findOneByData([
            UrlRewrite::REQUEST_PATH => trim($requestPath, '/'),
            UrlRewrite::STORE_ID => $storeId,
        ]);
    }
```

HOME

Magento U

Notes:

This URL finder is an instance of `Magento\UrlRewrite\Model\Storage\DbStorage` class.

Basically, it goes to the database, retrieves the data, and returns it.

6.9 URL Rewrites: url_rewrite Table

URL Rewrites | url_rewrite Table

Field	Type	Null	Key	Default	Extra
url_rewrite_id	int(10) unsigned	NO	PRI	NULL	auto_increment
entity_type	varchar(32)	NO		NULL	
entity_id	int(10) unsigned	NO		NULL	
request_path	varchar(255)	YES	MUL	NULL	
target_path	varchar(255)	YES	MUL	NULL	
redirect_type	smallint(5) unsigned	NO		0	
store_id	smallint(5) unsigned	NO	MUL	NULL	
description	varchar(255)	YES		NULL	
is_autogenerated	smallint(5) unsigned	NO		0	
metadata	varchar(255)	YES		NULL	

HOME

Magento U

Notes:

Here is an example of the `url_rewrite` table and its data.

6.10 URL Rewrites: url_rewrite Row Example

URL Rewrites | url_rewrite Row Example

url_rewrite_id:	41
entity_type:	product
entity_id:	1
request_path:	joust-duffle-bag.html
target_path:	catalog/product/view/id/1
redirect_type:	0
store_id:	1
description:	NULL
is_autogenerated:	1
metadata:	NULL

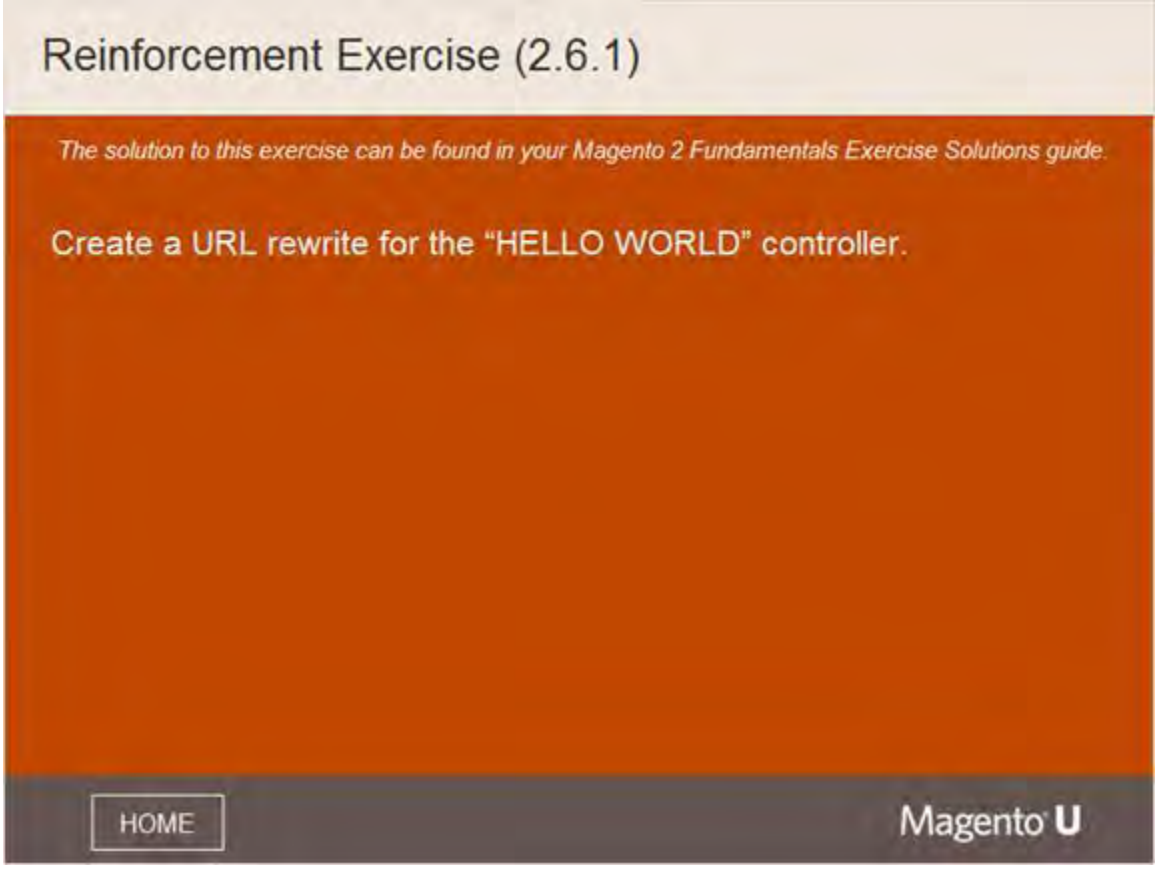
HOME

Magento U

Notes:

This is a display of representative data. Note the request path and the target path – they are substituted in the rewrite, as explained earlier.

6.11 Exercise 2.6.1



6.12 End of Unit Two