Magento® U

Student Guide

Fundamentals of Magento 2 Development
Unit Three

magento.com/training/overview

v.1.0

Magento® U

training@magento.com

magento.com/training/overview

magento.com/training/catalog/certification

Unit Three Contents

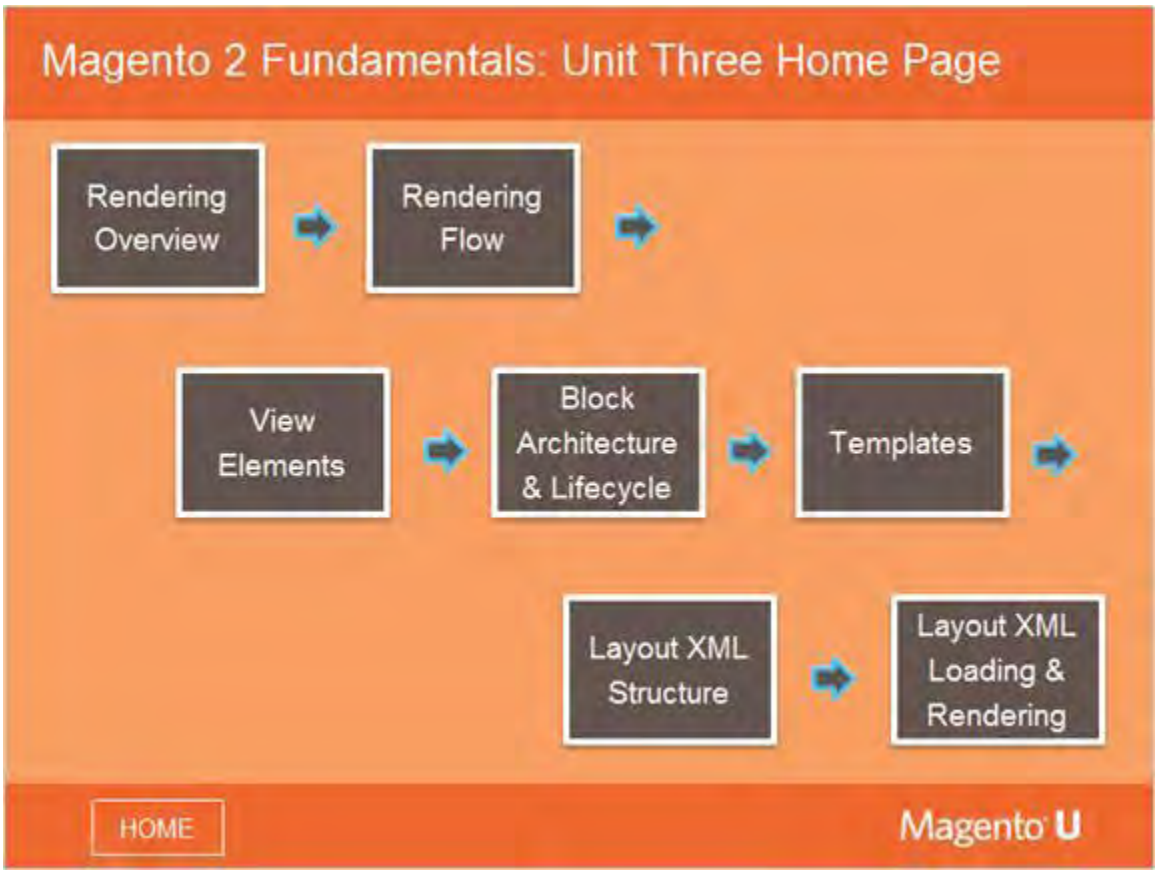
About This Guide

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. Navigating the Course

1.1 Home Page



Notes:

Unit Three of the Magento 2 Fundamentals course contains seven 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. System Overview

2.1 Rendering System

Notes:

In this module, you will be introduced to how templates, blocks, and the design layout / XML schema work in Magento 2.

2.2 Module Topics

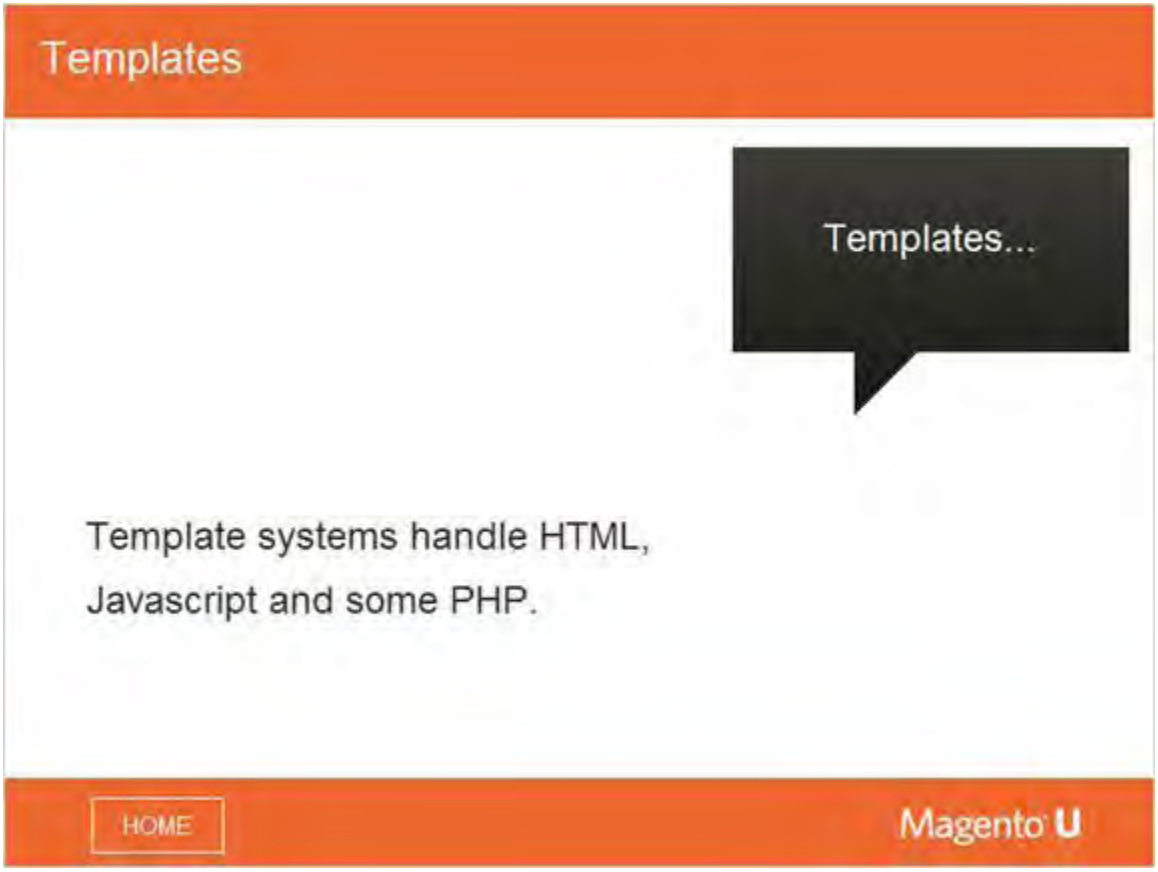
Notes:

In this module, we present an overview of the rendering system of Magento 2, focusing on templates, blocks, and layout xml.

Each of these three aspects will be taught first in isolation, to make important concepts easier to understand.

Then, the course will address how all the pieces fit together, using best practices.

2.3 Templates

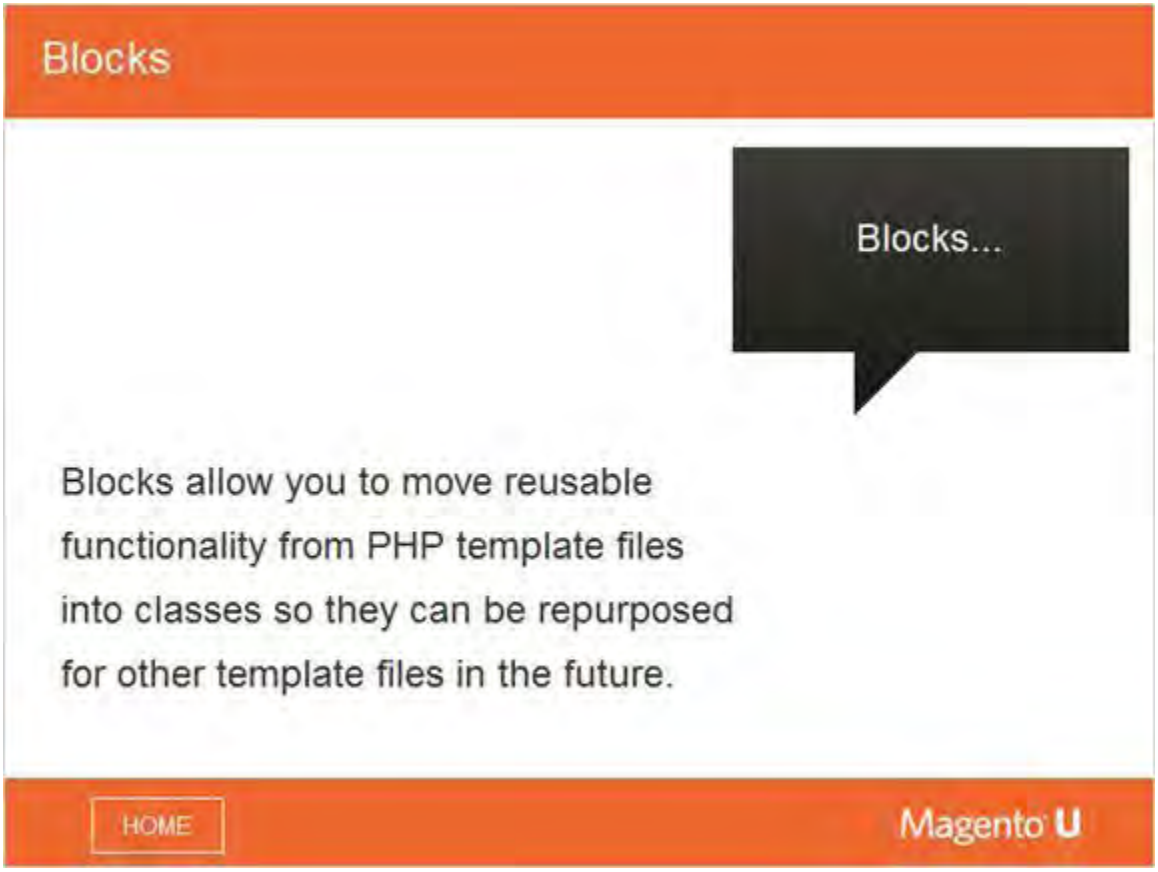


Notes:

Templates in Magento are phtml files that contain html instructions.

In Magento 1, phtml is the only option; in Magento 2, it is possible to use any rendering system.

2.4 Blocks

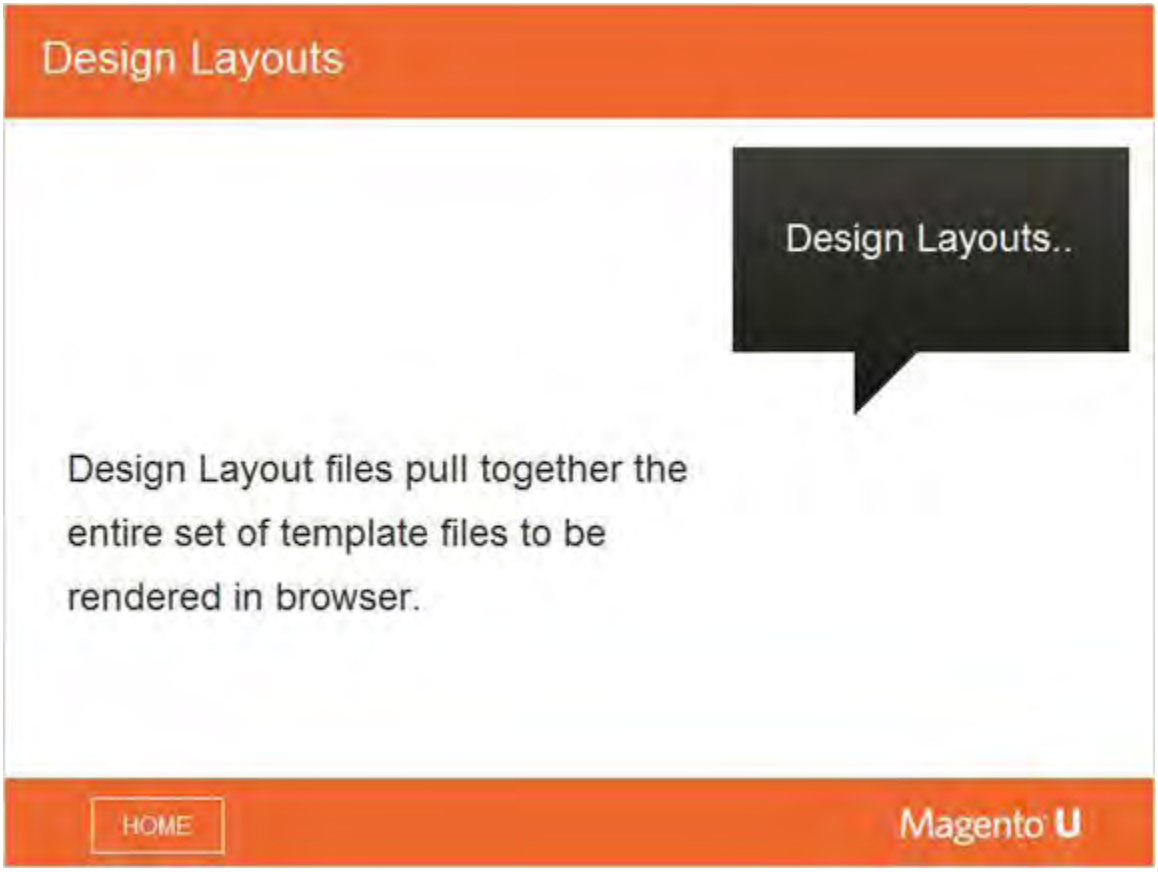


Notes:

The concept of blocks is a unique aspect of Magento. Blocks allow you to move reusable functionality from PHP template files into classes so they can be re-purposed for other template files in the future. There are several block types in Magento 2.

A block is a class that calls a template and provides data to that template. Blocks often instantiate models, which then can query databases, and so on.

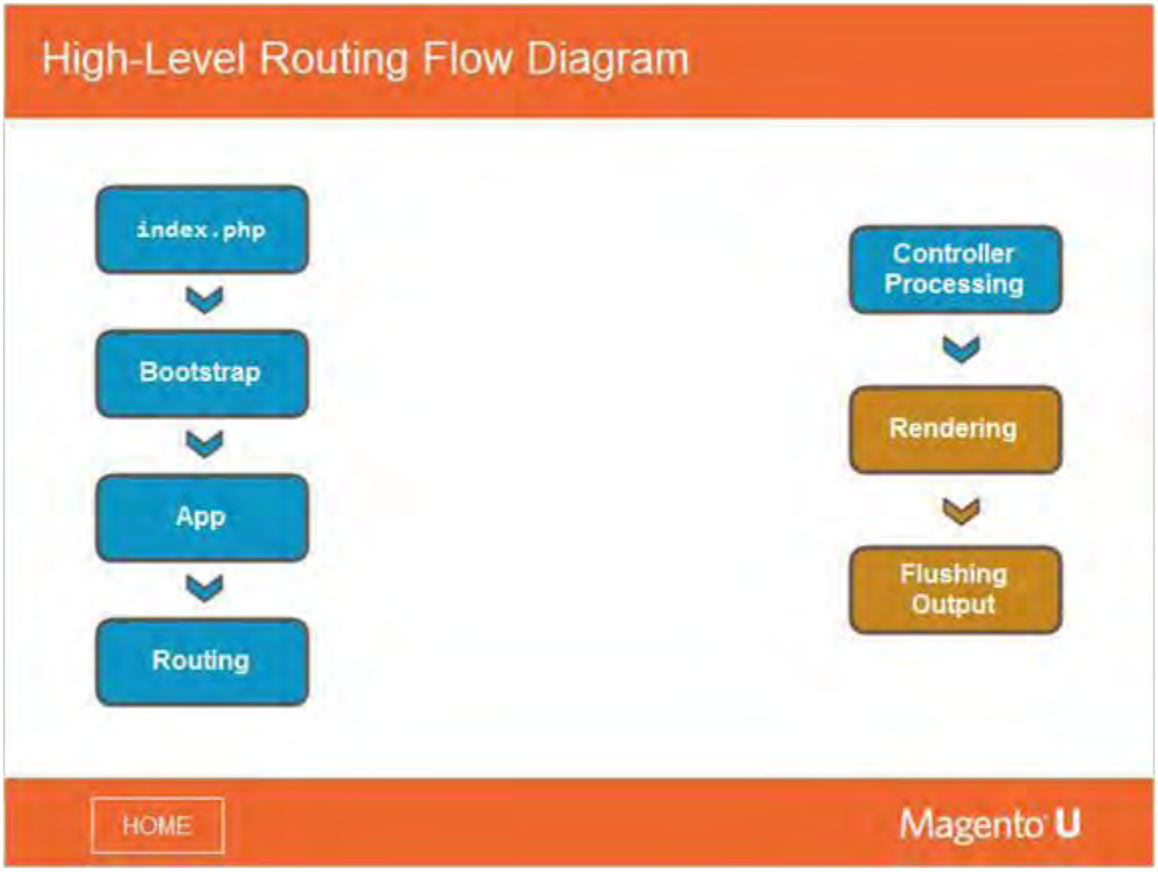
2.5 Design Layouts



Notes:

Layout XML is an xml file where you define the structure of a page using xml instructions. While layout itself has changed in Magento 2, the overall concepts behind layout remain the same.

2.6 High-Level Routing Flow Diagram



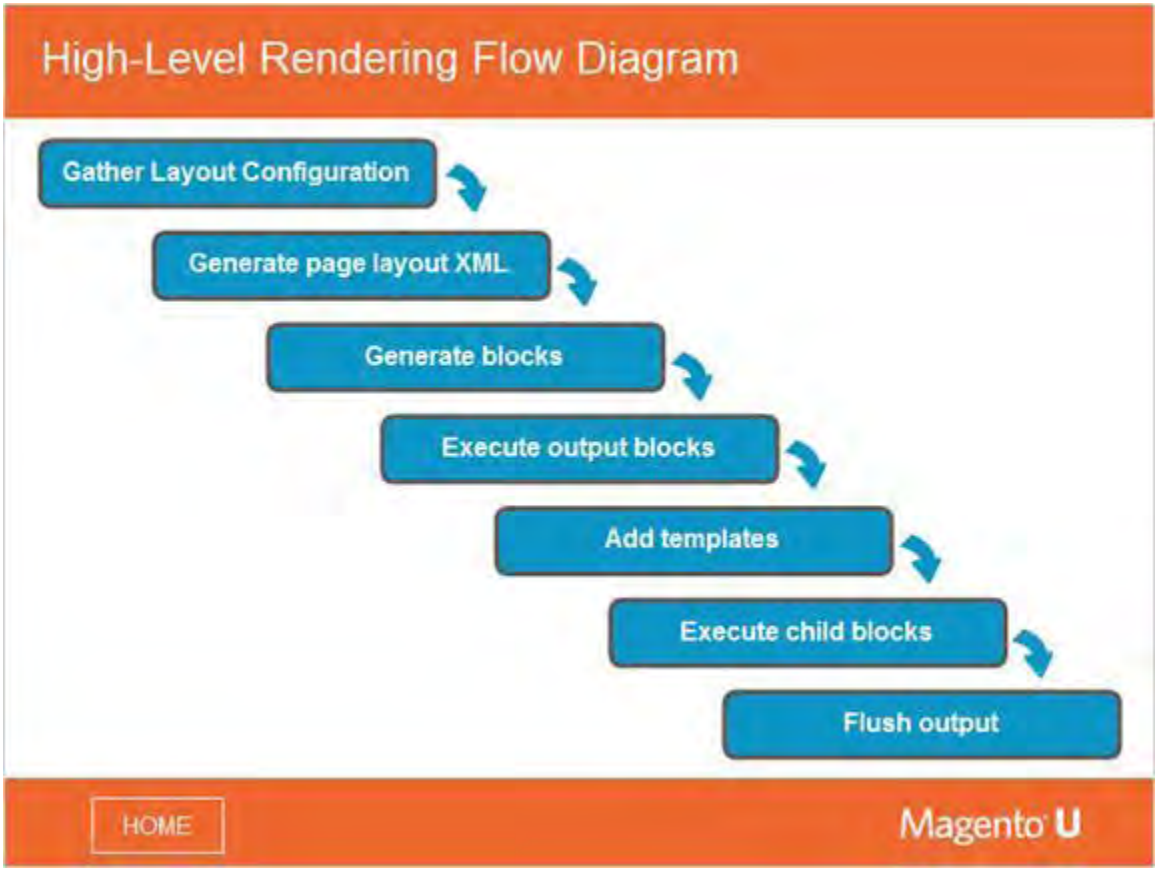
Notes:

Throughout the course, you have seen this diagram depicting the overall routing flow within an application.

In this unit, we will be focusing on the last two aspects - rendering and flushing output.

It is important to understand these steps to be able to debug each of them.

2.7 High-Level Rendering Flow Diagram



Notes:

This graphic shows the general process flow for rendering pages within a web site.

As mentioned earlier, the general concepts behind rendering are very similar in Magento 1 and 2.

In Magento 2, we have two rendering systems that are important. One of them provides the rendering layout, and operates similarly to Magento 1.

The second one is based on page objects. There are two rendering layouts: `View::loadLayout()` and `View::renderLayout()`.

3. Rendering Flow

3.1 Rendering Flow

Notes:

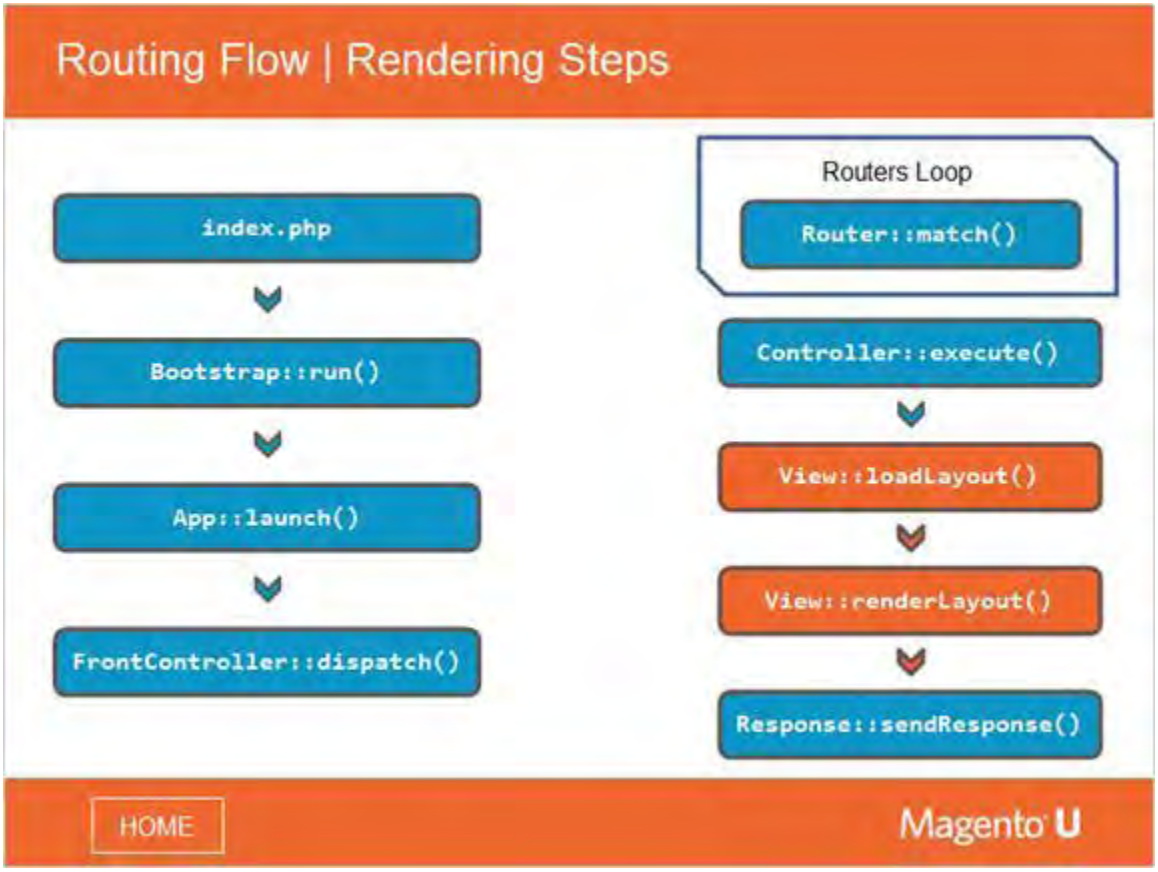
This module focuses on the rendering flow process.

3.2 Module Topics

Notes:

In this module, we discuss three aspects of the rendering flow process: the view, result object, and pages.

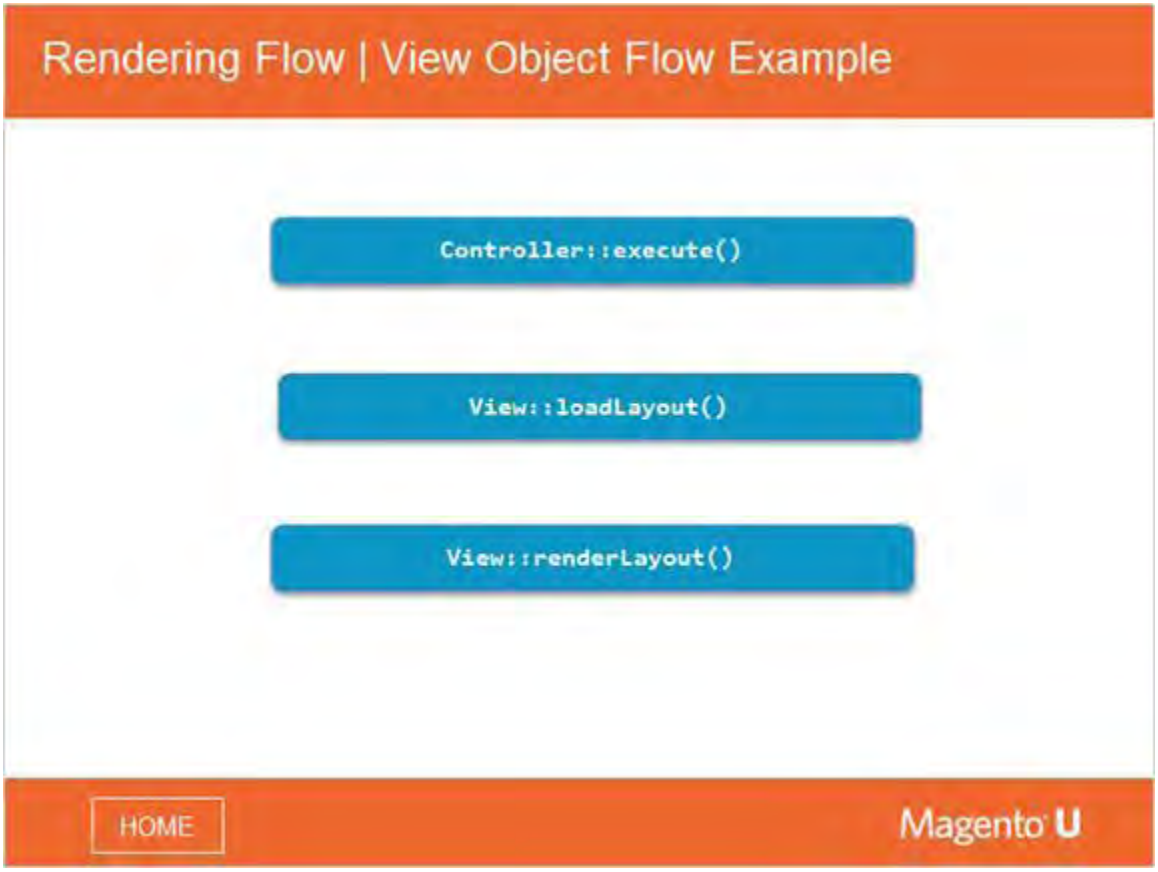
3.3 Rendering Steps



Notes:

This updated version of the routing flow diagram now shows you the methods that correspond to key steps in the rendering process.

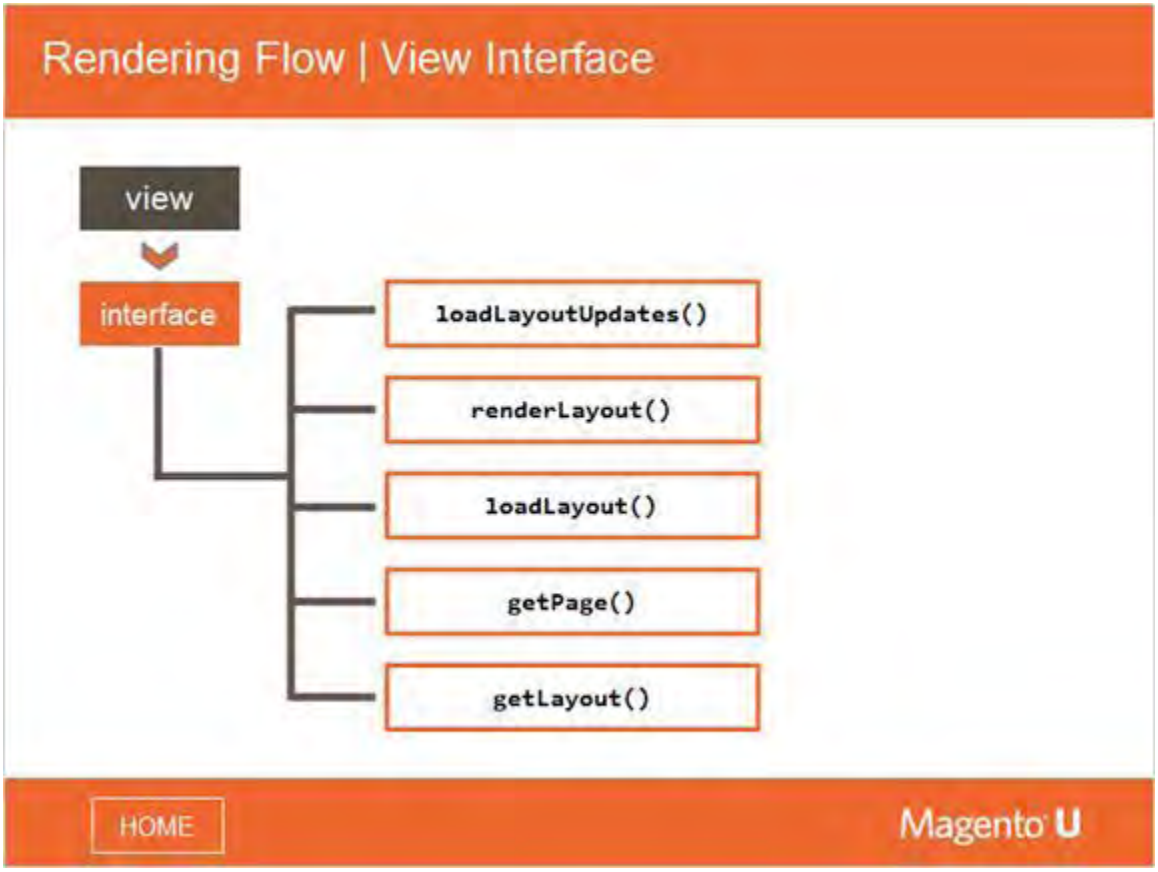
3.4 View Object Flow



Notes:

These sample methods are from the Checkout Module.

3.5 View Interface



Notes:

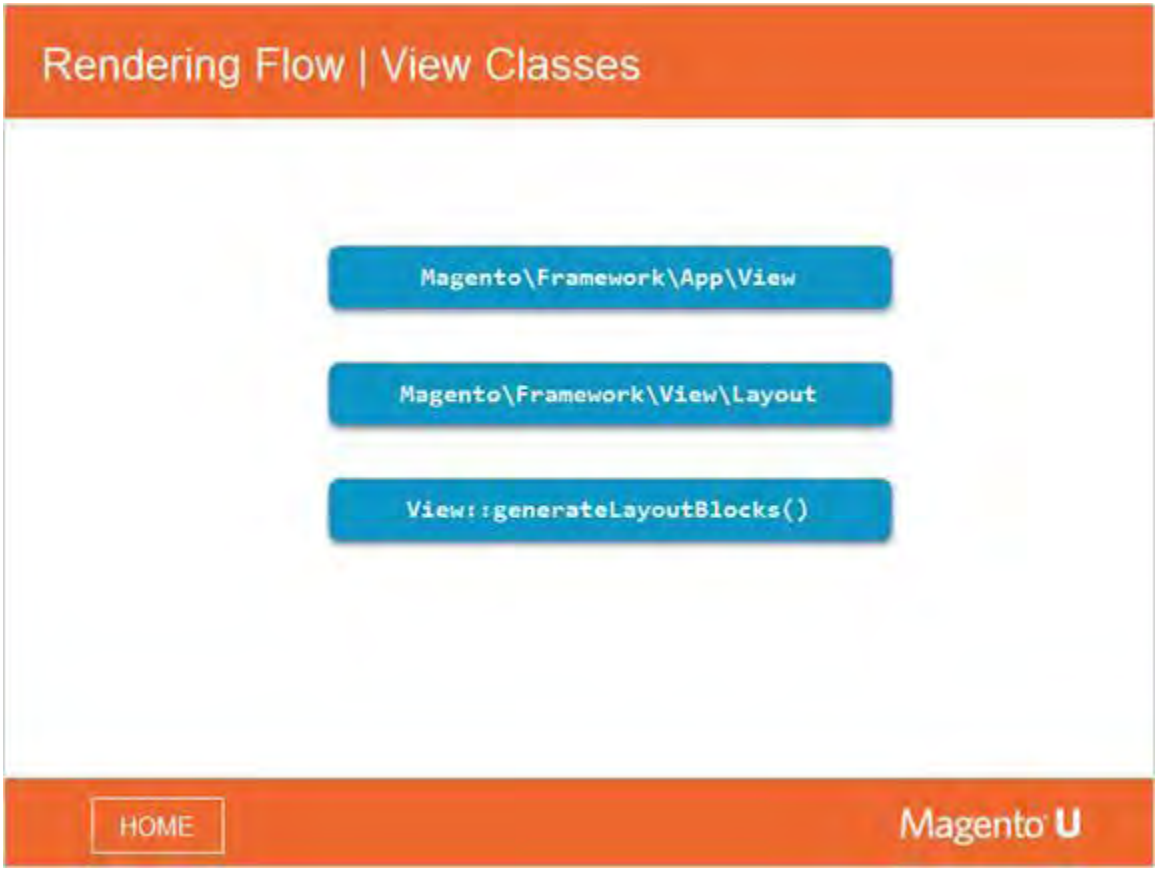
This diagram depicts the view interface and associated methods.

If you want to see a code example, you can look at `Magento/Framework/App/View.php` within your Magento 2 installation. This implements the view interface.

In Magento 1, action classes contain the `loadLayout()` and `renderLayout()` methods.

Magento 2 doesn't have this logic in its action classes but we still need to accomplish the same tasks. To do this, we have the view object. Its interface is almost the same as in Magento 1.

3.6 View Classes

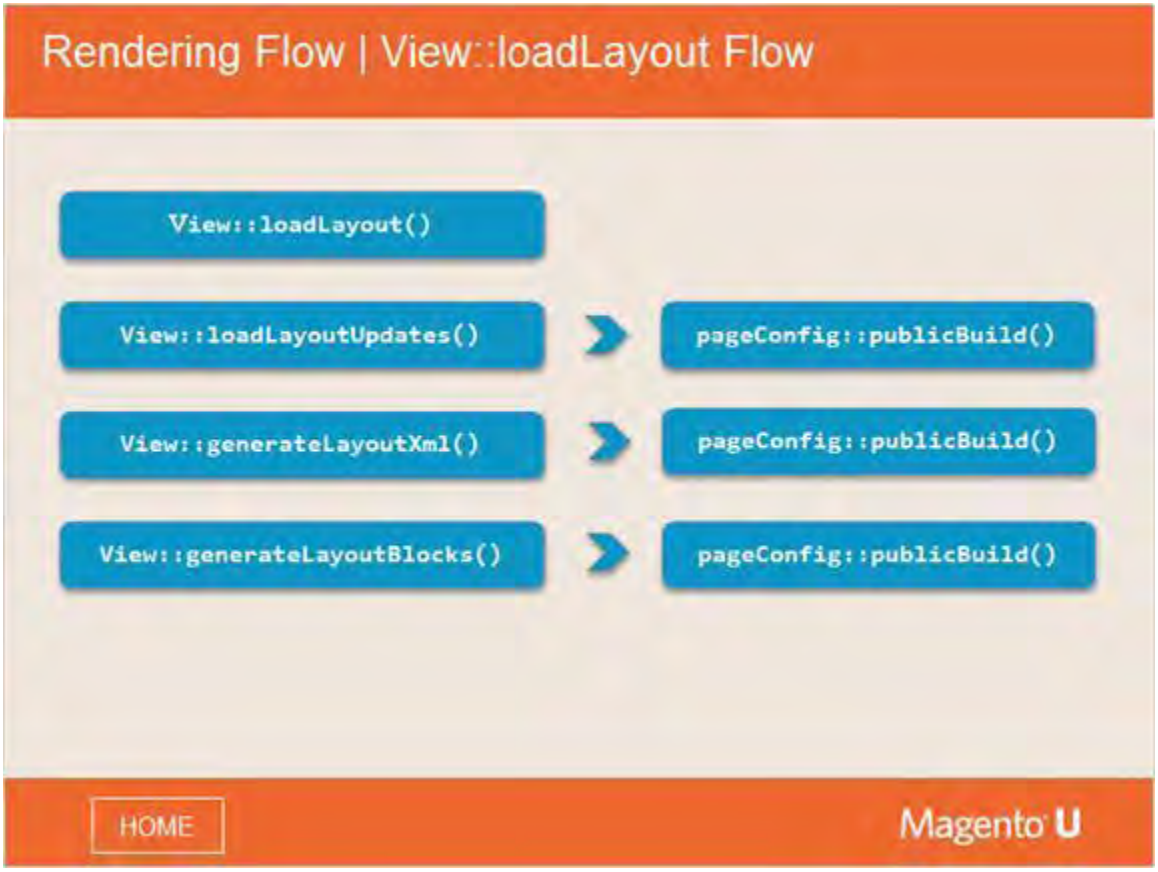


Notes:

There are two key classes shown on this slide - View and Layout classes. `\Magento\Framework\View\Layout` is the most important of the set of layout classes. It is an analogue of the `Mage_Core_Model_Layout` class in Magento 1. This class, together with the Merge class (an analogue of `Mage_Core_Model_Layout_Update`) implements the whole logic of loading, parsing and filtering layout xml files, generating layout xml for the current page, generating blocks' instances, and setting the right parameters into them.

The view class contains some functions that the `Mage_Core_Controller_Varien_Action` class in Magento 1 has, which are mostly `loadLayout()/renderLayout()` methods. However, this object provides access to the layout object. It is worth mentioning here that this mechanism, based on View, is somewhat deprecated. Magento 2 has another mechanism, based on the $page object, which should be implemented everywhere in the core, with the view class disappearing in a future release.

3.7 View::loadLayout Flow



Notes:

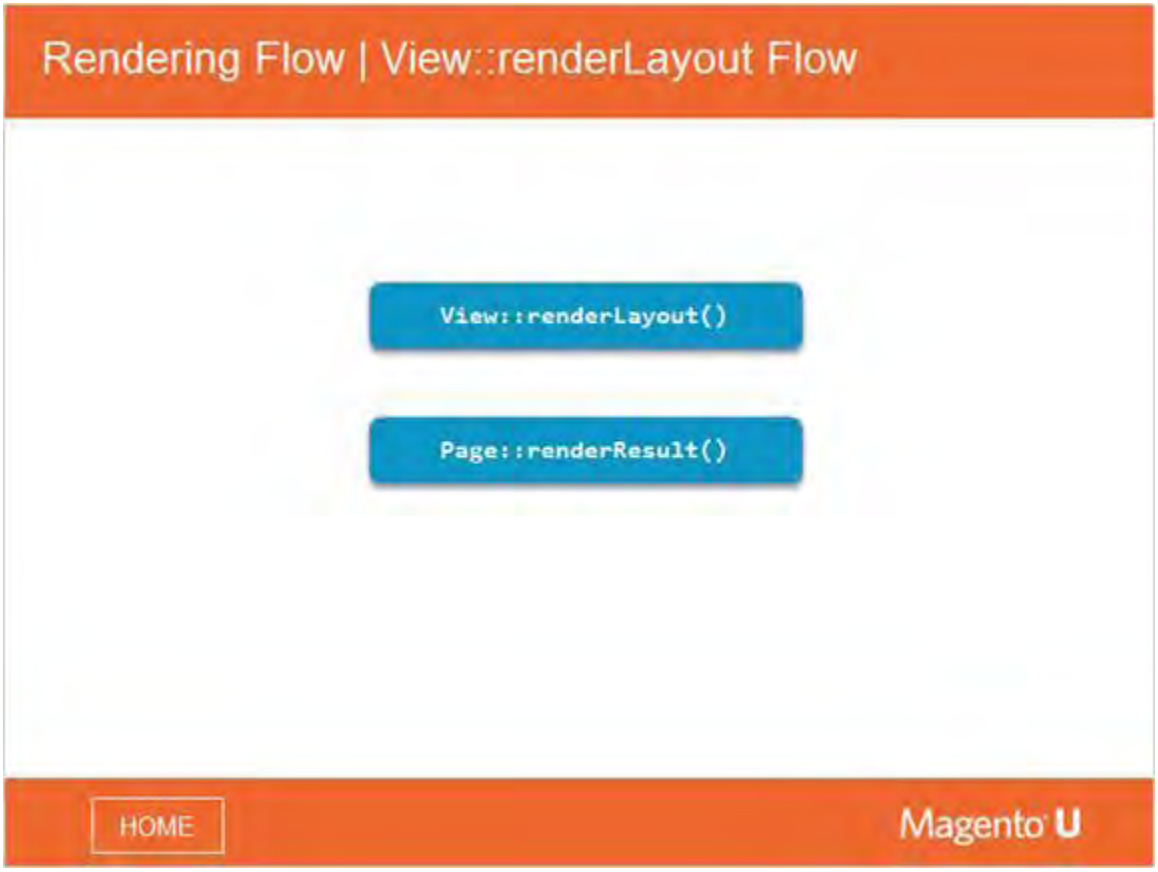
In Magento 1, there are two phases of rendering: `loadLayout()` and `renderLayout()`.

In the first phase, Magento 1 goes through all the xml files and loads them after it instantiates all the blocks.

Magento then creates a block structure, and prepares the layout for every block. In the second phase, the `toHtml()` method of the root block is called and blocks will start rendering.

Reference: Locate the View.php file in your Magento installation, and then look for View::loadLayoutUpdates(). This is the system that generates layout xml and layout blocks.

3.8 View::renderLayout Flow



Notes:

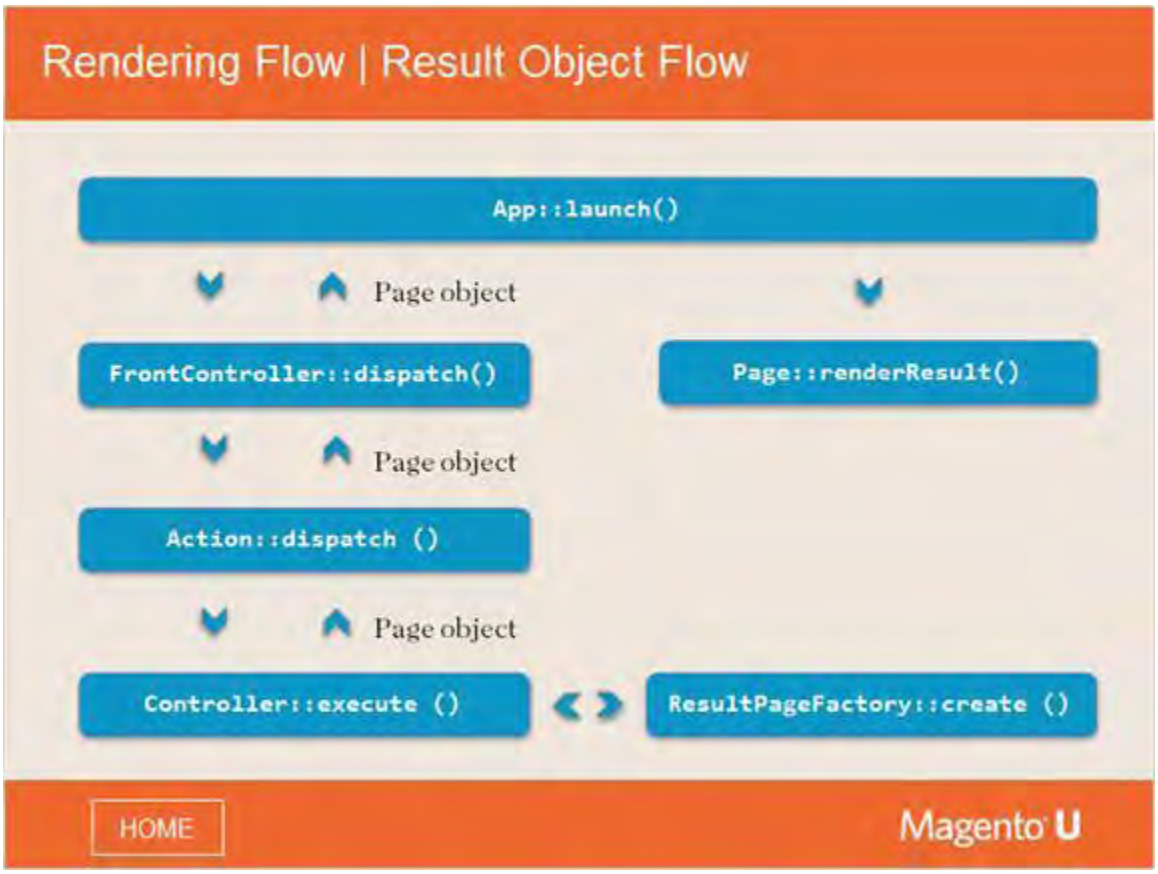
When rendering layout, a `renderLayout()` call is used, just as in Magento 1.

A quick reminder of how it works in Magento 1: there are certain “output” elements in the layout xml (usually a root block) that is rendered by the Layout object. All other blocks are rendered recursively from the root block.

In Magento 2, the rendering system is a bit more complex, but follows the same concept in general.

We will go into further detail later in the course.

3.9 Result Object Flow



Notes:

Here is high level depiction of the result object flow.

It starts with the App:: launch() and goes to `FrontController::dispatch()`, then to the action.

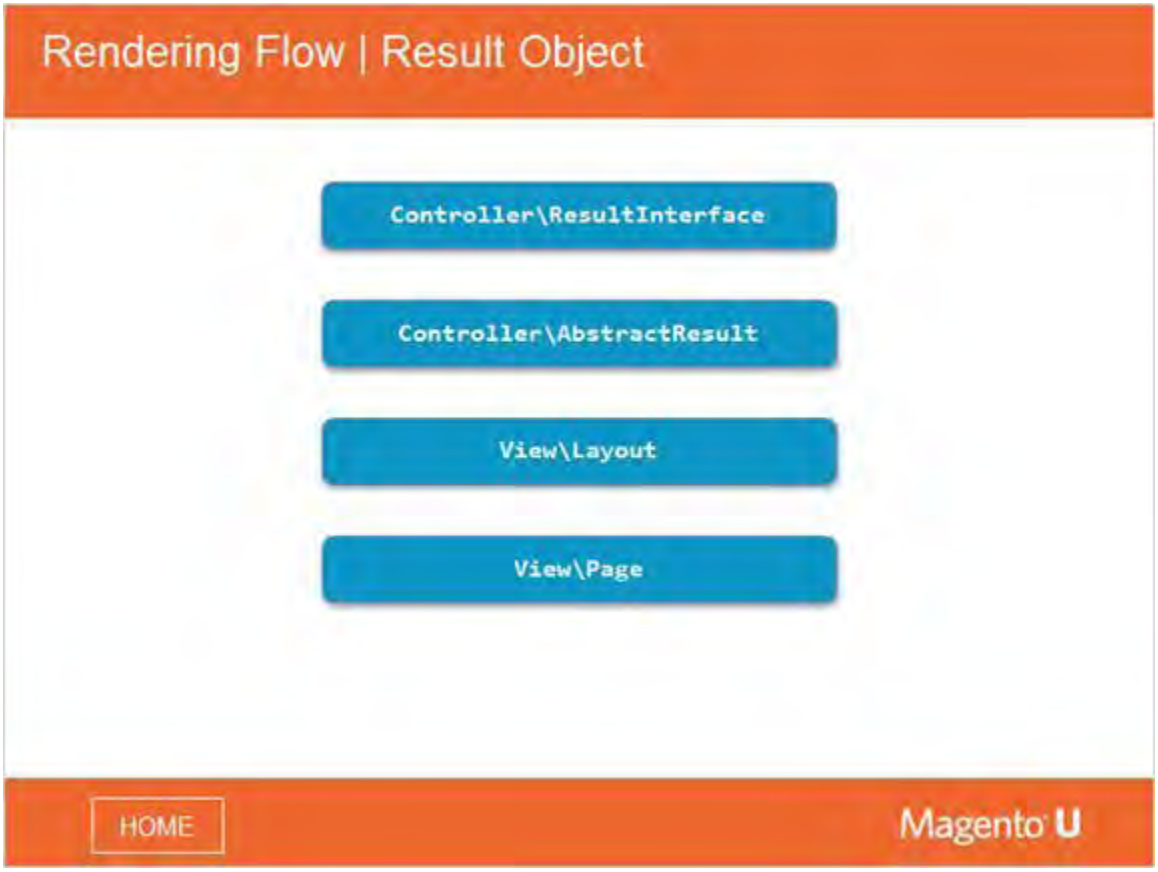
The controller then executes until `ResultFactory::create()` is created and returns a page object.

Afterwards, the application goes back to the object `Page::renderResult()`.

This is the new rendering flow. In Magento 1, the controller loads the layout whereas in the Magento 2, the controller only creates page objects and sends them back to `App::launch()`.

Note that only one step creates and renders the result page.

3.10 Result Object



Notes:

The result object has three methods - the most important one is `renderResult()`.

The implementation of that interface can be found in:
`Magento\Framework\Result\Page`, *which*

- extends `Magento\Framework\Result\Layout.`
- extends Magento`\Framework\Controller\AbstractResult.`
- implements `ResultInterface`.

So, in total, there are three methods and four classes within this rendering system (based on the result object).

Reference: Locate the result object in the Magento installation:
`<magento_root_dir>/lib/internal/Magento/Framework/Controller/ResultInterface.php`

3.11 ResultInterface

Rendering Flow | ResultInterface

```
interface ResultInterface
{
    /**
     * @param int $httpCode
     * @return $this
     */  public function setHttpResponseCode($httpCode);
    /**
     * Set a header

     * If $replace is true, replaces any headers already defined with that
     * $name.
     * @param string $name
     * @param string $value
     * @param boolean $replace
     * @return $this
     */
    public function setHeader($name, $value, $replace = false);
    /**
     * Render result and set to response
     *
     * @param ResponseInterface $response
     * @return $this
     */
    public function renderResult(ResponseInterface $response);
}
```

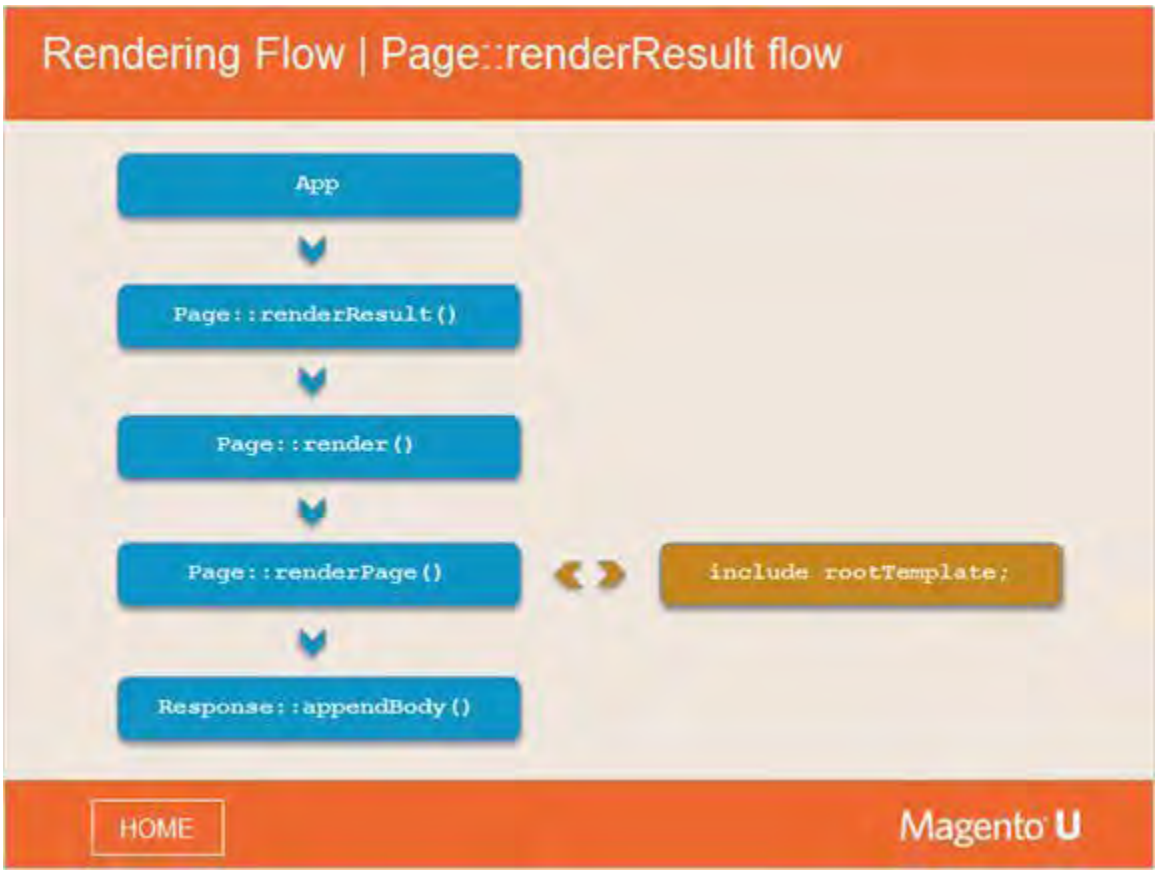
HOME

Magento U

Notes:

The most important part of this piece of code is the `renderResult()` method, which the result object has to implement to effect the rendering.

3.12 Page::renderResult flow

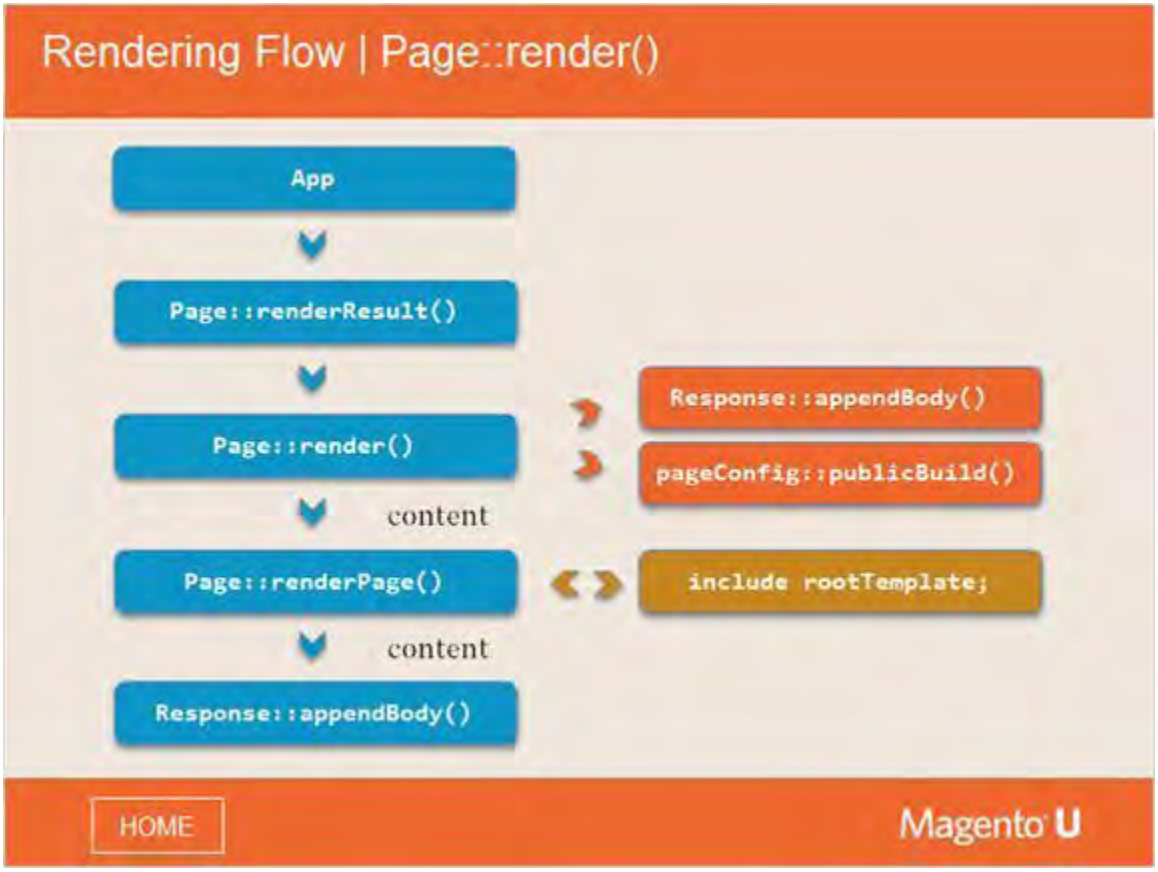


Notes:

This diagram shows how Page::`renderResult()` works. It takes a number of steps, starting with App and then going through the flow depicted on the slide. Note that Page::`renderPage()` includes the `rootTemplate`.

Reference: look at the renderPage method in your Magento installation and how the template name is obtained.

3.13 Page::render()



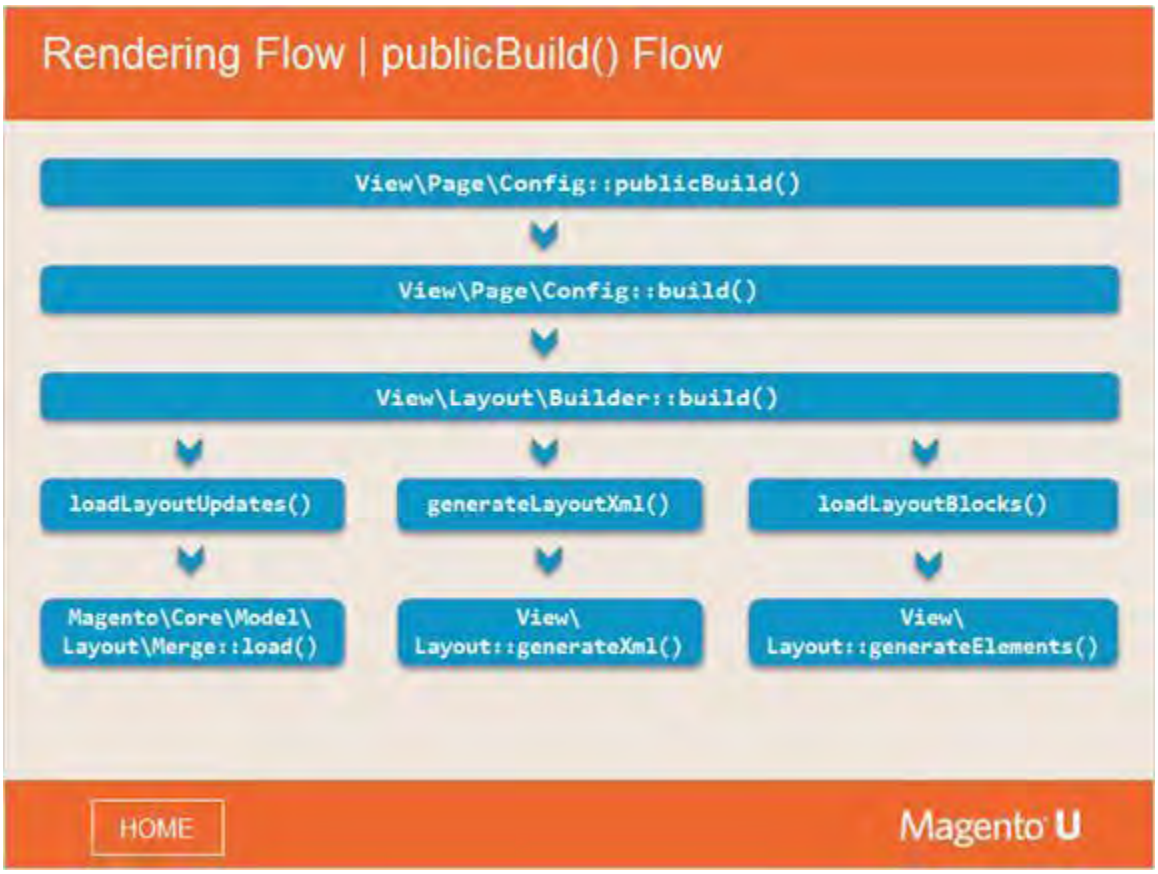
Notes:

In this updated flow diagram, you see that two methods have been added. They are called in by `Page::render()`.

First, a call to `Response::appendBody()` will add generated HTML to the response object. The second method should be familiar, as we've seen it already when going through the view object- based rendering system. It is the same call as `publicBuild()`.

We will see what happens there on the next slide.

3.14 publicBuild() Flow



Notes:

The new Magento 2 page system has two steps: build and render.

Magento 1 uses the interface `loadLayout()`, `generateLayout()` and then `generateBlocks()`.

The new Magento 2 system implements the same steps and code that we had in Magento 1, with the exception of:
`$this->layout->getUpdate()->load();` *inside of* `View\Layout\Builder::build()`

We have three actions: `loadLayout()`, `generateLayout()`, and `loadLayoutBlocks()`.

Remember Magento 2 has the same code as Magento 1, except with layers on top. The concept of the page builder is that you build a page, step-by-step.

Reference: To see how the build system works in the code, do the following:

- Go to `Magento/Framework/View/Page/Config.php` in your Magento installation. Note that it does not extend anything - it is just configuration, and does not parse any files. Public build just calls build.
- Next, look at the builder, which extends `View/Layout/Builder.php` and implements the builder interface. This code is very similar to that of Magento 1.

3.15 Code Demonstration | publicBuild() Flow

Notes:

3.16 Exercise 3.3.1

Reinforcement Exercise (3.3.1)

The solution for this exercise can be found in the Magento 2 Fundamentals Exercise Solutions guide.

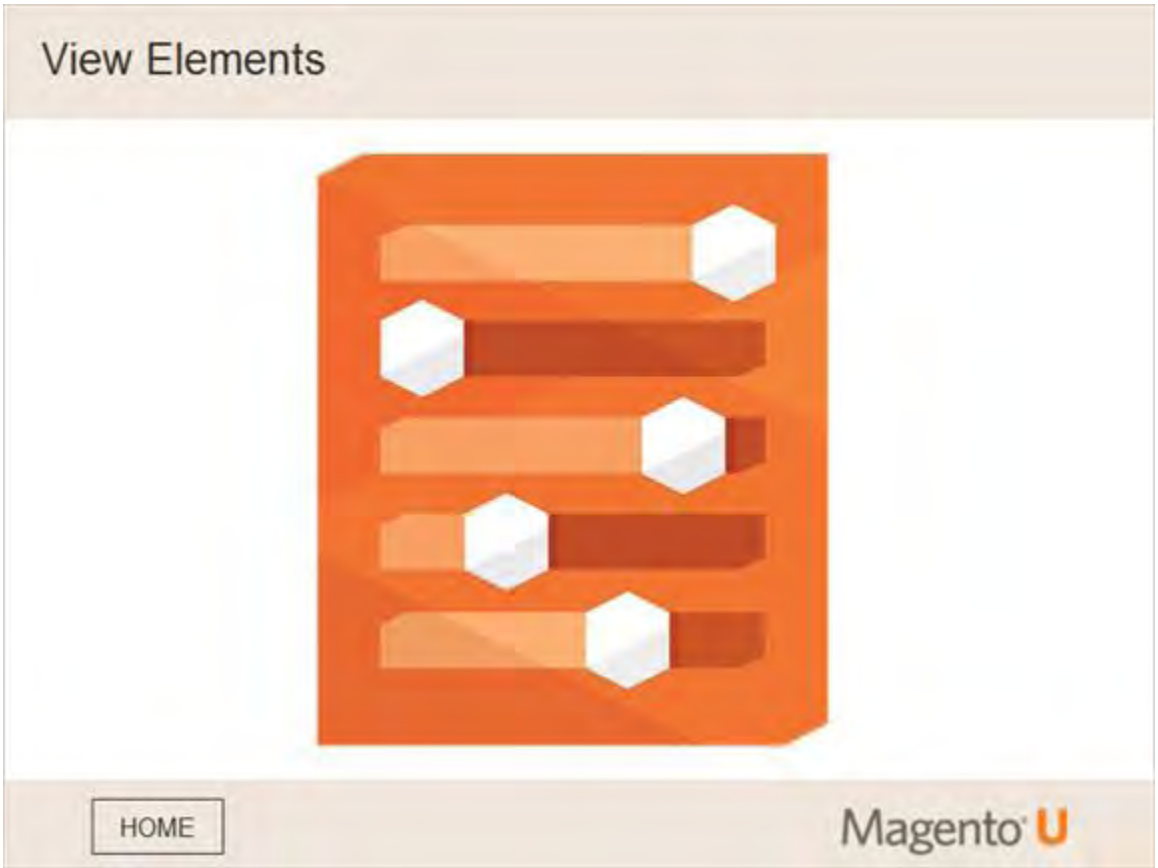
Find the correct location in the core files and print out `layout xml` for the product view.

HOME

Magento U

4. View Elements

4.1 View Elements



Notes:

In this module, we will discuss the primary view elements in Magento 2.

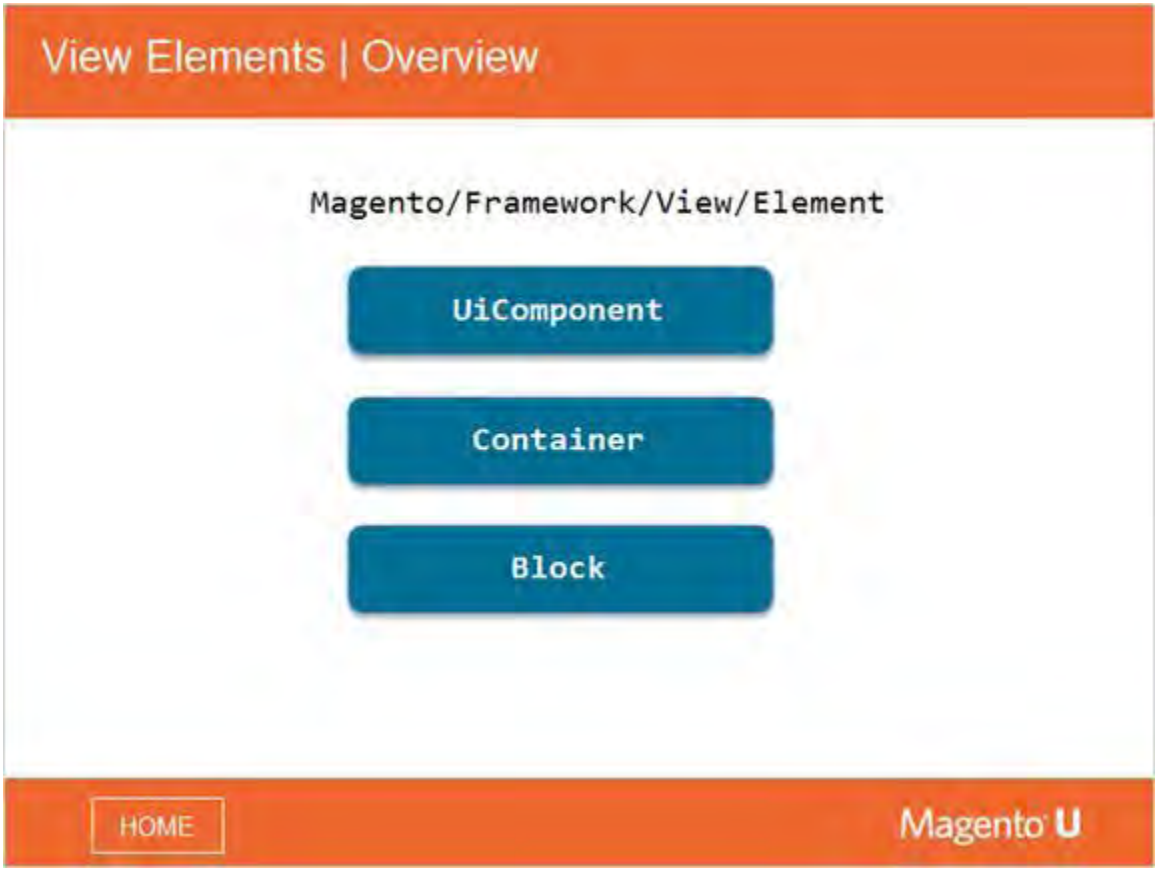
4.2 Module Topics

Notes:

In Unit One of this course, you were introduced to containers and blocks.

In this module, we will go into more detail about both of these elements, as well as introduce UiComponent.

4.3 View Elements | Overview

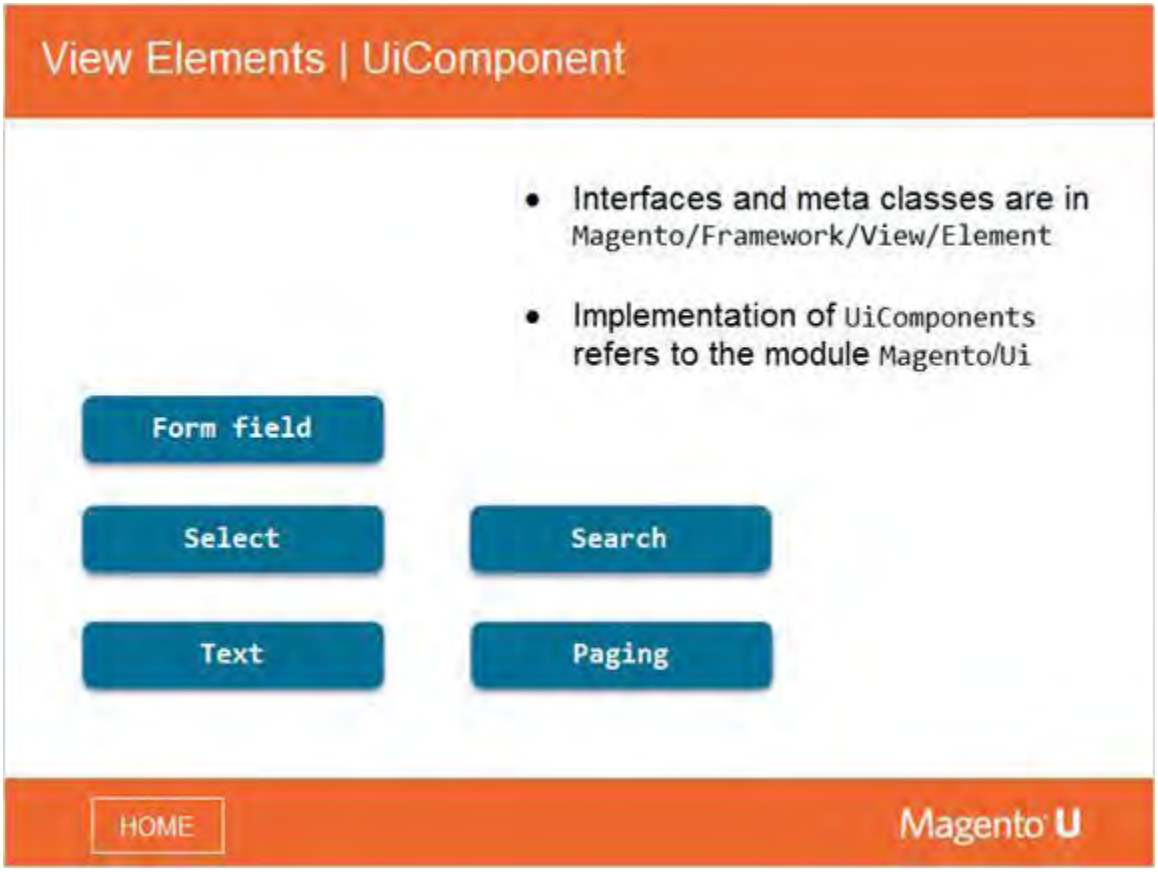


Notes:

There are three types of view elements in Magento 2 - UiComponents, Containers, and Blocks.

In Magento 1, we have only blocks and form elements, under `lib/Varien/Form`.

4.4 View Elements | UiComponent



Notes:

The implementation of the UI components refers to the module `Magento/Ui`. We have a full set of UI components that are located in `Magento/Framework/View/Element/Ui`. In there we have: `UiComponentInterface.php`, `UiComponentFactory.php`, and `UiElementFactory.php`

The component factory contains the builder and the `Config.php` files. The components themselves are in `Magento/Ui/Component/`.

4.5 View elements | UiComponent Interface

View elements | UiComponent Interface

```
interface UiComponentInterface extends BlockInterface
{
      public function update(array $arguments = []);

      function prepare();

      public function render(array $data = []);

      public function renderLabel();

      public function getContentTemplate();

      public function getName();

      public function getParentName();

      public function getRenderContext();
      public function getElements();

      public function setElements(array $elements);
      public function getConfigBuilder();
}
```

HOME

Magento U

Notes:

The `UiComponent` interface extends `BlockInterface`, which is logical as block is an element of the interface.

In fact, the `UiComponent` is very much like an advanced block.

4.6 View Elements | Render UiComponent

View Elements | Render UiComponent

```
protected function _renderUiComponent($name)
{
    $uiComponent = $this->getUiComponent($name);
    return $uiComponent ? $uiComponent->toHtml() : '';
}
```

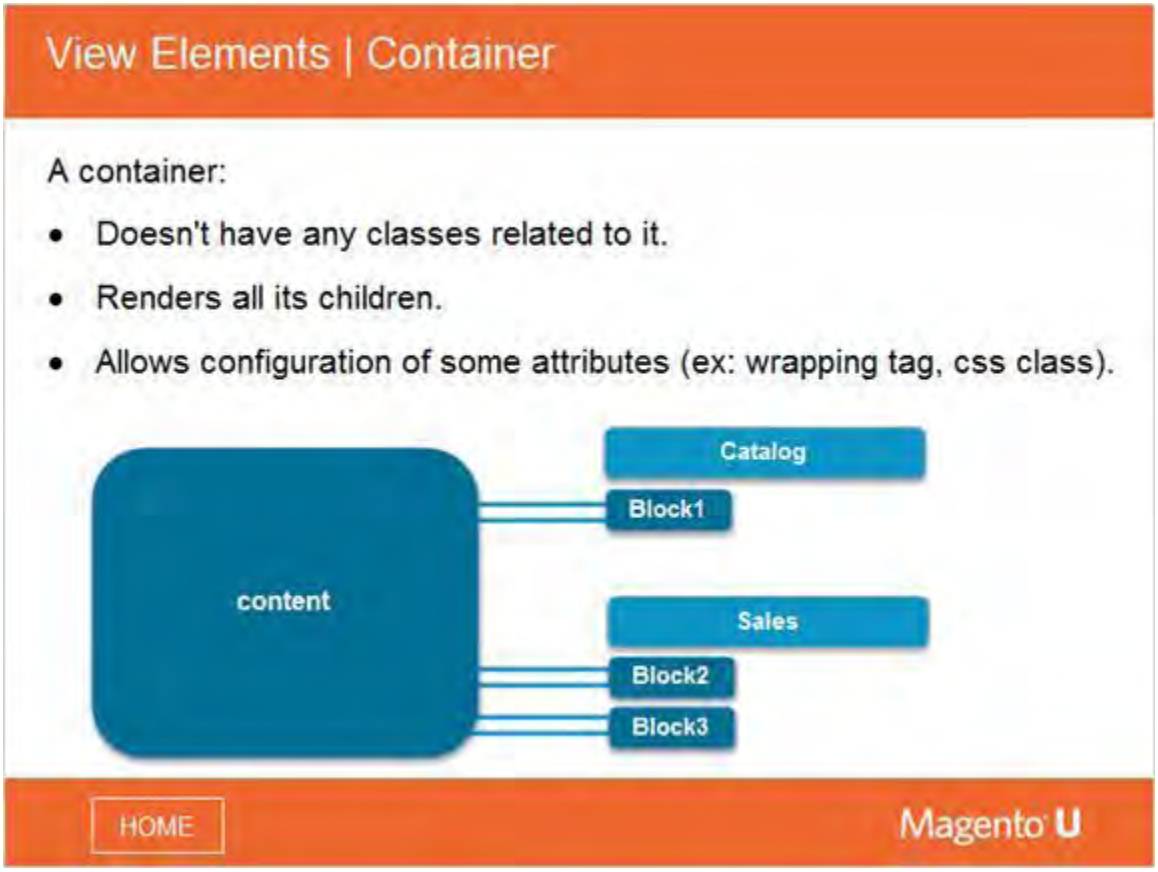
HOME

Magento U

Notes:

The code snippet on the slide shows how the `UiComponent` is rendered in the same way as a block.

4.7 View Elements | Container



Notes:

Simply attach any element to a container and it will render it automatically. With a container, you can define wrapping tags, CSS classes, and more.

The purpose of developing containers in the beginning was to create events in templates. You don't have to rewrite a template, you just place the code. This ended up being the same as text list blocks in Magento 1.

In Magento 2, the name 'container' is new but the concept behind it is familiar. Later we will see how container is rendered.

You cannot create instances of containers because they are an abstract concept, whereas you can create instances of blocks. You still have an analog of `core/text_list` blocks in Magento 2 and you can use them in the same way as in Magento 1.

4.8 View Elements | Render Container (View/Layout)

View Elements | Render Container (View/Layout)

```
protected function _renderContainer($name)
{
     $html = '';
     $children = $this->getChildNames($name);
     foreach ($children as $child) {
         $html .= $this->renderElement($child);
     }
     if ($html == '' || !$this->structure->getAttribute($name,
                                        Element::CONTAINER_OPT_HTML_TAG)) {
         return $html;
     }
        $htmlId = $this->structure->getAttribute($name, Element::CONTAINER_OPT_HTML_ID);
        if ($htmlId) {
            $htmlId = ' id="' . $htmlId . '"';
        }
        $htmlClass = $this->structure->getAttribute
                        ($name, Element::CONTAINER_OPT_HTML_CLASS);
        if ($htmlClass) {
            $htmlClass = ' class="' . $htmlClass . '"';
        }
        $htmlTag = $this->structure->getAttribute($name,
                                            Element::CONTAINER_OPT_HTML_TAG);
        $html = sprintf('<%1$s%2$s%3$s>%4$s</%1$s>',$htmlTag,$htmlId,$htmlClass, $html);
        return $html;
 }
```

HOME

Magento U

Notes:

This slide shows how a container is rendered. As we discussed, Magento won't create an instance of container, but will render all its children. We can see from this code, there are a couple of new possibilities that container brings when compared to the `core/text_list` block:

- Ability to define a wrapping tag.
- Ability to set attributes to the wrapping tag.

4.9 Code Demonstration | RenderContainer

Notes:

4.10 View Elements | Empty.xml

View Elements | Empty.xml

```
<?xml version="1.0"?>
<layout xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xsi:noNamespaceSchemaLocation="../../../../../../../lib
                    /internal/Magento/Framework/View/Layout/etc/page_layout.xsd">
    <container name="root">
       <container name="after.body.start" as="after.body.start" before="-"
                                                        label="Page Top"/>
       <container name="page.wrapper" as="page_wrapper" htmlTag="div"
                                              htmlClass="page-wrapper">
       <container name="global.notices" as="global_notices" before="-"/>
       <container name="main.content" htmlTag="main" htmlId="maincontent"
                                              htmlClass="page-main">
       <container name="columns.top" label="Before Main Columns"/>
       <container name="columns" htmlTag="div" htmlClass="columns">
       <container name="main" label="Main Content Container" htmlTag="div"
                                               htmlClass="column main"/>
       </container>
    </container>
       <container name="page.bottom" as="page_bottom" label="Before Page Footer
         Container" after="main.content" htmlTag="div" htmlClass="page-bottom"/>
       <container name="before.body.end" as="before_body_end" after="-"
                                              label="Page Bottom"/>
       </container>
    </container>
</layout>
```

HOME

Magento U

Notes:

This slide shows an example of `empty.xml`. It defines a basic page body structure by defining a group of containers. This is an analogue to `root.phtml` in Magento 1.

4.11 View Elements | Assign an Element to a Container

View Elements | Assign an Element to a Container

1column.xml

```
<layout xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:noNamespaceSchemaLocation= "../../../../../../
              ../lib/internal/Magento/Framework/View/Layout/
              etc/page_layout.xsd">
    <update handle="empty"/>
    <referenceContainer name="page.wrapper">
        <container name="header.container" as="header_container"
              label="Page Header Container"  htmlTag="header"
              htmlClass="page-header" before="main.content"/>
        <container name="page.top" as="page_top"
              label="After Page Header"
              after="header.container"/>
        <container name="footer-container" as="footer"
              before="before.body.end" label="Page Footer Container"
              htmlTag="footer" htmlClass="page-footer" />
    </referenceContainer>
</layout>
```

HOME

Magento U

Notes:

This slide demonstrates a `1column.xml` file. As you can see, it updates containers from `empty.xml`, generating a 1-column page structure.

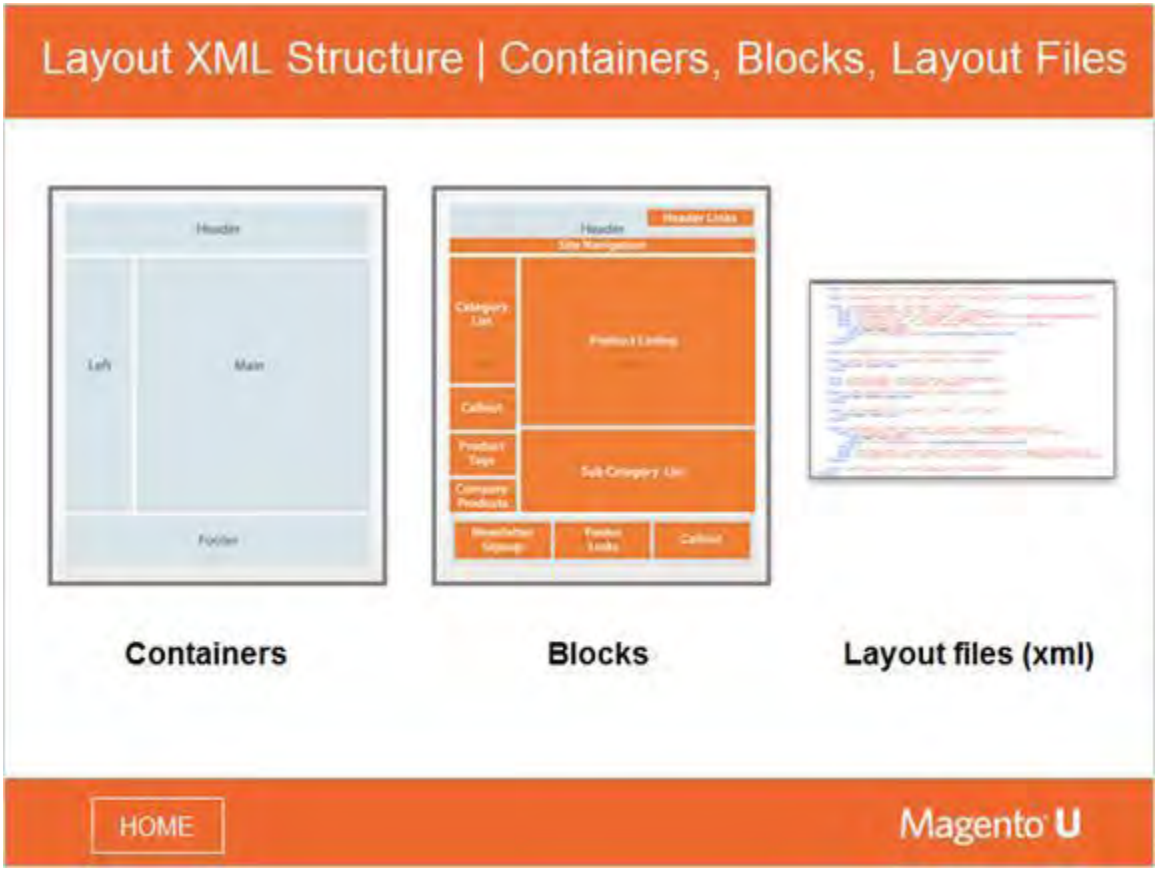
In Magento 1, we have the `1column.phtml` file; Magento 2 has an xml equivalent.

4.12 Blocks Definition

Notes:

In this section, we will go into more detail on the role of blocks in layout.

4.13 Containers, Blocks, Layout Files



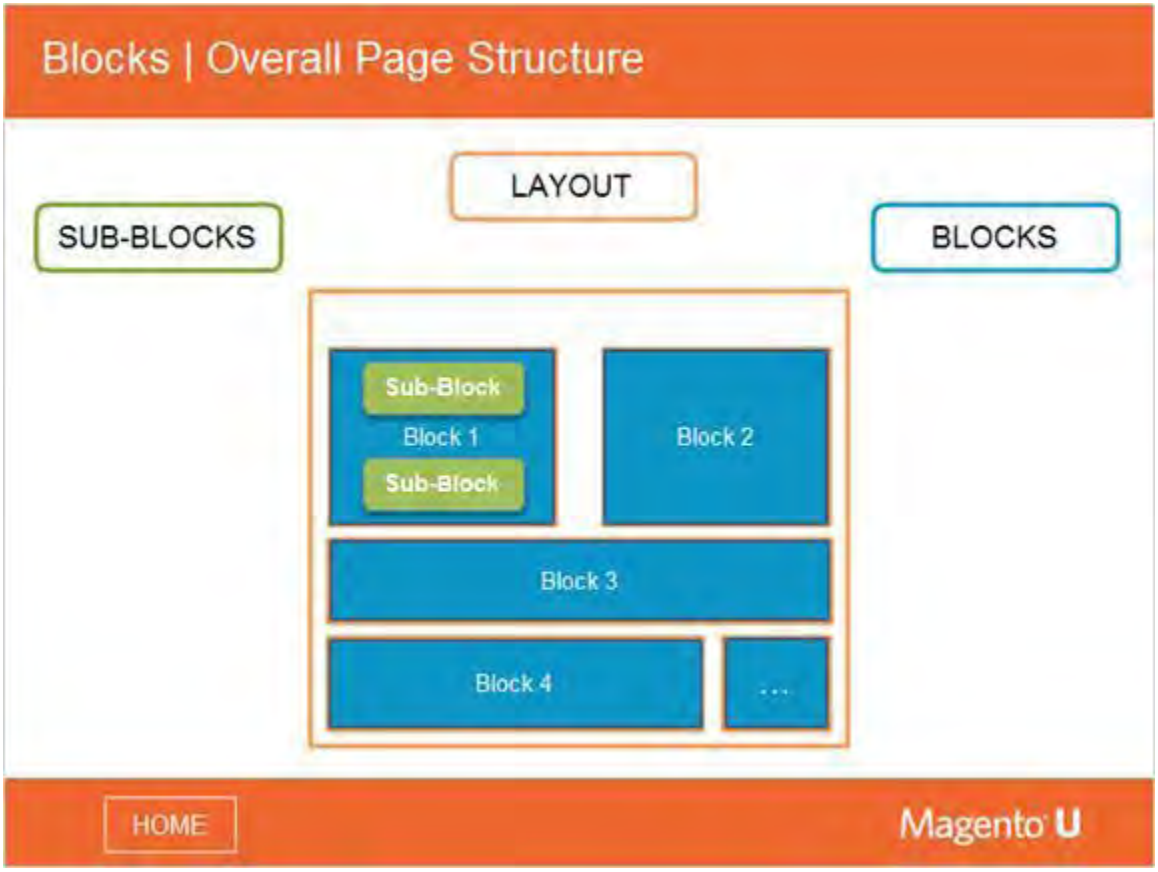
Notes:

The layout of a page is determined by containers, which act as a framework and do not contain actual content. The content segments within a container are called blocks. Examples for content blocks might be a filter navigation, a page title or a product add-to-cart form. Blocks can also be viewed as part of a page, an element of a page. Most blocks are templates, but some are not.

Each web page within Magento is a hierarchy of containers comprised of blocks that, in turn, can contain any number of child content blocks or child containers, and provide handy extension points to other modules that place content on the page in the same area.

Layout structure is discussed in detail in a later section.

4.14 Overall Page Structure

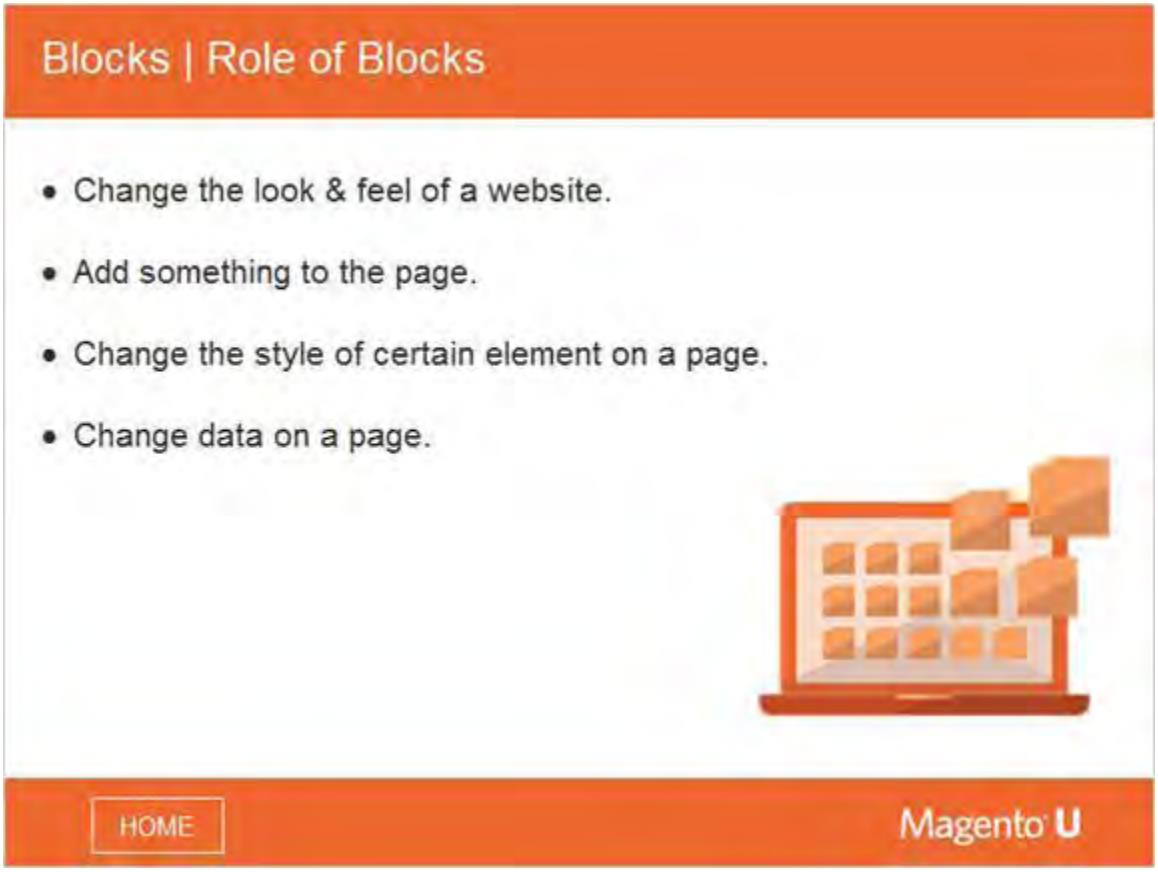


Notes:

Every page consists of a hierarchical structure of blocks. The layout does not define the location of blocks on the page; it just defines their sequence. How the blocks actually look on the page depends of how the page is rendered and on the CSS.

The slide diagram shows the relationship of blocks to the page structure.

4.15 Role of Blocks

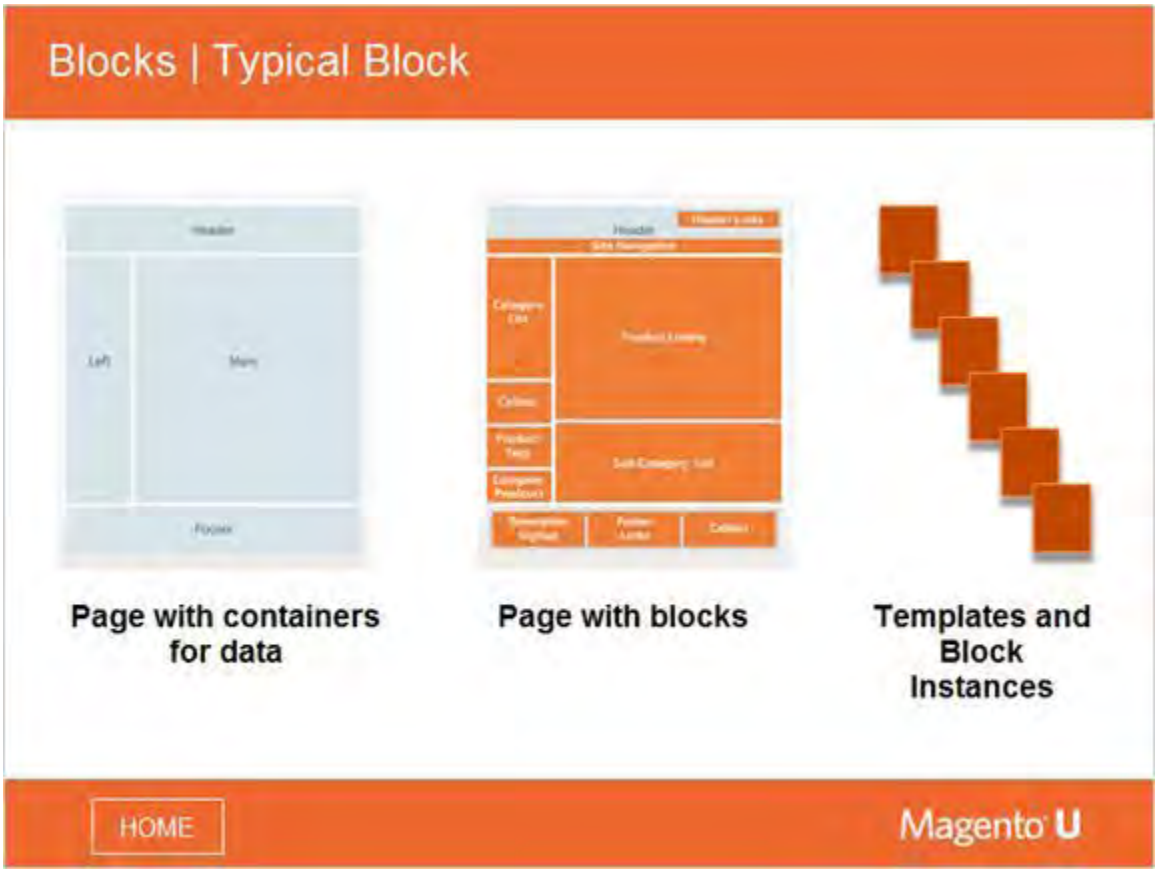


Notes:

There are different types of UI customizations that you can make to a website. For example, you can change the look and feel, add an image to the page, or change the style of the fonts on the product listing page.

In general, changes to the look and feel of a site involve the layout, while adding something to a page most likely involves blocks. Changing the style of an element on a page can be accomplished using CSS code.

4.16 Typical Block



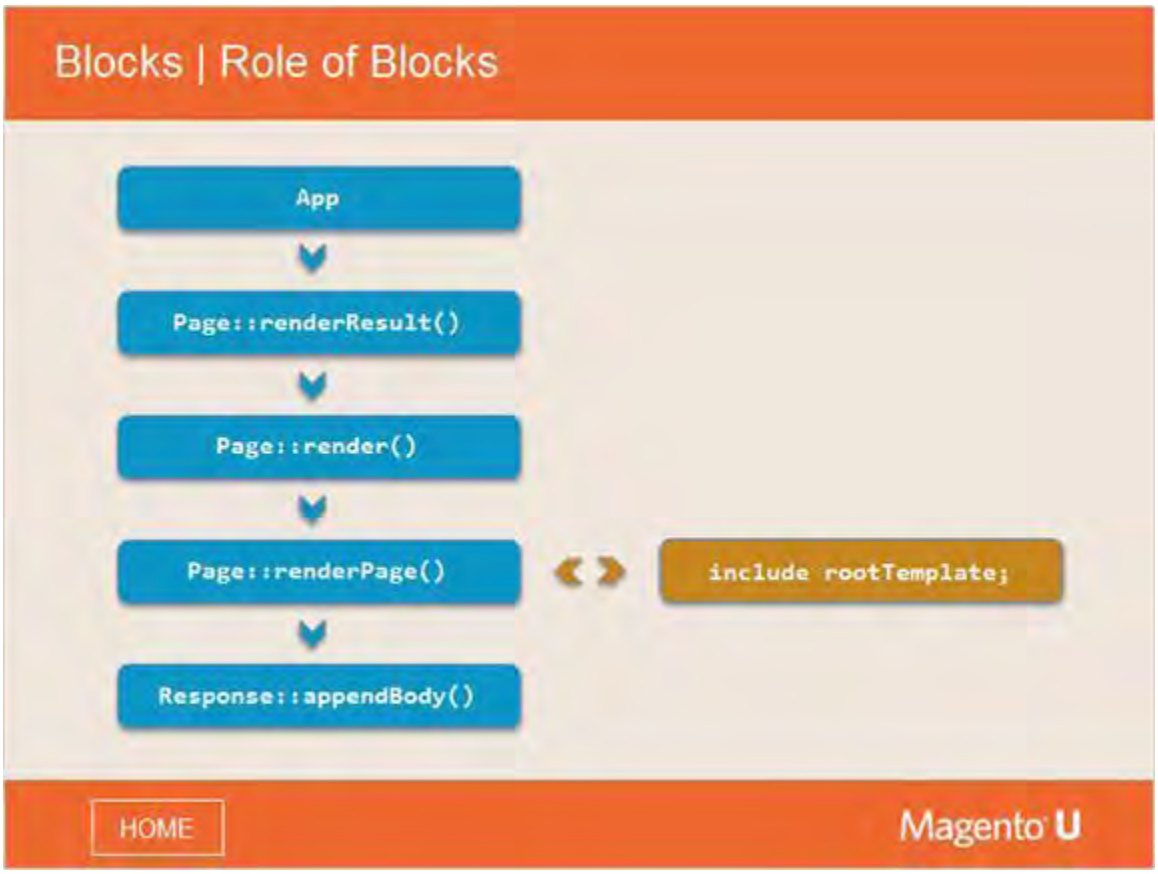
Notes:

In Magento 2, the structure of a page is more complex than in Magento 1, but conceptually, they are the same - the structure is defined by layout xml files, and blocks (usually connected to templates) are generating content in that structure.

Within each template, access to the block's instance provides access to the data.

So templates draw elements within a page, and blocks contain the data (or contain a method that allows the template access to the data).

4.17 Role of Blocks



Notes:

It is the `Page::render()` method that renders the root template, which includes other content blocks. As mentioned earlier, the role of the blocks is in the adding content to the structure defined by layout xml. Layout xml is a hierarchical structure of blocks. Magento renders those blocks and generates html content to be included into `$layoutContent`.

For a high-level overview:
Magento includes `root.phtml`. `$layoutContent` is included In its <body> section. Layout content is generated, based on `empty.xml`, and `1(2,3)column.xml`, together with all layout updates for the current page. This all generates layout xml for the page.

We will cover this process in more detail later.

4.18 Root Template

Blocks | Root Template

```
<!doctype html>
<html <?php echo $htmlAttributes ?>>
    <head <?php echo $headAttributes ?>>
        <?php echo $requireJs ?>
        <?php echo $headContent ?>
        <?php echo $headAdditional ?>
    </head>
    <body data-container="body" data-mage-init='{
         "loaderAjax": {}, "loader": {
         "icon": "<?php echo $loaderIcon; ?>"}}'
         <?php echo $bodyAttributes ?>>
         <?php echo $layoutContent ?>
    </body>
</html>
```

HOME

Magento U

Notes:

This is the code for root template; `$layoutContent` is basically where all layouts are rendered.

As you can see, the root template is only a few lines of code. It contains PHP `echo` commands, which display certain variables.

The question is - where do these variables come from?

To answer this, we need to go back to the `render()` method.

4.19 Page::Render()

Blocks | Page::render()

```
protected function render(ResponseInterface $response)
   {
       $this->pageConfig->publicBuild();
       if ($this->getPageLayout()) {
           $config = $this->getConfig();
           $this->addDefaultBodyClasses();
           $addBlock = $this->getLayout()->getBlock('head.additional');  //todo
           $requireJs = $this->getLayout()->getBlock('require.js');
           $this->assign([
               'requireJs' => $requireJs ? $requireJs->toHtml() : null,
               'headContent' => $this->pageConfigRenderer->renderHeadContent(),
               'headAdditional' => $addBlock ? $addBlock->toHtml() : null,
               'htmlAttributes' => $this->pageConfigRenderer
                       ->renderElementAttributes($config::ELEMENT_TYPE_HTML),
               'headAttributes' => $this->pageConfigRenderer
                       ->renderElementAttributes($config::ELEMENT_TYPE_HEAD),
               'bodyAttributes' => $this->pageConfigRenderer
                       ->renderElementAttributes($config::ELEMENT_TYPE_BODY),
               'loaderIcon' => $this->getViewFileUrl('images/loader-2.gif'), ]);
           $output = $this->getLayout()->getOutput();
           $this->assign('layoutContent', $output);
           $output = $this->renderPage();
           $this->translateInline->processResponseBody($output);
           $response->appendBody($output);
```

HOME

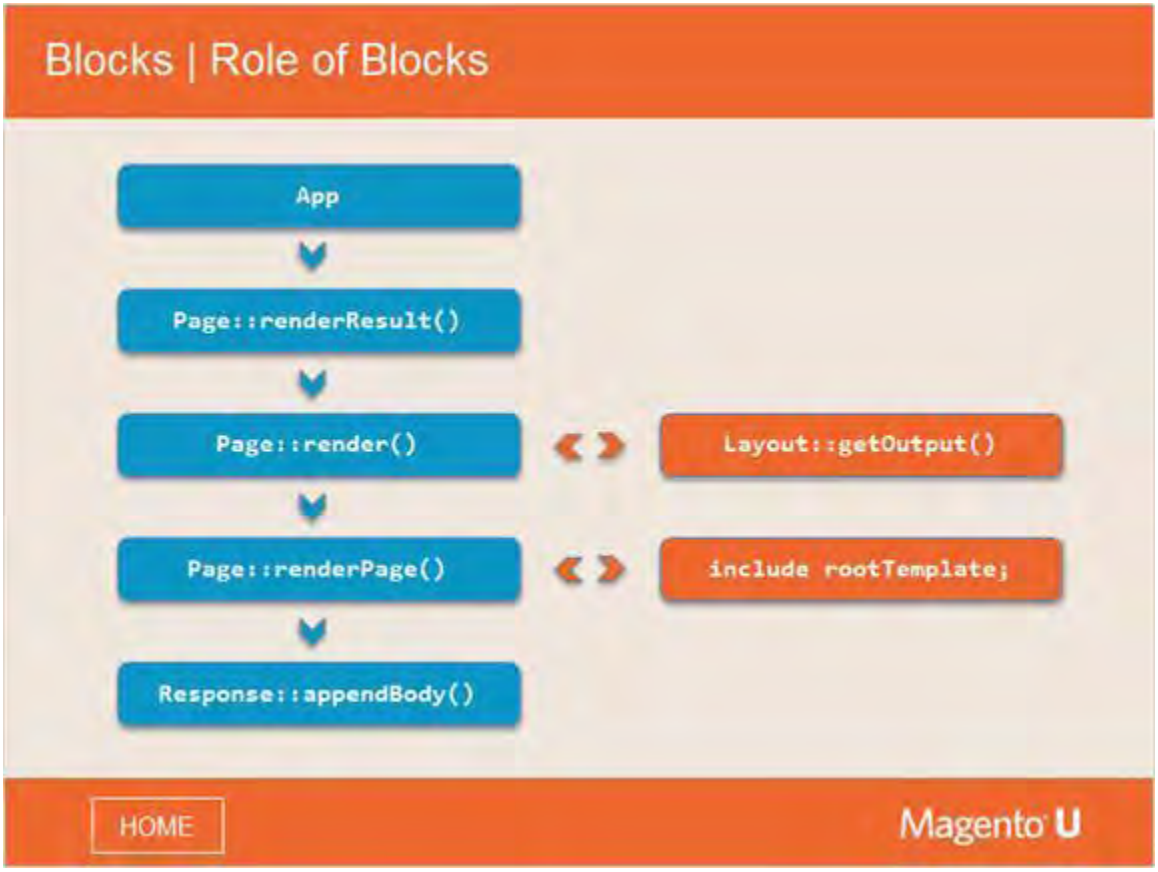
Magento U

Notes:

The code highlighted in orange shows the layout output, and how the generated `$layoutContent` is being assigned to be included in `root.phtml` later.

The data comes from `Page::render()`, the parameters come from the layout, and the `$layoutContent` variable comes from `$this->getLayout()->getOutput()`.

4.20 Role of Blocks

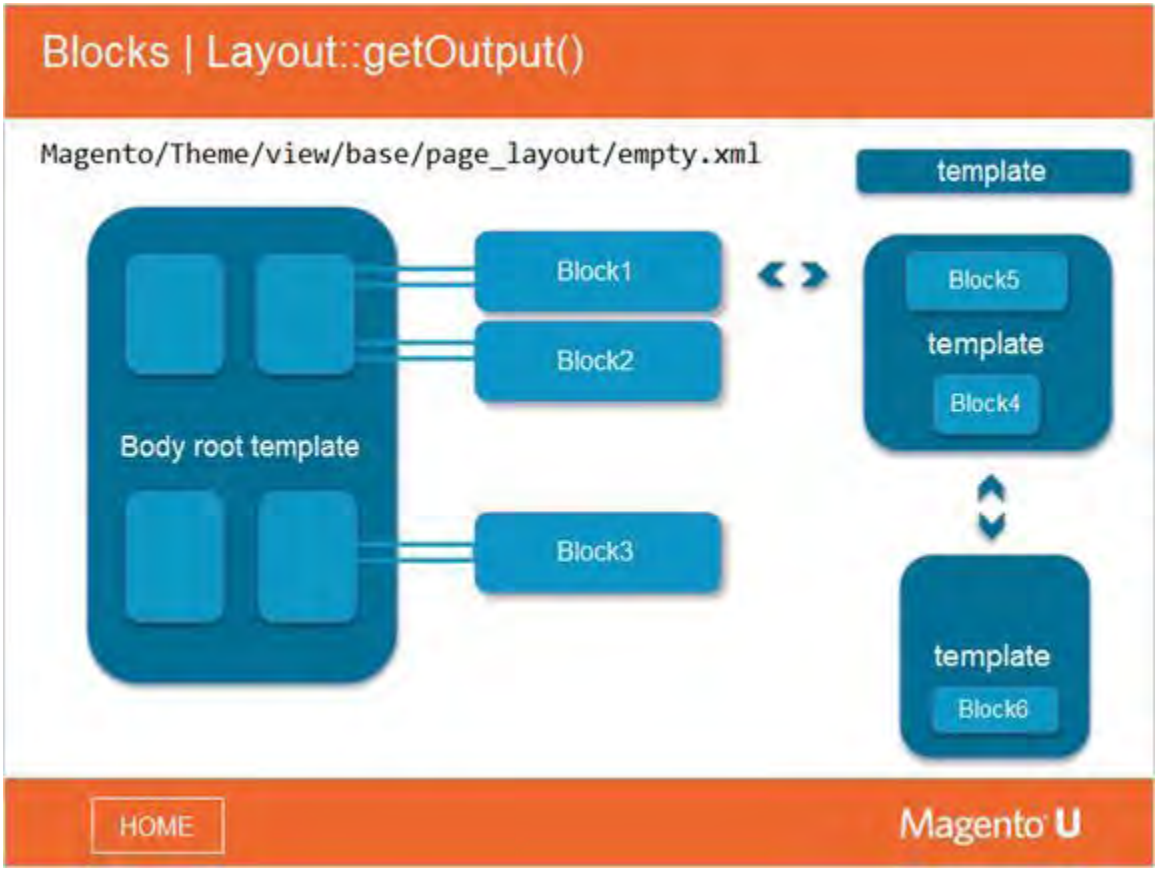


Notes:

Now we can improve the flow diagram. Note that before including the `rootTemplate`, we have the step `Layout::getOutput()`, next to `Page::render()`. So, it will go to `Page::render()`, then `Layout::getOutput()`, before it comes back to the normal flow.

When `Layout::getOutput()` is executed, most elements on the page are rendered. This is important because blocks play a big role here.

4.21 Blocks | Layout::getOutput()



Notes:

This slide illustrates what is happening with the `Layout::getOutput()`.

Recall that in Magento 1, `getOutput()` goes through every output block and performs a call to the method specified in the “output” attribute, which is always toHtml().

In Magento 2, the rendering system will take into account the `view/base/page_layout/empty.xml`. It `(empty.xml)` consists of a list of containers, and every container has some blocks attached to it by other layout updates.

Note that each container has blocks assigned to it. Each block will usually (but not always) render a template. The template may or may not call another block, which would then call another template, and so on.

A block is rendered in Magento 2 when it is called from the template. So even if it is generated, a block will not be rendered until it is called from another template. (There are a few exceptions, such as the core block).

5. Block Architecture & Lifecycle

5.1 Block Architecture & Lifecycle

Notes:

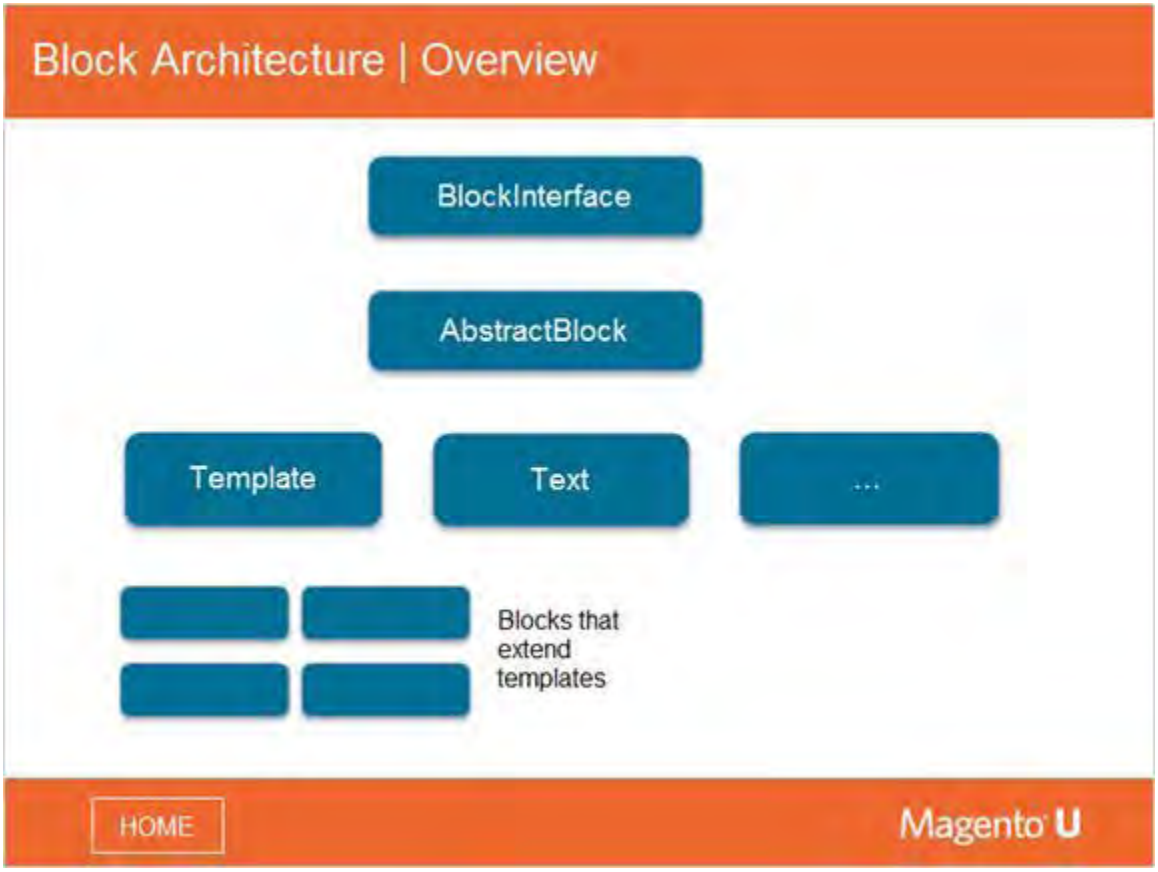
This module discusses block architecture and lifecycle.

5.2 Module Topics

Notes:

In this module, we will study blocks in further depth, examining their architecture and lifecycle.

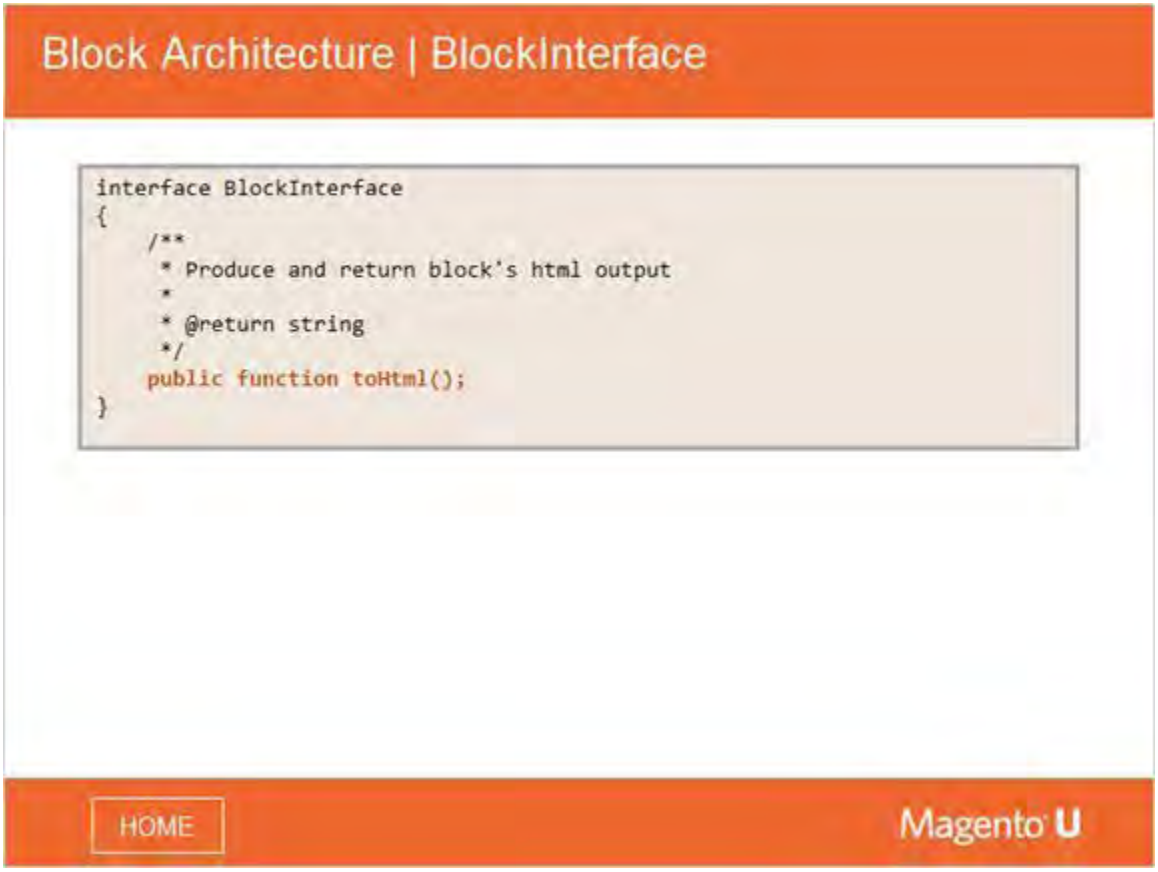
5.3 Block Architecture | Overview



Notes:

We will start our analysis with `BlockInterface` and `AbstractBlock`.

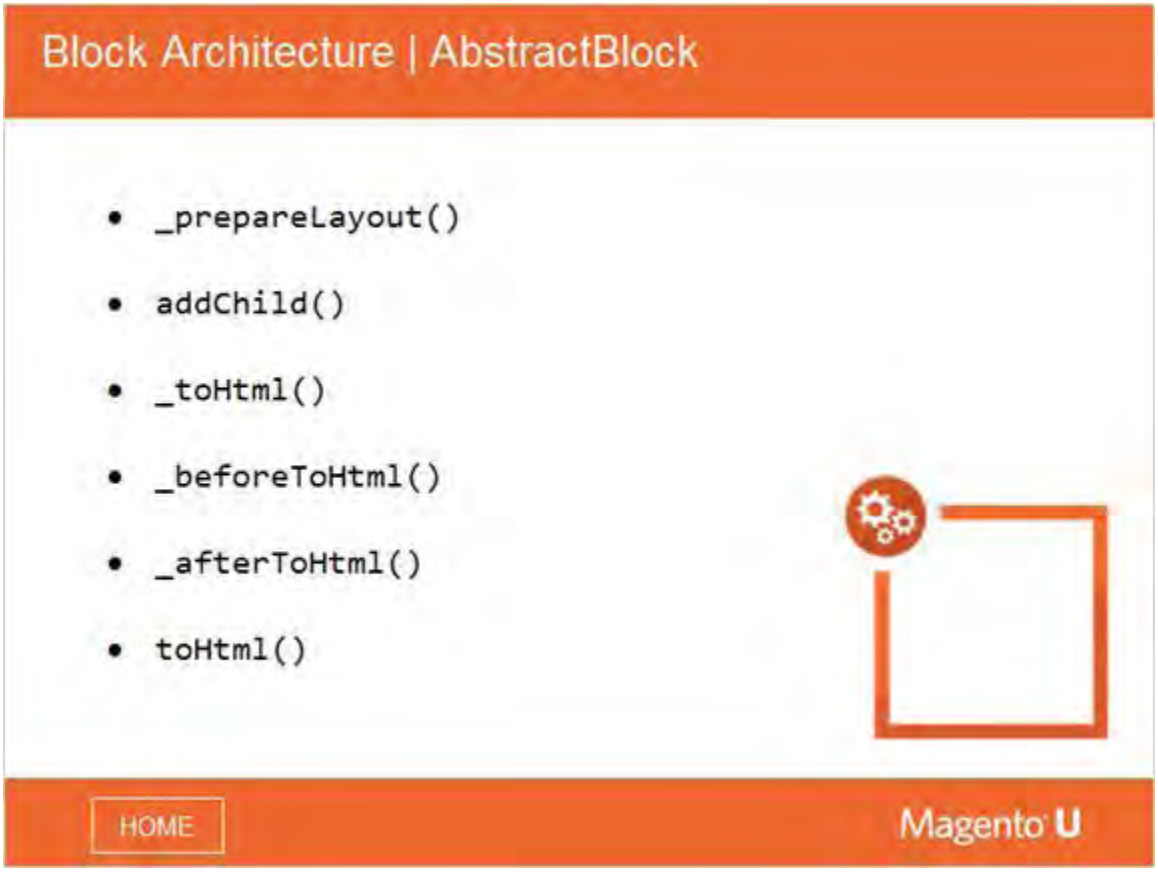
5.4 Block Architecture | BlockInterface



Notes:

`BlockInterface` has only one method to be implemented.

5.5 Block Architecture | AbstractBlock



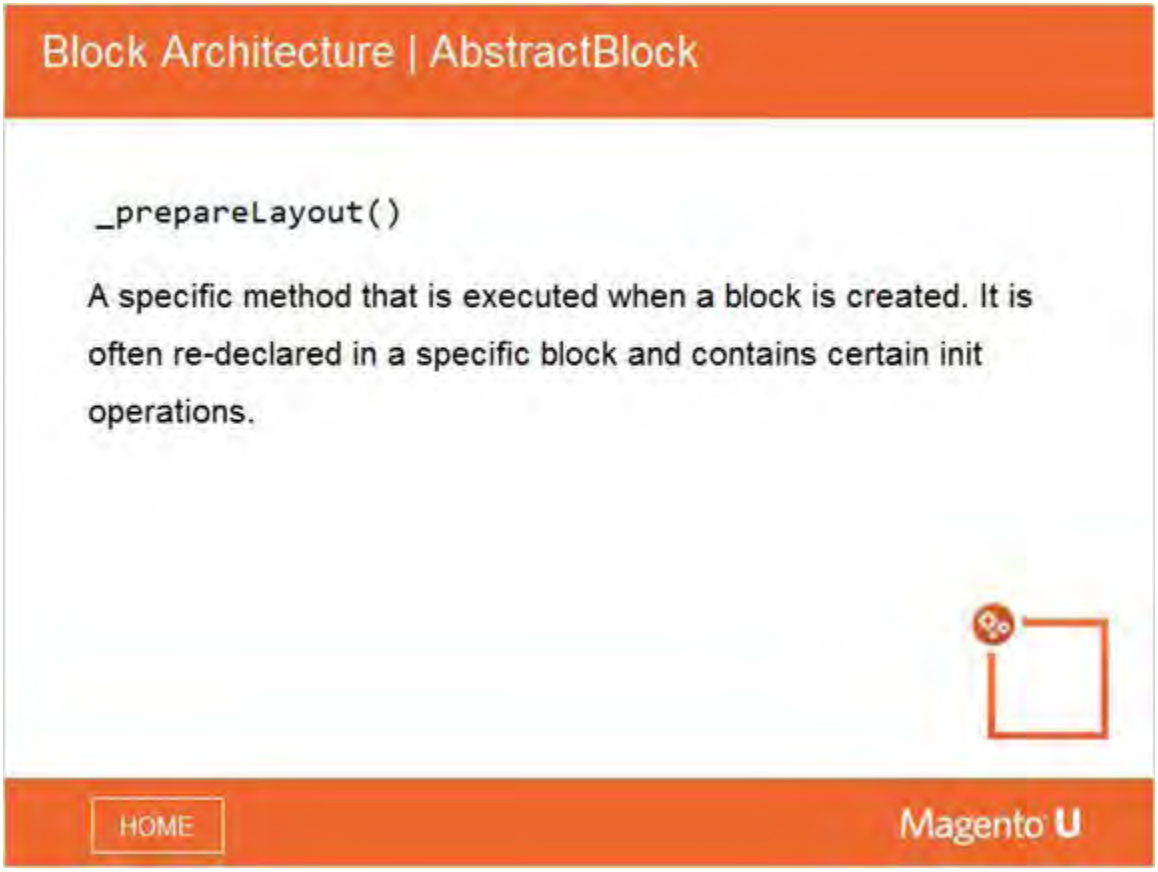
Notes:

On this slide, you can see a list of important methods that the `AbstractBlock` class contains, namely:

- `_prepareLayout()`
- `addChild()`
- `_toHtml()`
- `_beforeToHtml()`
- `_afterToHtml()` `toHtml()`

Later on in this chapter, while analyzing the different phases of the block lifecycle, we will discuss these methods further.

5.6 Block Architecture | AbstractBlock

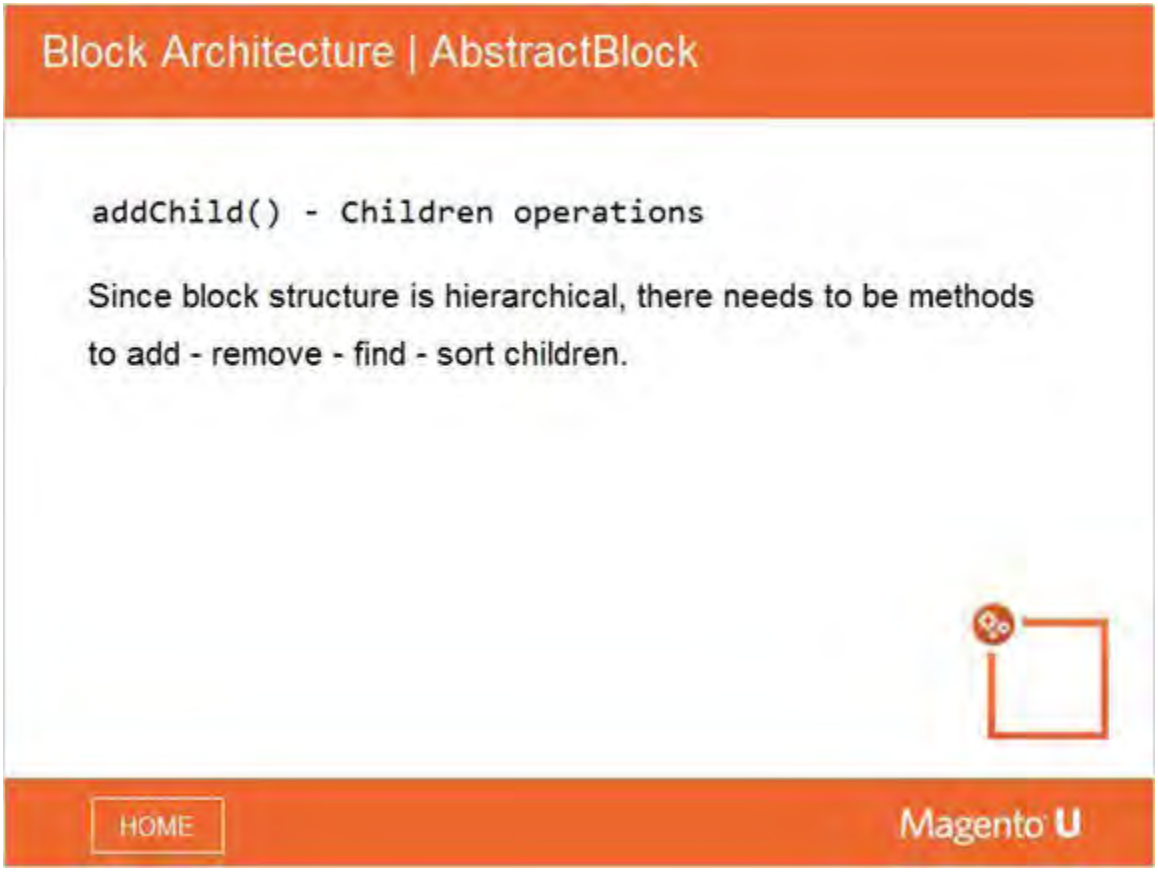


Notes:

`_prepareLayout()`

A specific method that is executed when a block is created. It is often re-declared in a specific block and contains certain init operations.

5.7 Block Architecture | AbstractBlock

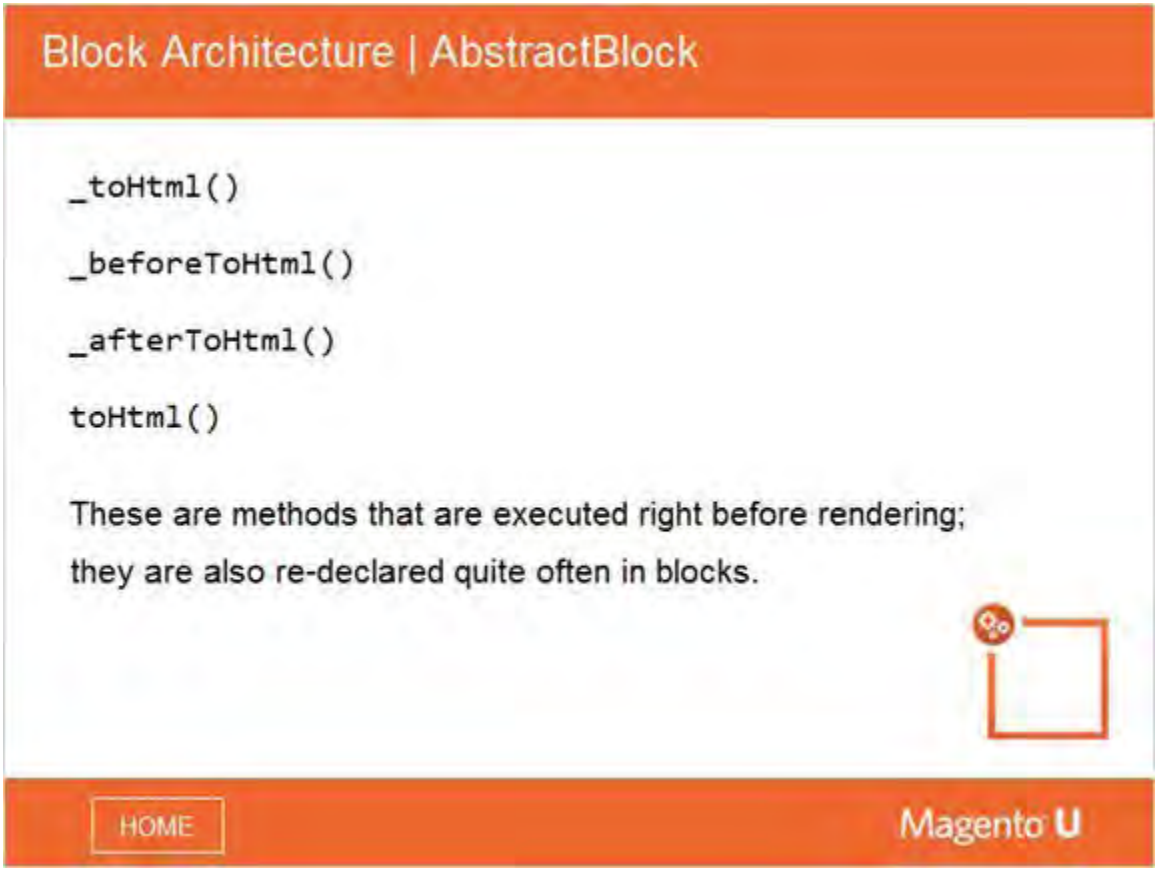


Notes:

Children operations:

Since block structure is hierarchical, there needs to be methods to add - remove - find - sort children.

5.8 Block Architecture | AbstractBlock



Notes:

- `_toHtml()`
- `_beforeToHtml()`
- `_afterToHtml()`
- `toHtml()`

These are methods that are executed right before rendering; they are also re-declared quite often in blocks.

5.9 Block Architecture | AbstractBlock::toHtml()

Block Architecture | AbstractBlock::toHtml()

```
public function toHtml()
   {
       $this->_eventManager->dispatch('view_block_abstract_to_html_before',
                                      ['block' => $this]);
       if ($this->_scopeConfig->getValue(
           'advanced/modules_disable_output/' . $this->getModuleName(),
           \Magento\Framework\Store\ScopeInterface::SCOPE_STORE
       )) {
           return '';
       }
   $html = $this->_loadCache();
   if ($html === false) {
       if ($this->hasData('translate_inline')) {
            $this->inlineTranslation->suspend($this->getData('translate_inline')); }
           $this->_beforeToHtml();
           $html = $this->_toHtml();
           $this->_saveCache($html);

           if ($this->hasData('translate_inline')) {
             $this->inlineTranslation->resume();  }
      }
       $html = $this->_afterToHtml($html);
      return $html;
   }
```

HOME

Magento U

Notes:

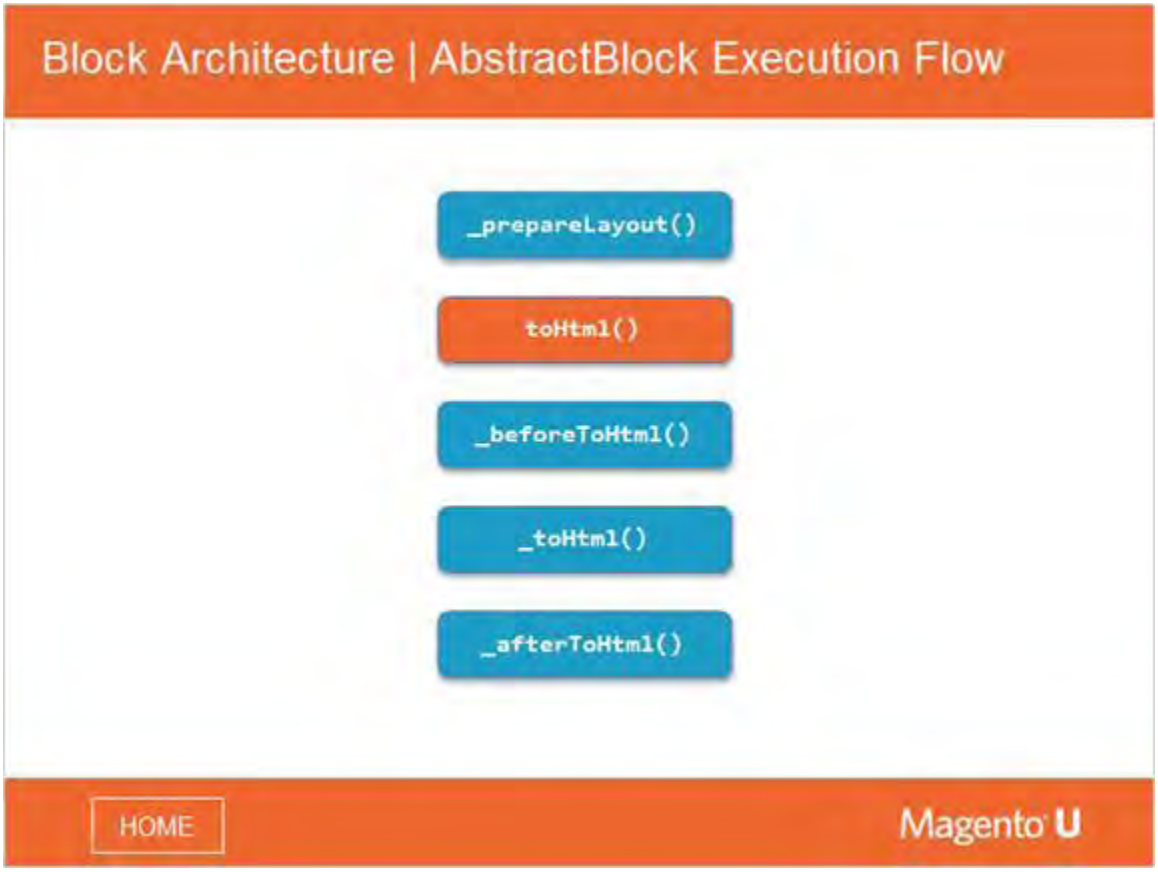
Note that `toHtml()` needs to be implemented.

`_toHtml()` is the place in the code where rendering actually happens, and `_beforeToHtml()` is a trigger that allows certain actions to be performed right before rendering.

The most important code lines in the example are the following:

```
$this->_beforeToHtml();
$html = $this->_toHtml();
…
$html = $this->_afterToHtml($html)
```


5.10 Block Architecture | AbstractBlock Execution Flow

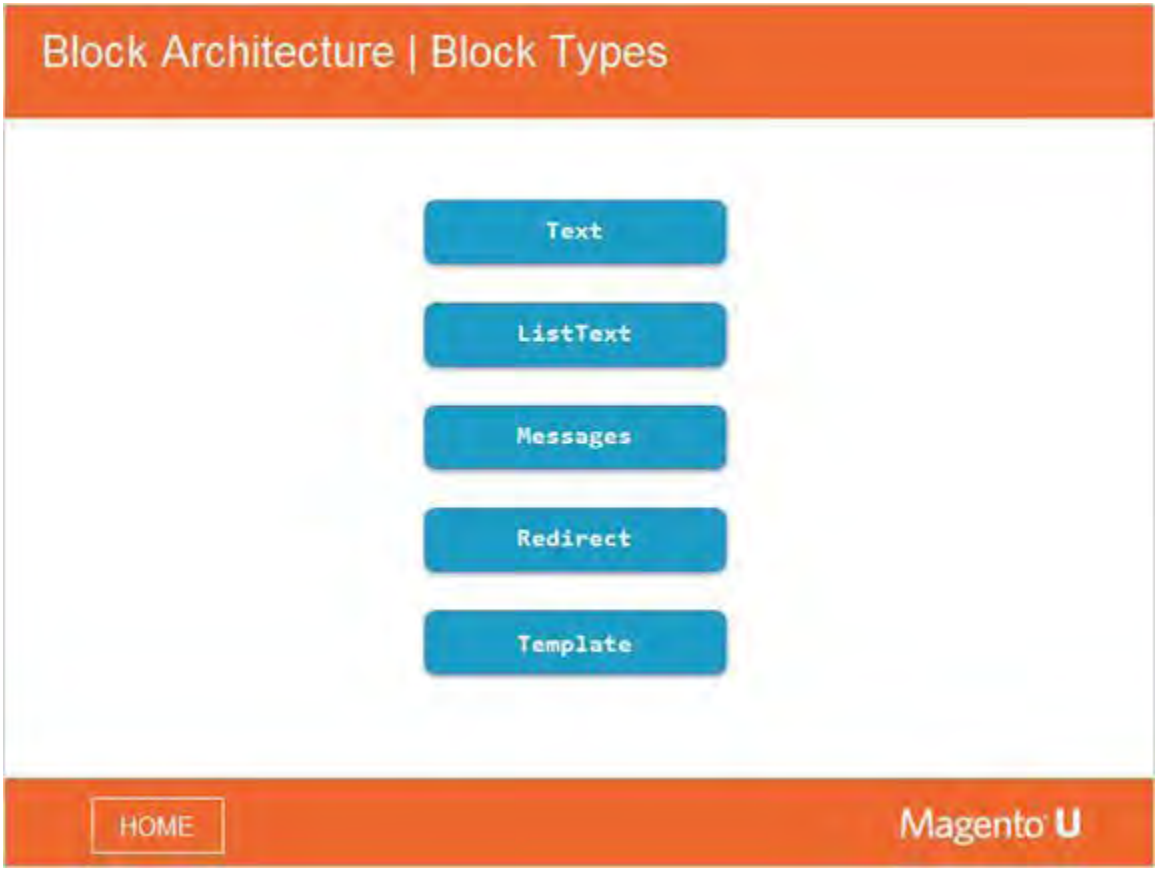


Notes:

This diagram depicts the `AbstractBlock` execution flow. It starts with `_prepareLayout()` and flows through a set of methods until it reaches `_afterToHtml()`. This is, in essence, what you have to know about block execution flow.

Note that `toHtml()` is final in Magento 1; in Magento 2, it is not final and it is not recommended to override this method. What is recommended is to override `_toHtml()`. That is where every block type can implement its specific rendering logic.

5.11 Block Architecture | Block Types



Notes:

Let's take a look at different block types. The most important block types are: Text, ListText, Messages, Redirect, and Template.

What is block type?

It is basically an implementation of an abstract block. We have seen previously that in the abstract block flow, the `_toHtml()` method has to be implemented for every block type because it is not defined.

The different classes extend the abstract block. Abstract block implements its own `_toHtml()` method, where the `_toHtml()` method is called. Each specific block type implements `_toHtml()`, where the type-specific rendering logic is located.

5.12 Block Architecture | Block Types - Text

Block Architecture | Block Types - Text

```
public function addText($text, $before = false)
   {
     if ($before) {
           $this->setText($text . $this->getText());
       } else {
           $this->setText($this->getText() . $text);
       }
   }

   /**
    * Render html output
    *
    * @return string
    */
   protected function _toHtml()
   {
       if (!$this->_beforeToHtml()) {
           return '';
       }
       return $this->getText();
   }
```

HOME

Magento U

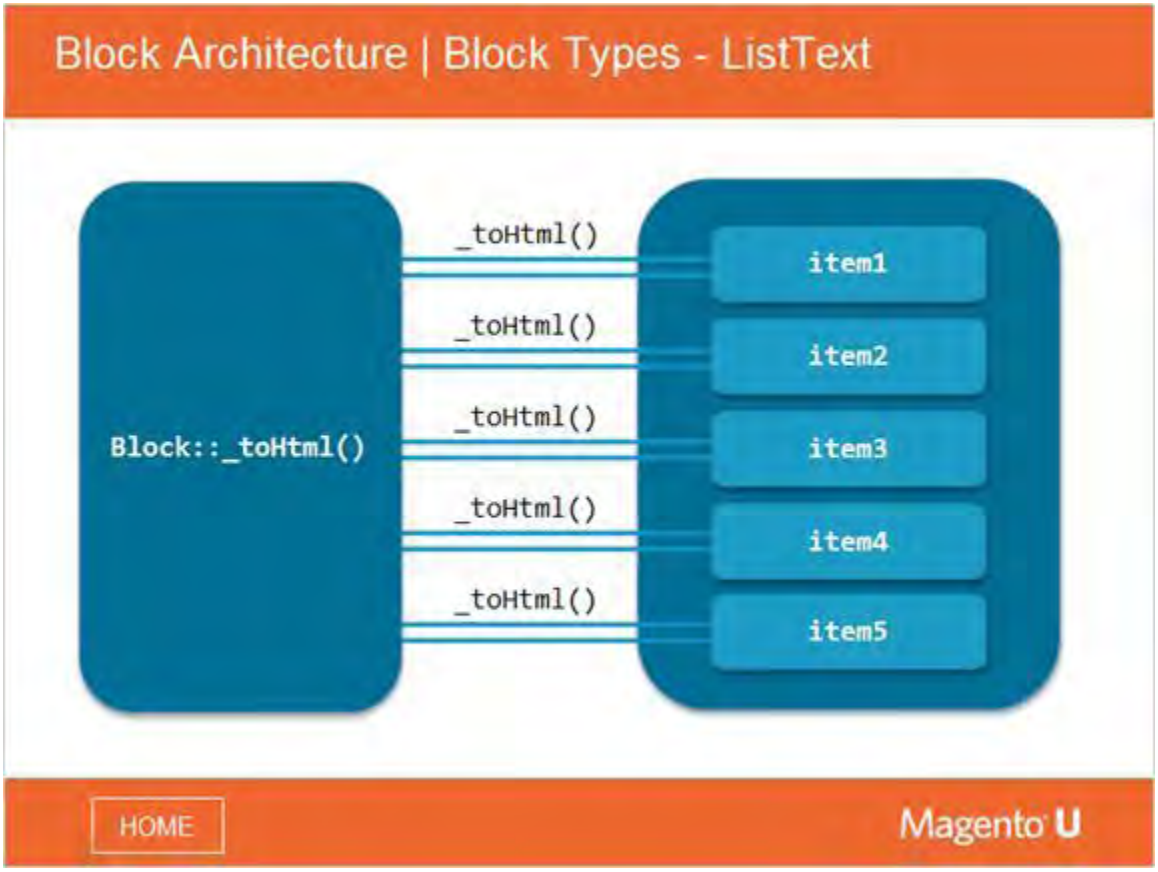
Notes:

To understand what the text block type does, let's take a look at its `_toHtml()` method.

The most important code line, which demonstrates rendering, is highlighted in the example.

```
Return $this->getText()
```

5.13 Block Architecture | Block Types - ListText



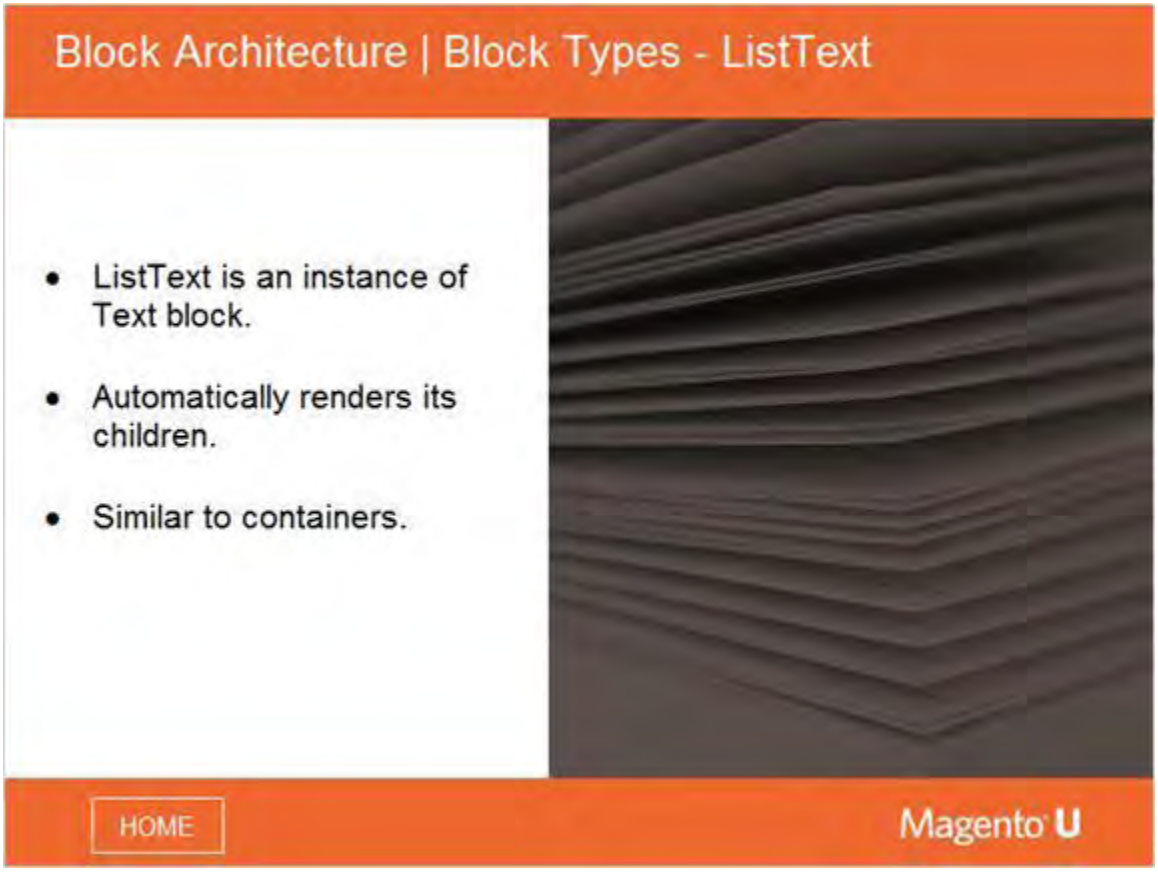
Notes:

In this diagram, items 1 to 5 represent child 1 to 5.

In other words, the block's `_toHtml()` goes to every child and calls that child's `toHtml()` method. This is a unique case where the method for `ListText` automatically renders all the children.

Reference: In your Magento 2 installation, locate the file `<magento_root_dir>/lib/internal/Magento/Framework/View/Element/Text/ListText.php`

5.14 Block Architecture | Block Types - ListText



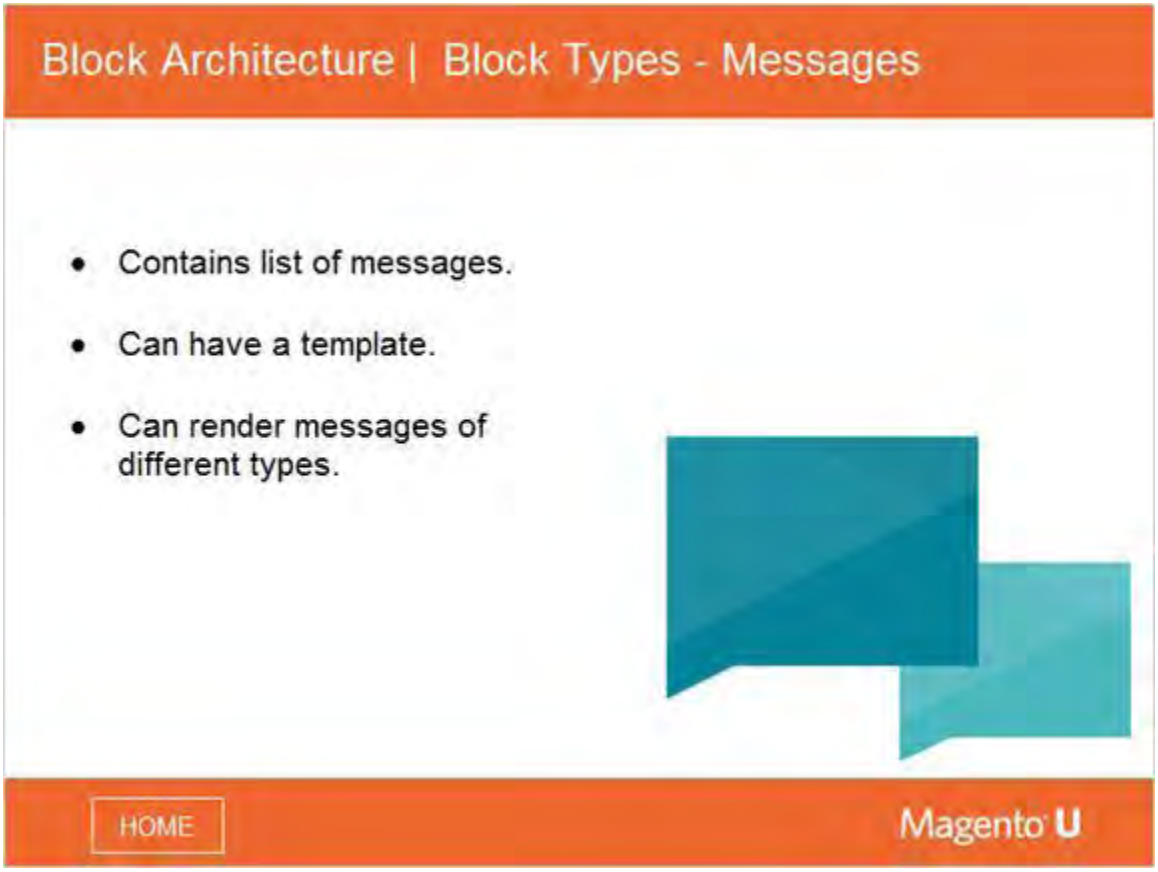
Notes:

`ListText` compiles and renders its children in one text variable. The fact that `ListText` renders its children is an exception.

Usually, block should be called from some other template to be rendered.

Note that this is similar to containers. So, we have several methods that accomplish the same tasks but containers are stricter than `ListText`.

5.15 Block Architecture | Block Types - Messages

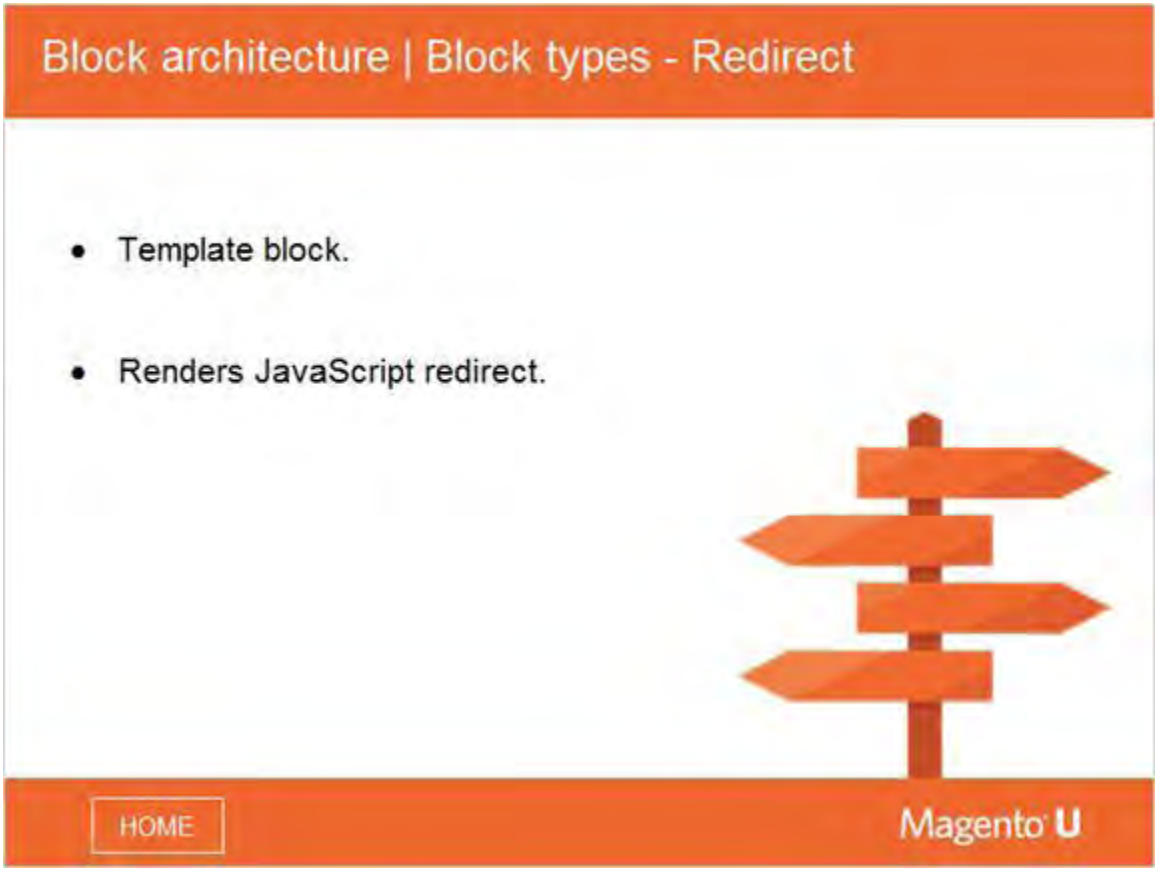


Notes:

The block type, Messages, is designed to render messages in different ways.

For example “error” will be rendered as a red message in a pink box, “note” will be a green message in a beige box.

5.16 Block architecture | Block types - Redirect

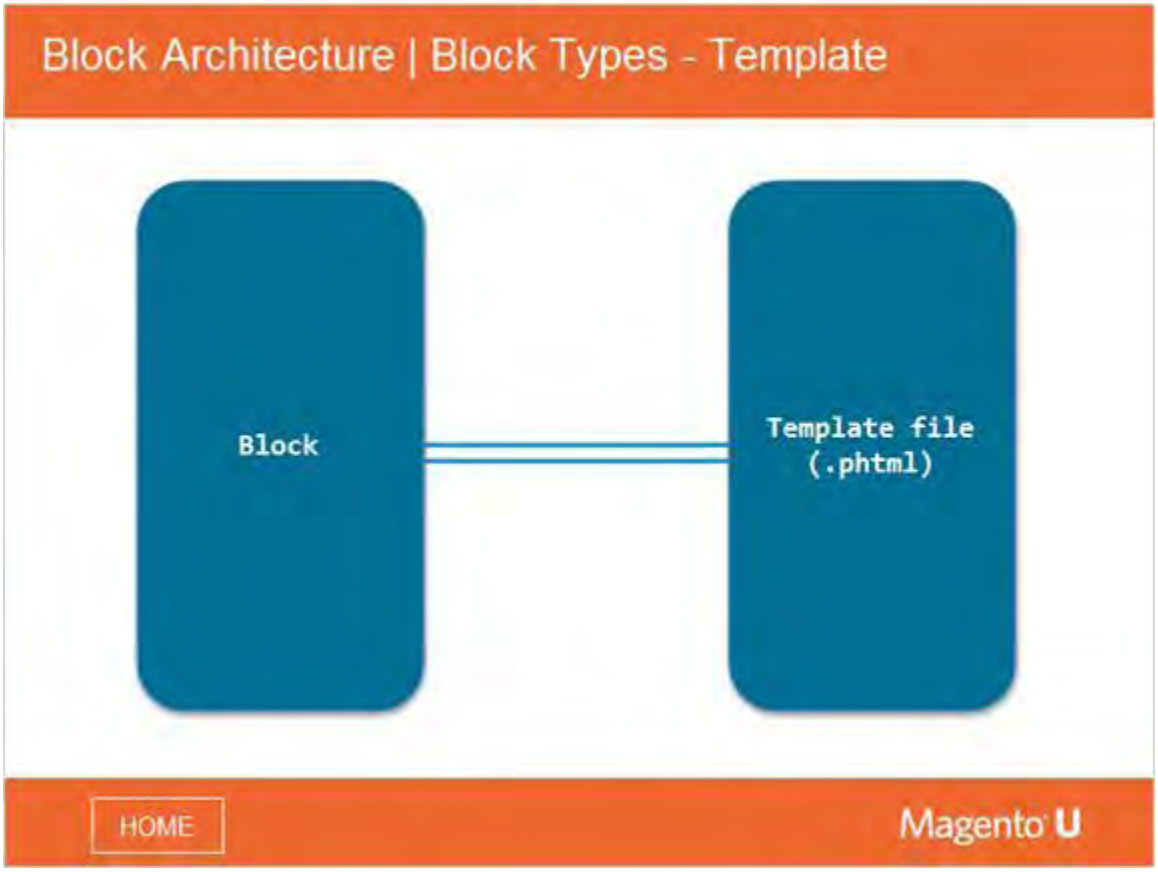


Notes:

Redirect is a popular feature - you click on something and then you have a page that redirects you to another page if you don't see the page you want.

In our case, redirect block renders a JavaScript redirect. There is also an option to perform a PHP redirect but it is not used in this case.

5.17 Block Architecture | Block Types - Template

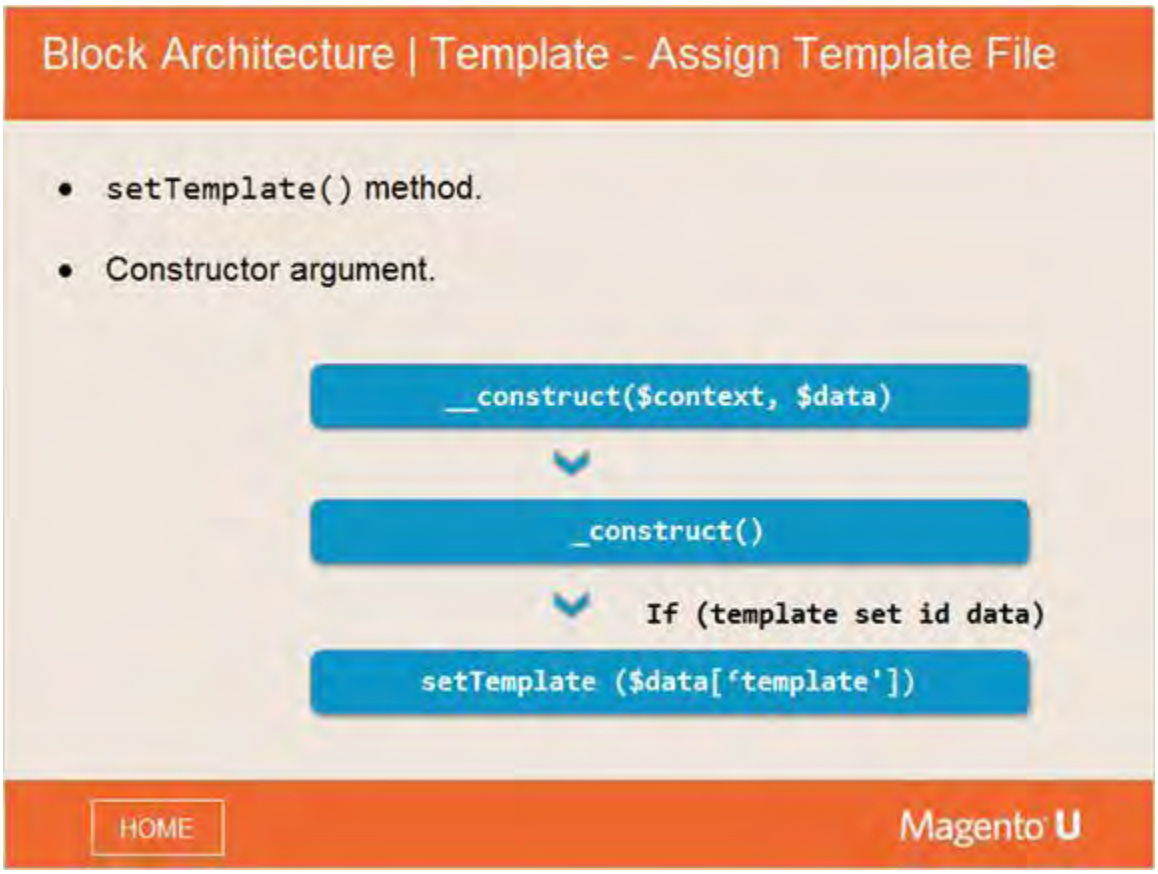


Notes:

When a template block is rendered, the template file is included in the context of a block, and its content is captured with the `ob_start()`, `ob_end_clean()` methods.

We will go into more depth on this process later in the course.

5.18 Block Architecture | Template - Assign Template File



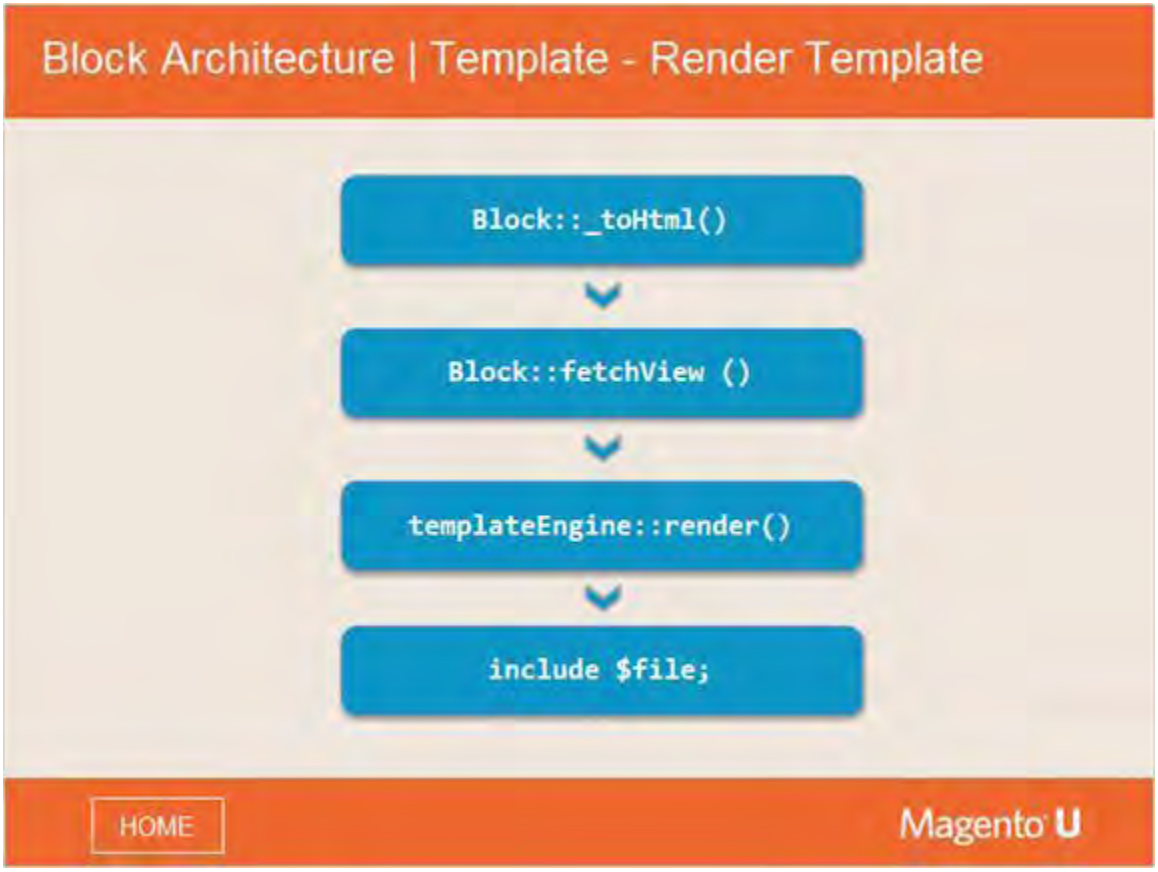
Notes:

There are multiple ways to pass template files to a block:

- `setTemplate()` method - use when you have physical access to the block instance.
- Constructor argument - in the data array; it then goes to the `_construct() method.`

The diagram represents the flow used when changing a template layout.

5.19 Block Architecture | Template - Render Template

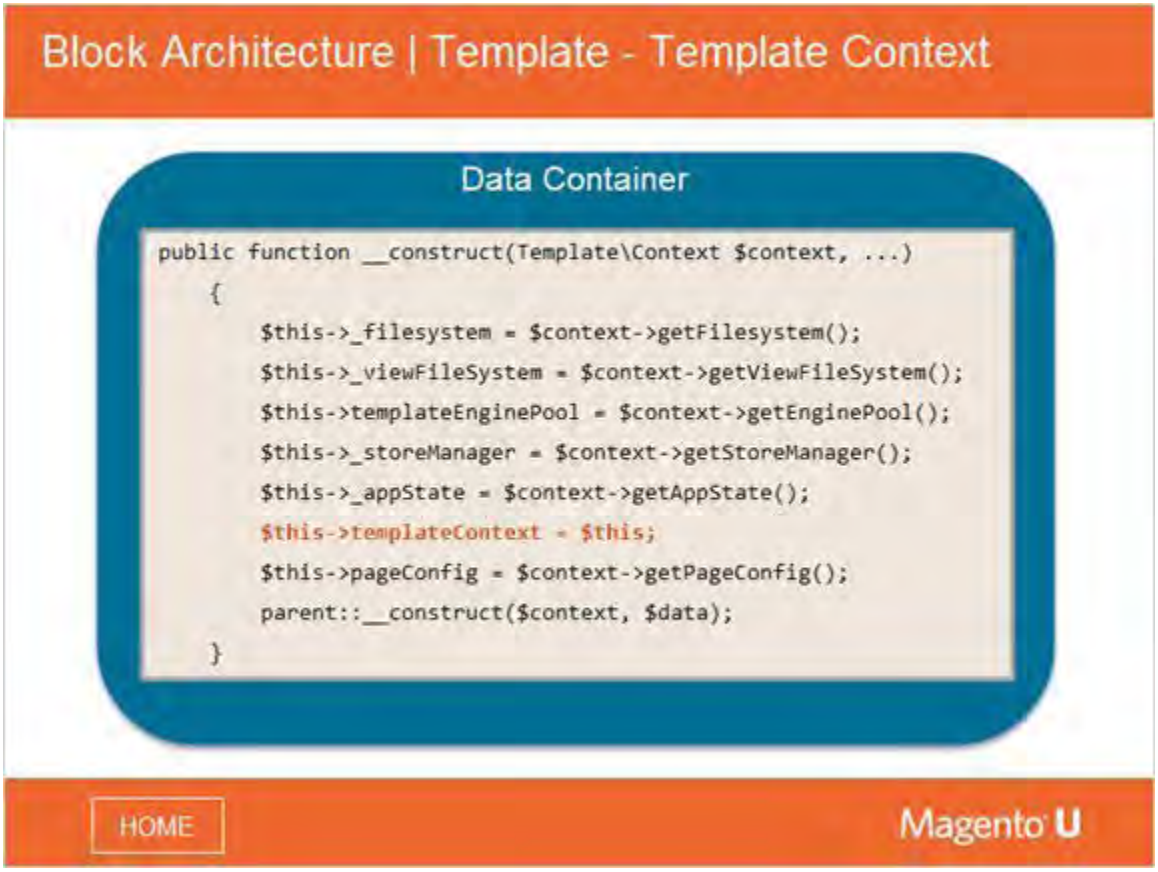


Notes:

The flow for rendering a template goes from `_toHtml()` to `fetchView()`, after which there is a call to `templateEngine::render()`.

Finally, it includes the file (`$file`).

5.20 Block Architecture | Template - Template Context



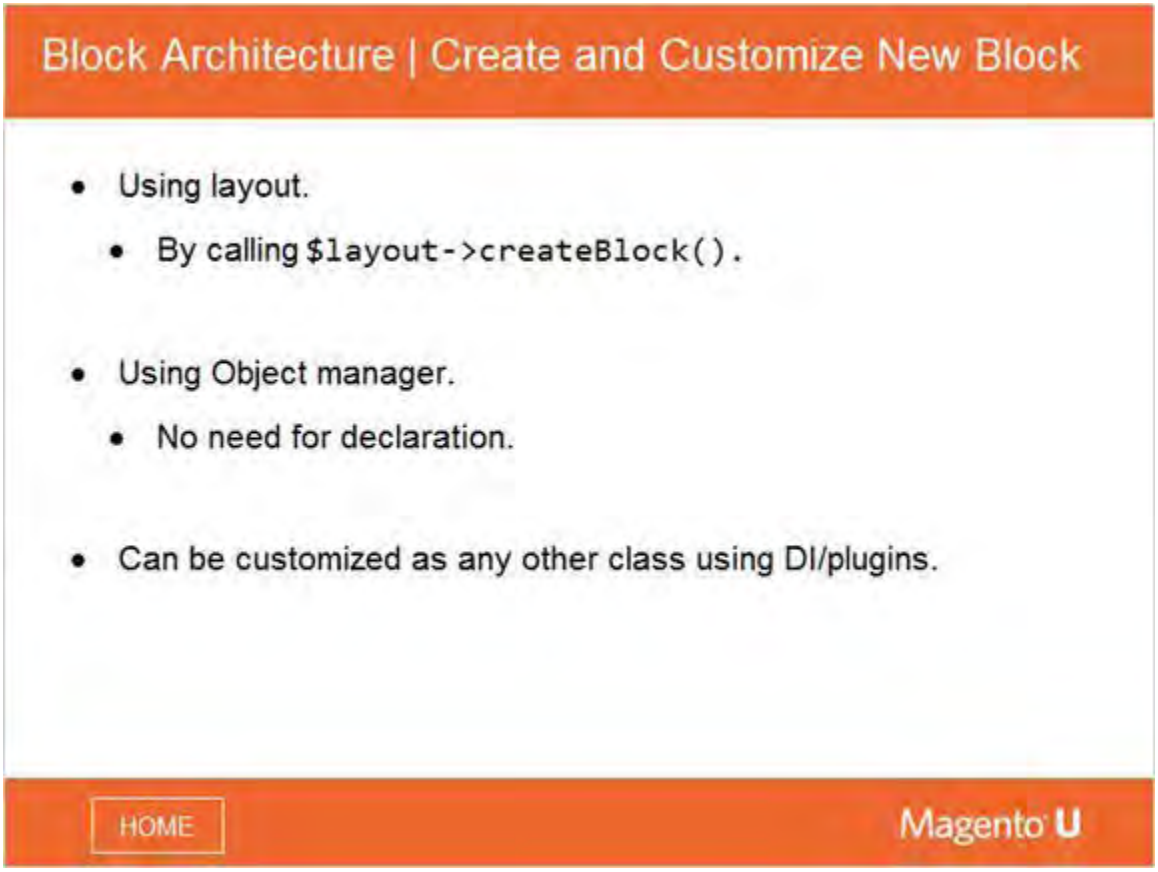
Notes:

Template context is a data container. Earlier, we described a block as a container of data.

All variables from the block are available as a public function in the template. However, it is possible to change a template's context.

The template context shown on this slide is the default code. It will result in an instance of a block but potentially you can change the template context into something else.

5.21 Block Architecture | Create and Customize New Block



Notes:

Creating a new block can be accomplished using layout, more specifically by calling
`$layout->createBlock()`

Note that this is the preferred way. If needed, the layout method will call the object manager.

Also, you can directly use the object manager.

Blocks can be customized as with any other class by using dependency injection and plugins.

5.22 Exercise 3.5.1

Reinforcement Exercise (3.5.1)

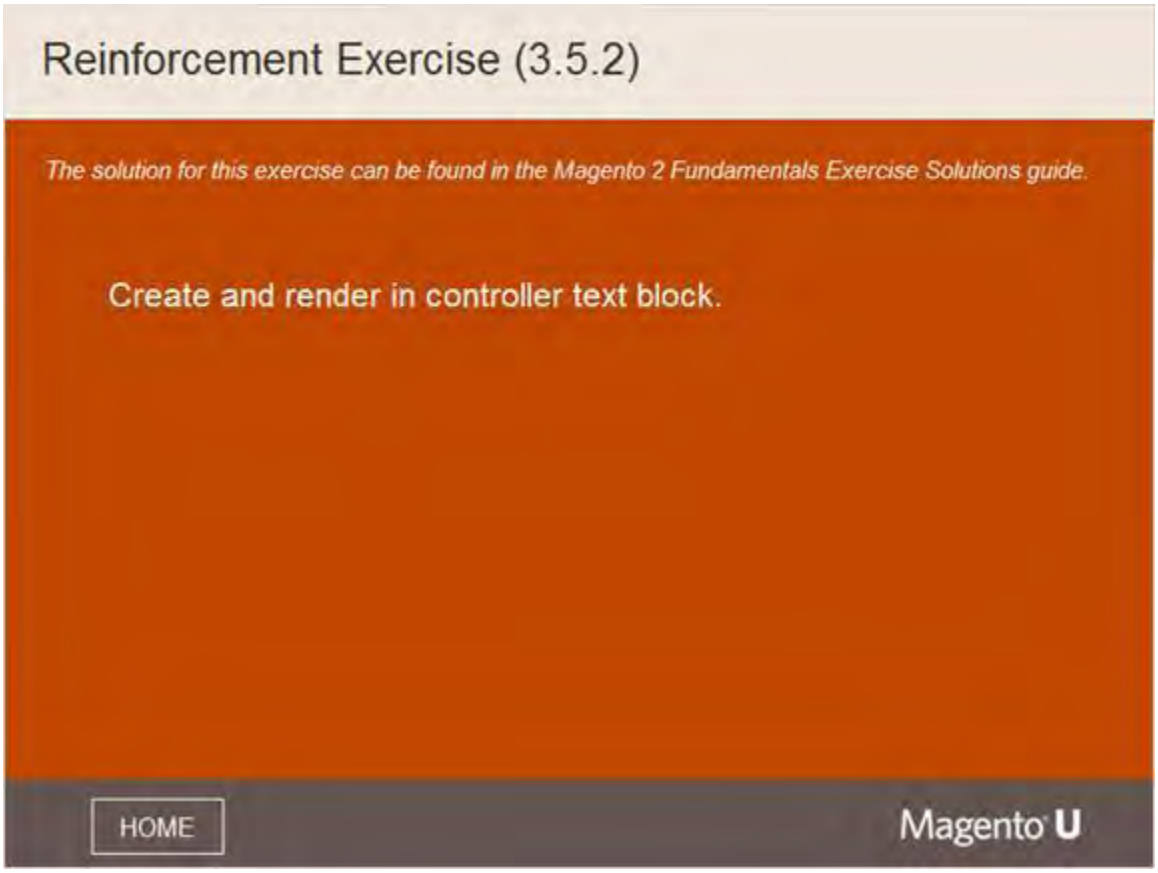
The solution for this exercise can be found in the Magento 2 Fundamentals Exercise Solutions guide.

- Create a block extending `AbstractBlock`.
- Implement the `_toHtml()` method.
- Render that block in the new controller.

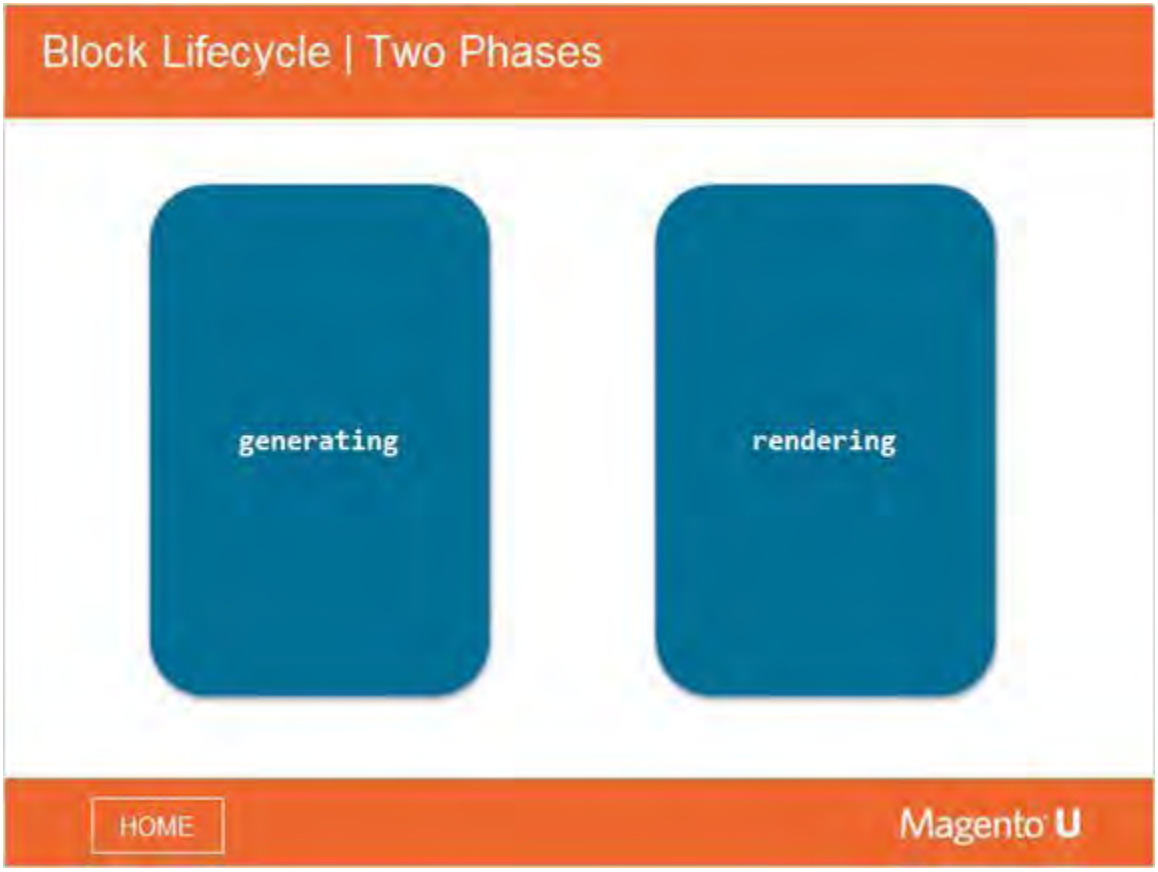
HOME

Magento U

5.23 Exercise 3.5.2



5.24 Block Lifecycle | Two Phases



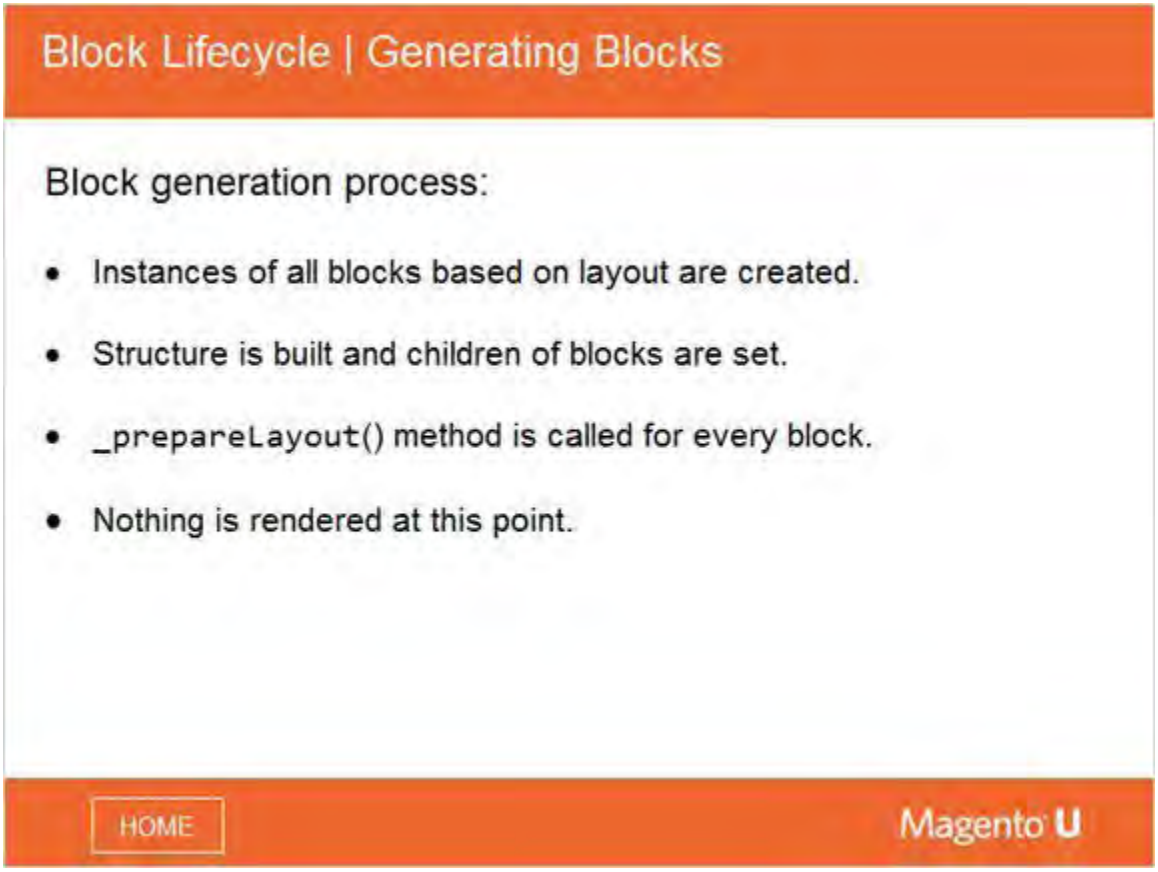
Notes:

We will now switch to discussing block lifecycle. In this topic, we will see how a block is created and then rendered.

In Magento 1, there are two phases of the block lifecycle: generating blocks and rendering blocks.

Magento 2 also follows this two-phase schema of generating and rendering.

5.25 Block Lifecycle | Generating Blocks



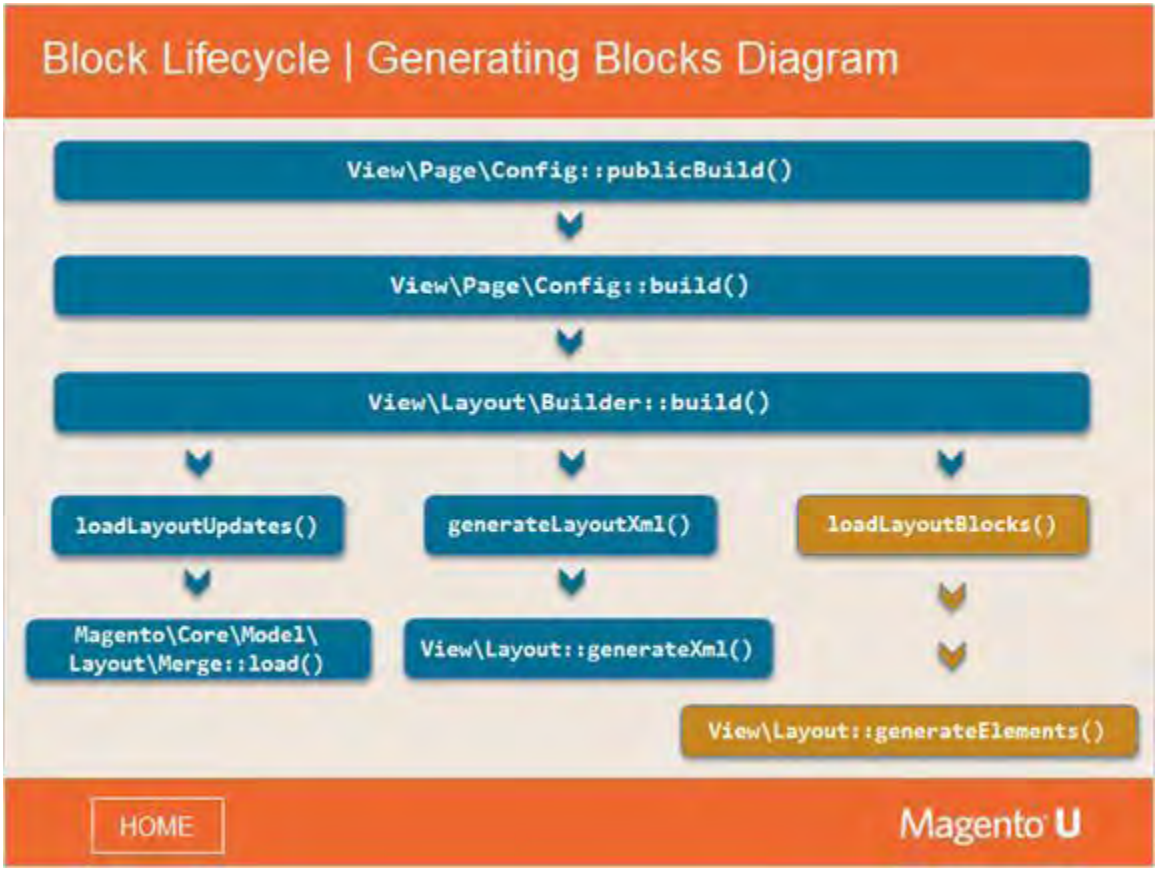
Notes:

Blocks are instantiated at the moment the layout is created. They are not executed at that time, just instantiated.

Also during this phase, the structure is built, the children of blocks are set, and for each block the `prepareLayout()` method is called.

However, nothing is rendered, as that happens in the later rendering phase.

5.26 Block Lifecycle | Generating Blocks Diagram



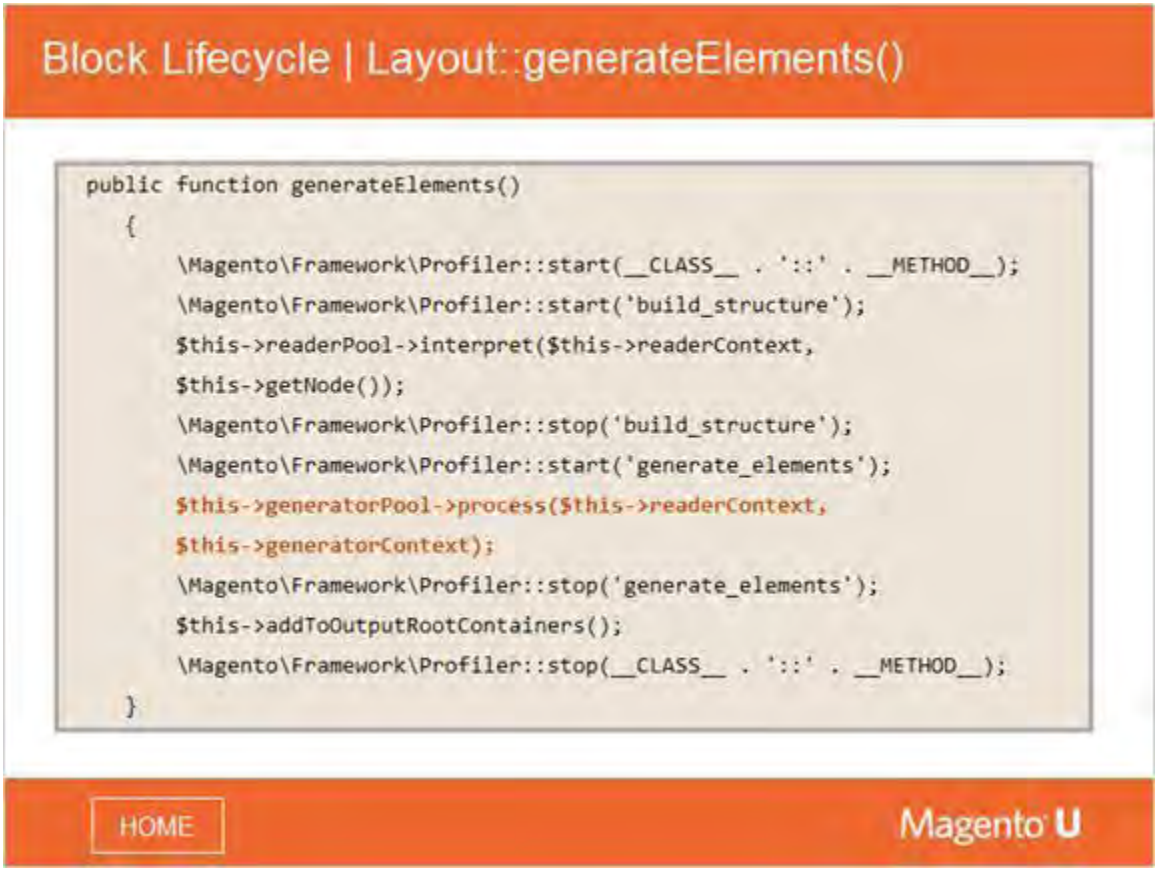
Notes:

You have seen this diagram before in this class; it shows how blocks are generated.

It has three steps, as we have seen before with the build method. The results are `loadlayoutUpdates()`, `generateLayoutXml()`, and `loadLayoutBlocks()`.

In Magento 1, there are similar methods.

5.27 Block Lifecycle | Layout::generateElements()

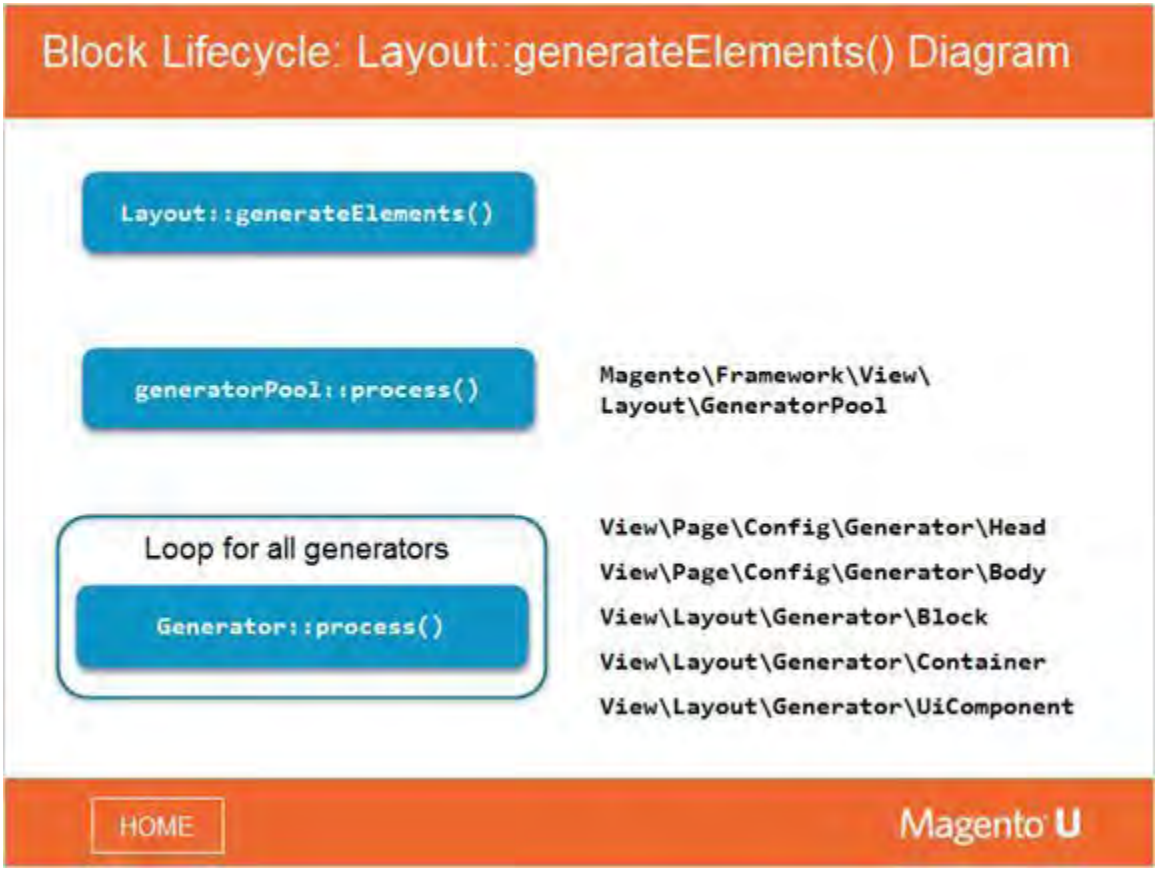


Notes:

Note the line where elements are being generated in the code example:

```
$this->generatePool->process($this->readerContext,
$this->generateContext);
```


5.28 Block Lifecycle: Layout::generateElements() Diagram



Notes:

This slide displays details about `Layout::generateElements()`. `GeneratePool` is an instance of `Magento\Framework\View\Layout\GeneratePool`, and it has generators with the following elements:

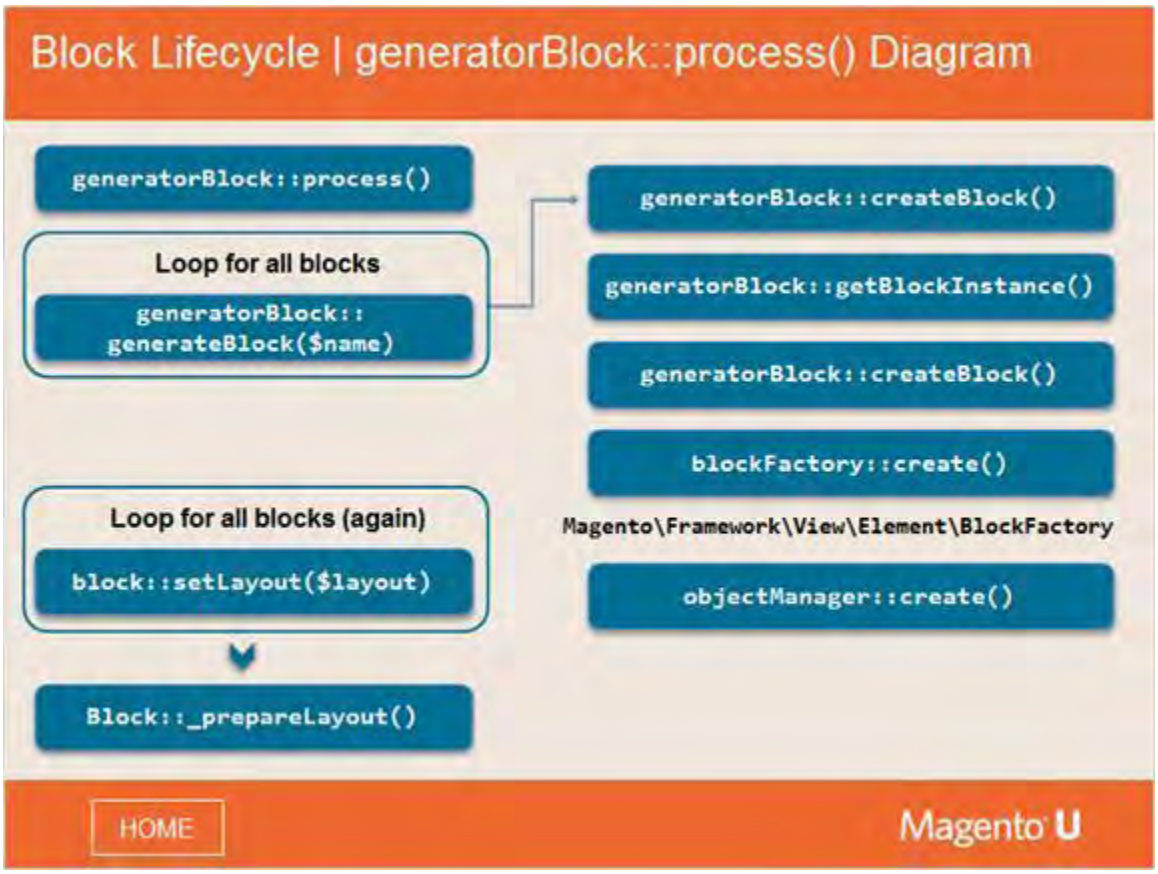
- `View\Page\Config\Generator\Head`
- `View\Page\Config\Generator\Body`
- `View\Layout\Generator\Block`
- `View\Layout\Generator\Container`
- `View\Layout\Generator\UiComponent`

Reference: Locate this file in your Magento installation.

`/lib/internal/Magento/Framework/View/Layout/GeneratorPool.php`

Simply put, this file goes through all the generators. It is worth taking a look at the `GeneratorInterface` to understand what it does. It has two methods: `process() and getType()`.

5.29 Block Lifecycle | generatorBlock::process() Diagram



Notes:

The diagram demonstrates how the `process()` method of the `generatorBlock` class works.

First, the `generatorBlock` object goes through all the block declarations and creates an instance of every block by calling the `generateBlock()` method.
The right side of the diagram shows you how that works. You should understand, at least at a high level, how classes are being instantiated, as illustrated on this slide.

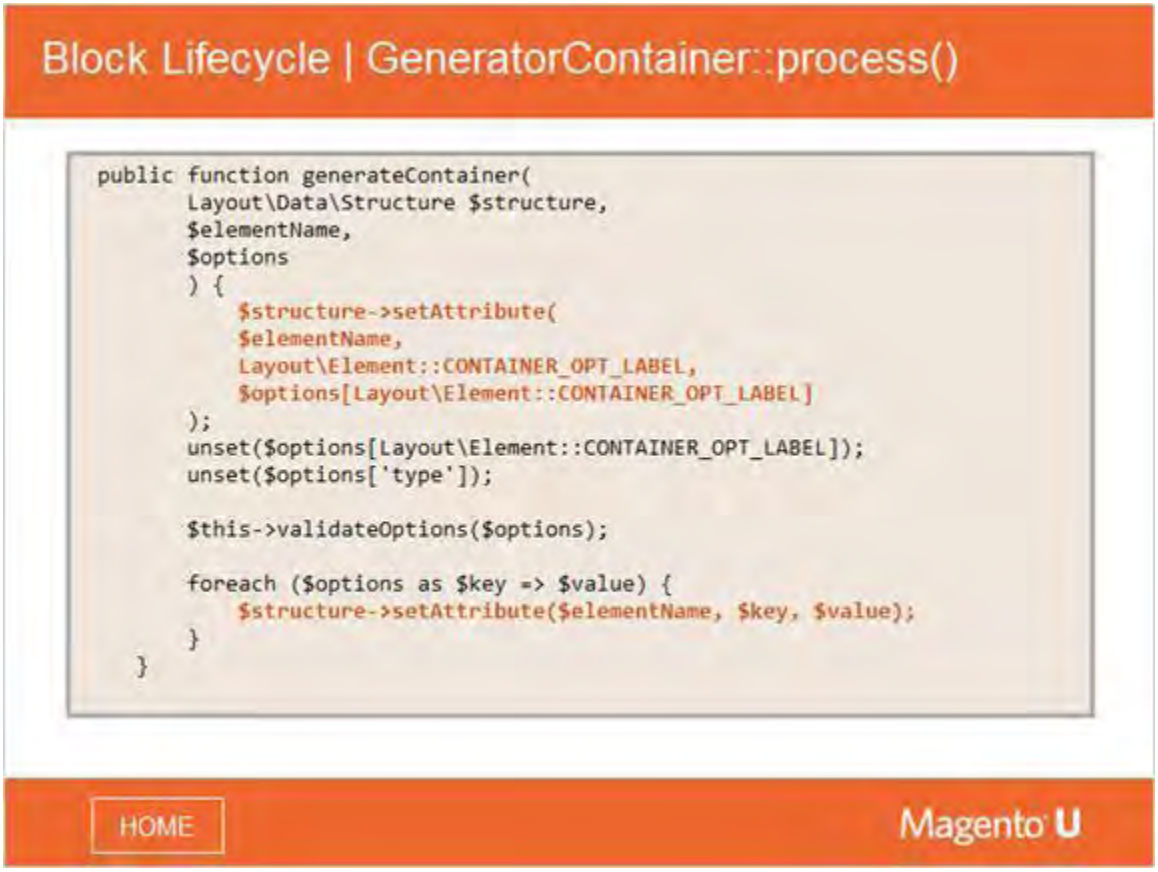
There are five intermediate calls before the execution flow goes to the `objectManager`, which creates all the objects in Magento.
Next, the `generatorBlock` object goes through all generated blocks again and performs a call to the `setLayout()` method, which calls `_prepareLayout()`.

It is very important to understand that the `_prepareLayout()` method is called at the moment of block instantiation. There are many (probably most) classes in Magento that have `_prepareLayout()` overridden. Usually there is some `configuration/initialization` code located in this method. We will see more examples of this code later.

5.30 Code Demonstration | _generatorBlock

Notes:

5.31 Block Lifecycle | GeneratorContainer::process()



Notes:

This slide shows how containers are generated.

Since a container is not a physical class, there is very little to generate, which corresponds to what we're seeing in the code snippet.

The `process()` method of a `GeneratorContainer` only manages attributes.

5.32 Block Lifecycle | GeneratorUiComponent::generateComponent()

Block Lifecycle | GeneratorUiComponent::generateComponent()

```
protected function generateComponent(Layout\Data\Structure $structure,
                                     $elementName, $data)
   {
       $attributes = $data['attributes'];
       if (!empty($attributes['group'])) {
           $structure->addToParentGroup($elementName,
           $attributes['group']);
       }
       $arguments = empty($data['arguments']) ? [] :
           $this->evaluateArguments($data['arguments']);
       $componentName = isset($attributes['component']) ?
                                $attributes['component'] : '';
       $uiComponent = $this->uiComponentFactory
                           ->createUiComponent
                            ($componentName, $elementName, $arguments);
       return $uiComponent;
   }
```

HOME

Magento U

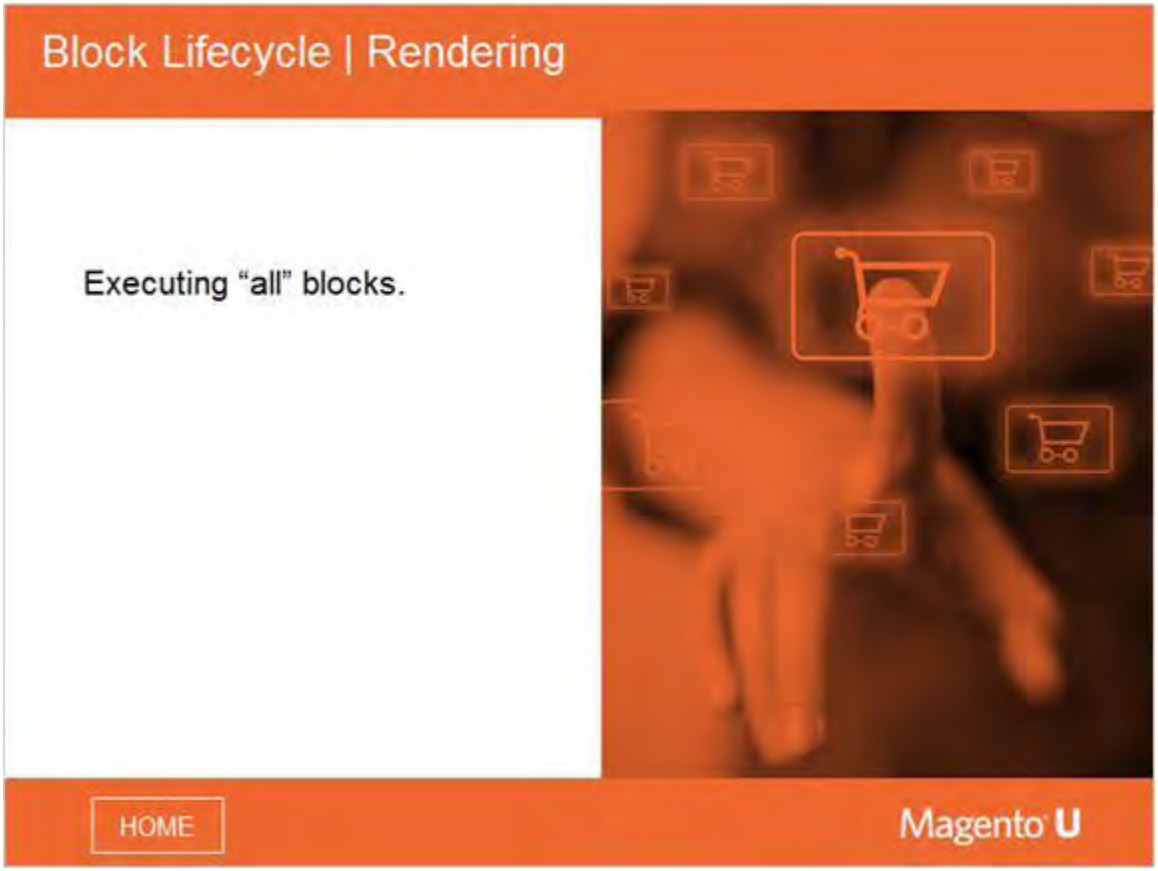
Notes:

This code shows how the `UiComponent` is generated. `UiComponent` is a class, and often a block, which is why it has to be instantiated.

Highlighted lines show what is happening in the `uiComponentFactory` object (an instance of the `Magento\Framework\View\Element\UiComponentFactory`).

At the moment of this class' creation, that functionality hasn't been totally finished.

5.33 Block Lifecycle | Rendering

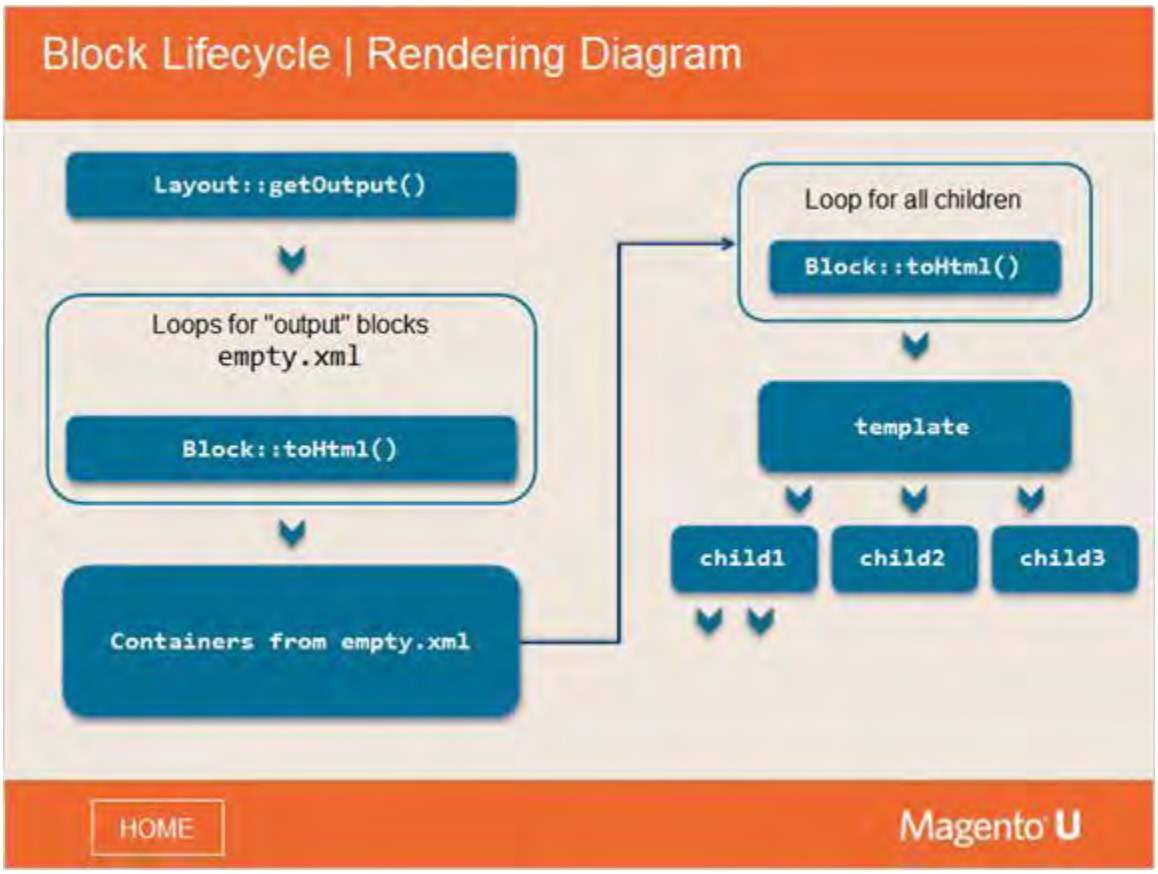


Notes:

We now switch to the second phase of the block lifecycle, block rendering. By rendering, we mean the process of generating a block’s HTML.

Usually it comes from a template, but as we’ve seen before, other options are also possible.

5.34 Block Lifecycle | Rendering Diagram



Notes:

This diagram shows the complete rendering process.

First, it starts with the `getOutput()` call to the layout class. At this moment, we already have all the layout xml for the page generated, and all the block classes are created.

Next, Magento renders all the containers from `empty.xml`. Each container has blocks assigned to it by other layout updates. By rendering containers, all the blocks are rendered by calling their `toHtml()` method.

The `toHtml()` method of a template block includes a template file that will call back the block for data it might need. Also, the template may have `<?php $this->getChildHtml(..) ?>` calls, which will render a block's child, and each child will render its children in the same way. So this is a recursive process.

In summary, the process starts with `Layout::getOutput()`, so that all the containers from `empty.xml` will be rendered, which then causes all blocks assigned to containers to be rendered, which starts a recursive process of rendering templates (`.phtml` files).

5.35 Exercise 3.5.3

Reinforcement Exercise (3.5.3)

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

- Customize the `Catalog\Product\View\Description` block... *then*
- Implement the `_beforeToHtml()` method ... *then*
- Set a custom description for the product here.

HOME

Magento U

6. Templates

6.1 Templates

Notes:

This module discusses templates in more depth.

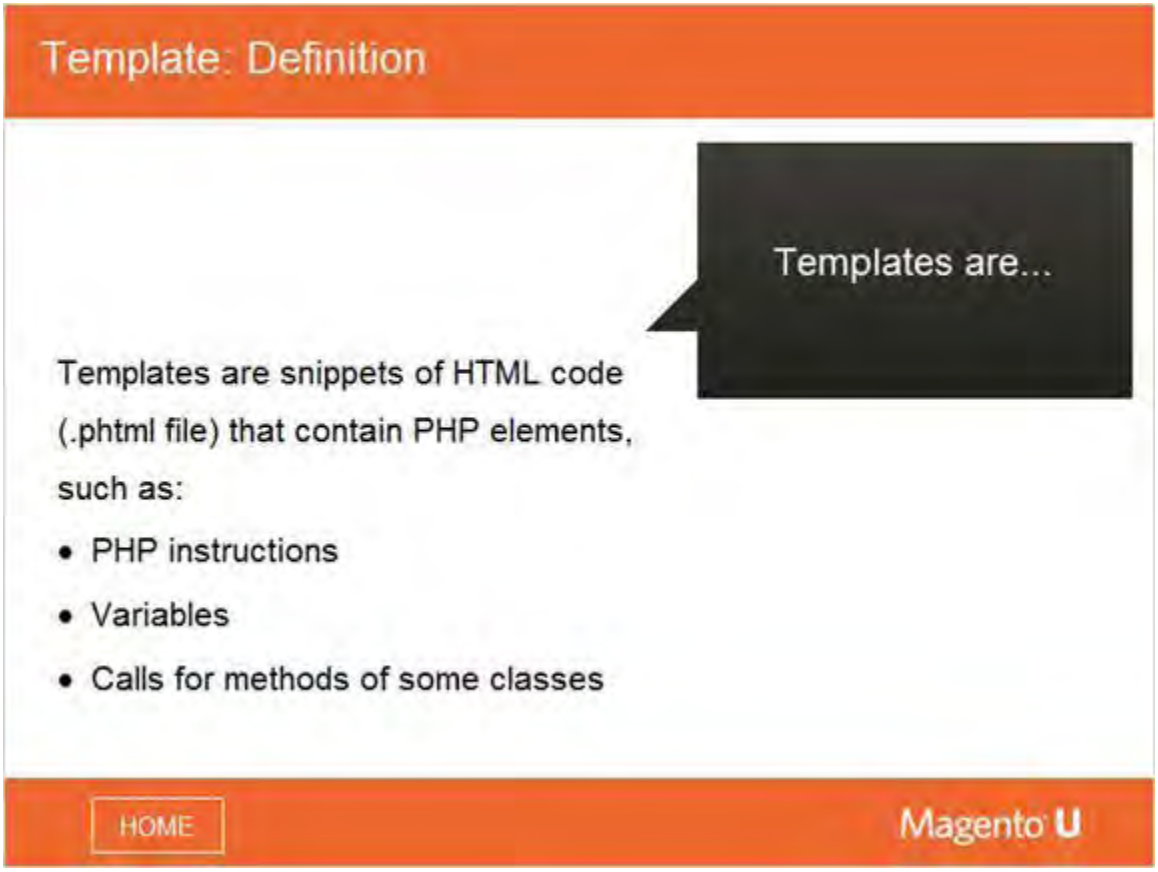
6.2 Module Topics

Notes:

In this module we will discuss:

- Template location
- Template variables
- Template rendering
- Fallback
- Customization

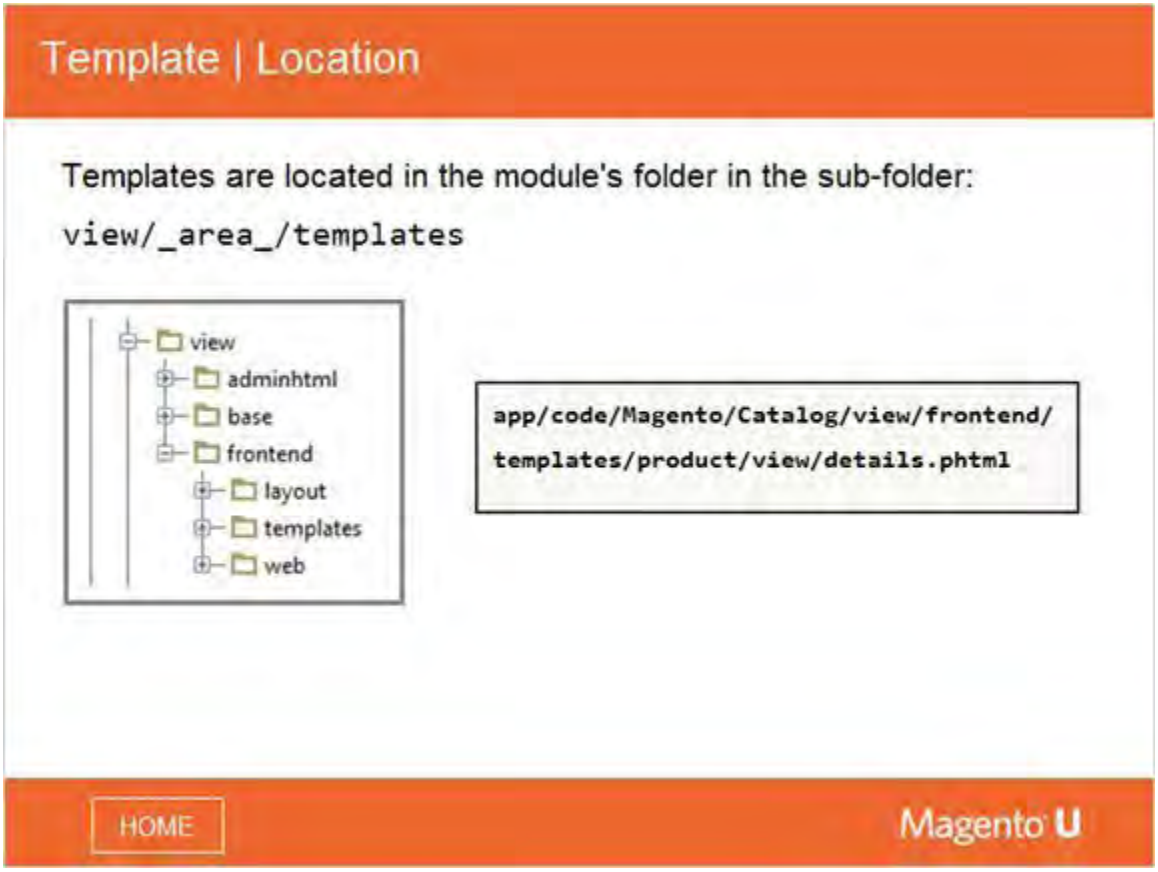
6.3 Template: Definition



Notes:

Templates are snippets of `phtml` code that contain PHP elements, such as PHP instructions, variables, and calls for class methods.

6.4 Template: Location



Notes:

Magento 2 uses phtml files as templates (just as in Magento 1).

In Magento 2, templates are located in the module, more exactly in the module sub-folder

`view/_area_/templates`. (In Magento 1, they were located in `app/design`).

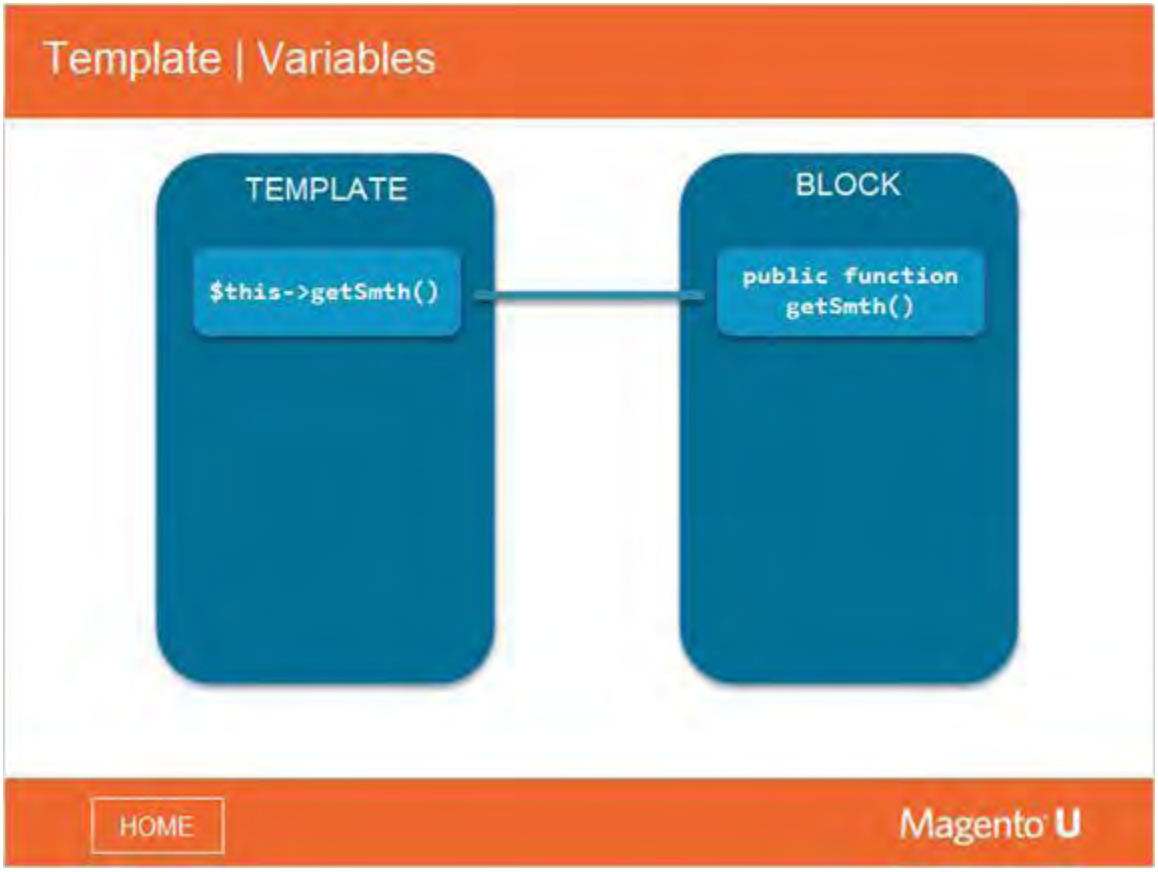
As templates can belong to different modules, you should prepend the template with the module name as a best practice. This will help you to locate template files and avoid file conflicts.

You type the name of the module, double single colon, and then the name.

Ex: `catalog/product/view/price.phtml` becomes

```
Magento_Catalog::catalog/product/view/price.phtml
```


6.5 Template | Variables



Notes:

Templates are fragments of html code with PHP elements. We could have PHP variables, PHP instructions and calls for the methods of some classes.

Potentially, we can define other types of templates. Templates will render a piece of html with the data taken from the HTML block.

The difference in Magento 2 vs. Magento 1 is that this is not really a block but a template engine.

How does it work? The diagram schematic provides an overview.

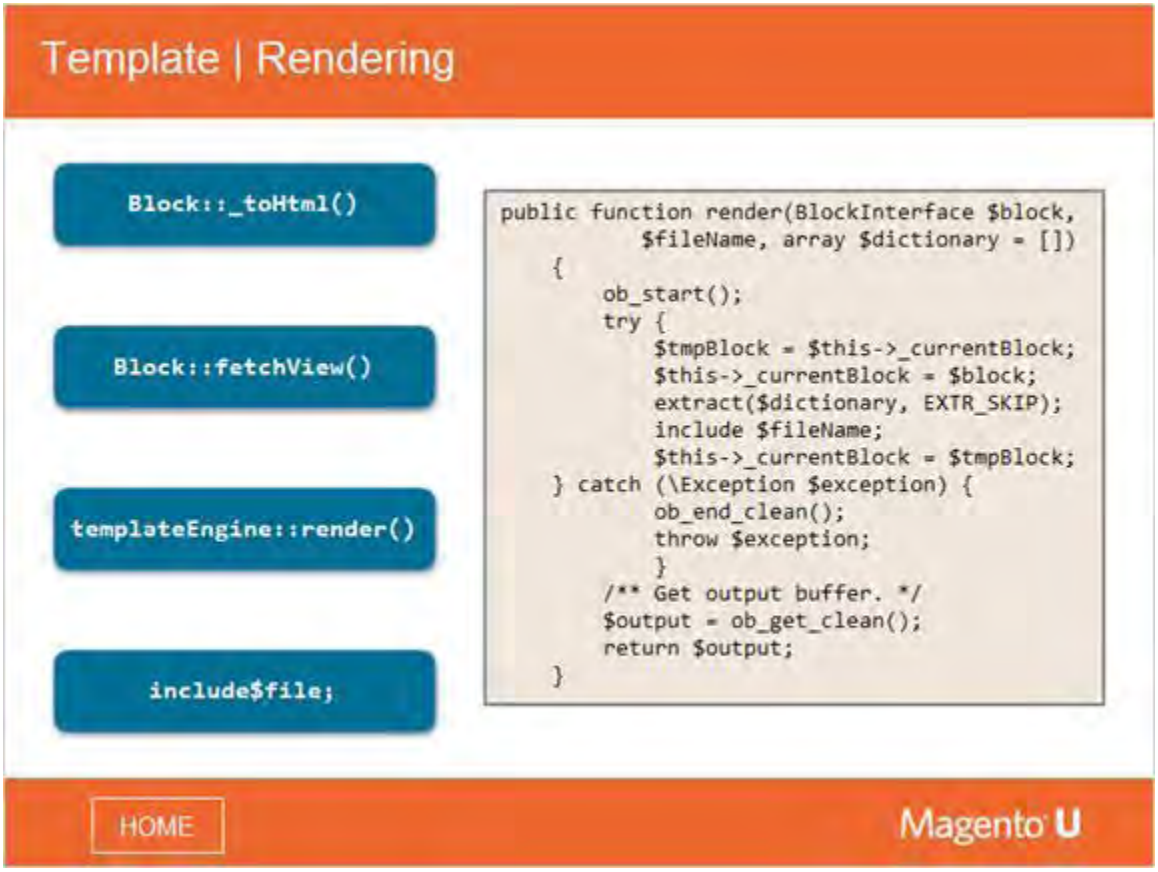
In a template, there is often a call that refers to the block and performs the call to the `getSmth()` method.

The `getSmth()` function could be public, private or protected in Magento 1, but a Magento 2 template can only call a public function.

```
$this->getSmth()      [this refers to the block; all templates are connected to blocks]
      ->public function getSmth()
```

We will see why it works this way in the next two slides.

6.6 Template: Rendering



Notes:

The diagram and code presents an example of how a template is rendered.

A template is rendered when `Block::_toHtml()` goes to `fetchView()`, then to `templateEngine::render()` to include the filename.

The code example shows the function render, which includes the filename inside the template engine.

6.7 Template | Rendering Engine __call()

Template | Rendering Engine __call()

Magento\Framework\View\TemplateEngine\Php

```
public function __call($method, $args)
{
    return call_user_func_array([$this->_currentBlock,
                                 $method], $args);
}
```

HOME

Magento U

Notes:

The template file will be included inside the template engine, and references the `TemplateEngine\Php (or Magento\Framework\View\TemplateEngine\Php)`.

This will forward every call from the template engine to the block, as shown on the previous diagram.

6.8 Template | Example

Template | Example

```
<?php if ($detailedInfoGroup = $this->getGroupChildNames('detailed_info',
'getChildHtml')):?>
    <div class="product info detailed">
        <?php $layout = $this->getLayout(); ?>
        <div role="tablist" class="product data items"
                data-mage-init='{"tabs":{"openedState":"active"}}'>
            <?php foreach ($detailedInfoGroup as $name):?>
                <?php
                    $html = $layout->renderElement($name);
                    if (!trim($html)) {
                        continue;
                    }
                    $alias = $layout->getElementAlias($name);
                    $label = $this->getChildData($alias, 'title');
                ?>
                …
            <?php endforeach;?>
        </div>
    </div>
<?php endif; ?>
```

HOME

Magento U

Notes:

This code is an example of a template that contains a mix of PHP and HTML, with calls to the block:

```
<?php $layout = $this->getLayout(); ?>
```


6.9 Fallback: Definition

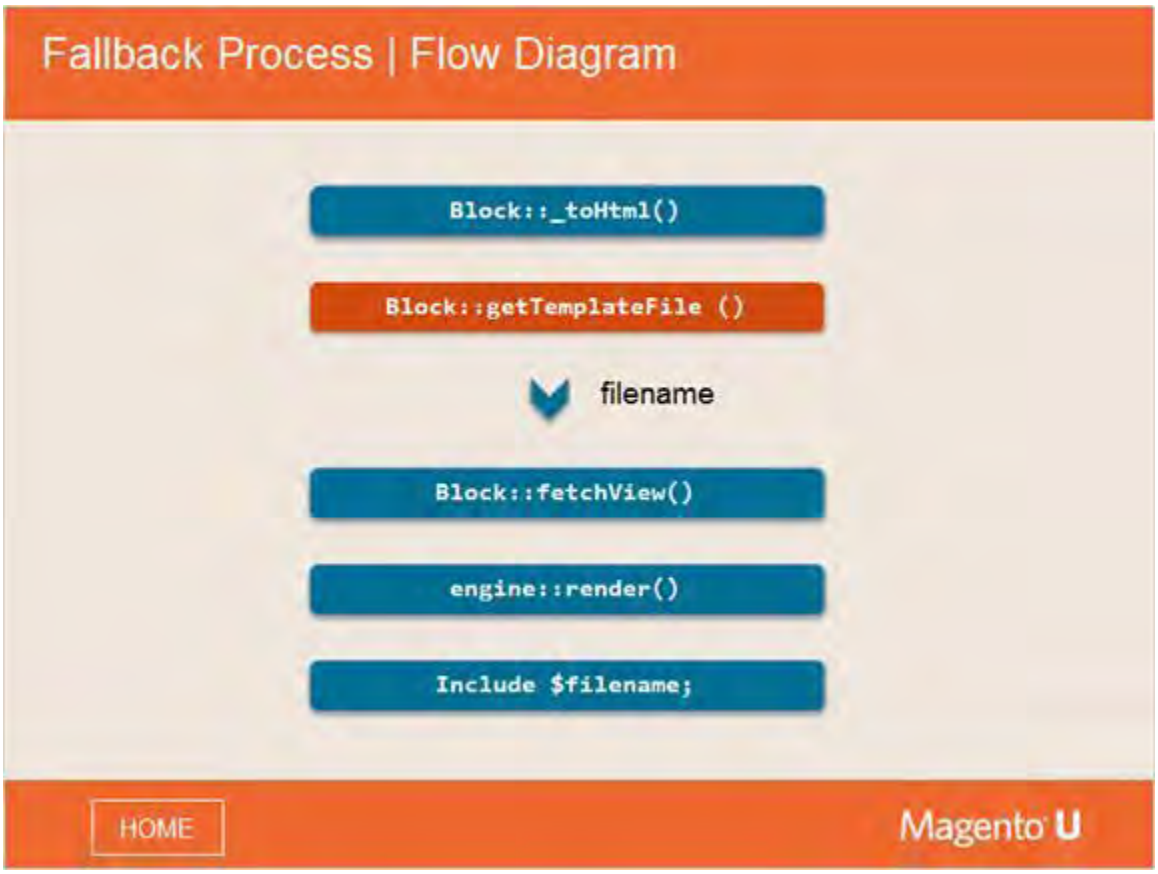
Notes:

When creating a template, the filename might be incomplete. Fallback resolves the file path, defining the opening parameters of the complete filename so the file can be found in the system when called or referenced.

Example: `product/view/details.phtml` falls back to `Magento_Catalog/view/frontend/templates/product/view/details.phtml.`

Fallback is useful in debugging, as going back through the fallback steps lets you know exactly what file is being rendered. It also facilitates customizing templates.

6.10 Fallback Process | Flow Diagram



Notes:

In Magento 1, the fallback process is concentrated in the design package. The `Mage_Core_Model_Design_Package::getFilename()` method makes a series of nested calls, which makes it difficult to debug.

In Magento 2, the process is simpler. Magento 2 instead uses `getTemplateFile()`, which then goes to the `Block::fetchView()`, the `engine::render()`, and includes the filename ($filename).

`Block::getTemplateFile()` is the critical method in this process.

6.11 Fallback | Template::getTemplateFile()

Fallback | Template::getTemplateFile()

```
public function getTemplateFile($template = null)
{
        $params = ['module' => $this->getModuleName()];
        $area = $this->getArea();
        if ($area) {
            $params['area'] = $area;
        }
        $templateName = $this->_viewFileSystem
                            ->getTemplateFileName($template ?:
                        $this->getTemplate(), $params);
        return $templateName;
}
```

HOME

Magento U

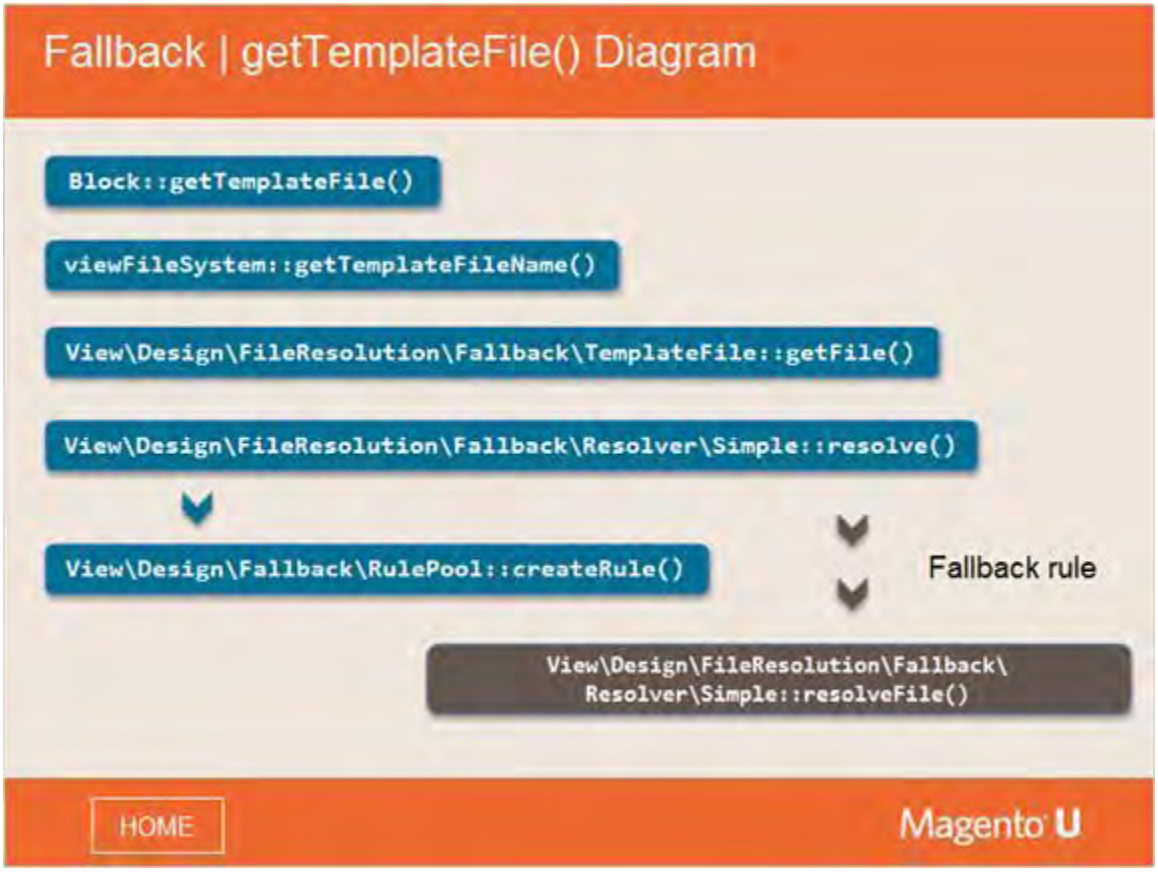
Notes:

This example displays the code for the method `getTemplateFile()`.
Note: `_viewFileSystem` returns the template name.

In this code example, the most important code lines are those where the fallback process happens. These are highlighted in orange.

Reference: Locate the following file in your Magento installation:
`Magento/Framework/View/Filesystem.php`

6.12 Fallback | getTemplateFile() Diagram



Notes:

This diagram show files involved in the fallback process.

The rule created via the `RulePool` detects where the file is located.

6.13 Code Demonstration | getTemplateFile() Diagram

Notes:

6.14 Fallback | createTemplateFileRule()

Fallback | createTemplateFileRule()

```
protected function createTemplateFileRule()
{
     $themesDir = $this->filesystem->getDirectoryRead
                  (DirectoryList::THEMES)->getAbsolutePath();
     $modulesDir = $this->filesystem->getDirectoryRead
                   (DirectoryList::MODULES)->getAbsolutePath();
       return new ModularSwitch(
           new Theme(
           new Simple("$themesDir/<area>/<theme_path>/templates")
           ),
           new Composite(
            [
 new Theme(new Simple("$themesDir/<area>/<theme_path>/
                                  <namespace>_<module>/templates")),
 new Simple("$modulesDir/<namespace>/<module>/view/<area>/templates"),
 new Simple("$modulesDir/<namespace>/<module>/view/base/templates"),
            ]
           )
      );
 }
```

HOME

Magento U

Notes:

The code example shows the creation of a template file rule. Magento will check for the filename in these folders.

6.15 Fallback | Resolver::resolve() Method

Fallback | Resolver::resolve() Method

```
protected function createTemplateFileRule()
{
     $themesDir = $this->filesystem->getDirectoryRead
                  (DirectoryList::THEMES)->getAbsolutePath();
     $modulesDir = $this->filesystem->getDirectoryRead
                  (DirectoryList::MODULES)->getAbsolutePath();
       return new ModularSwitch(
           new Theme(
           new Simple("$themesDir/<area>/<theme_path>/templates")
           ),
           new Composite(
            [
new Theme(new Simple("$themesDir/<area>/<theme_path>/
                     <namespace>_<module>/templates")),
new Simple("$modulesDir/<namespace>/<module>/view/<area>/templates"),
new Simple("$modulesDir/<namespace>/<module>/view/base/templates"),
             ]
            )
      );
}
```

HOME

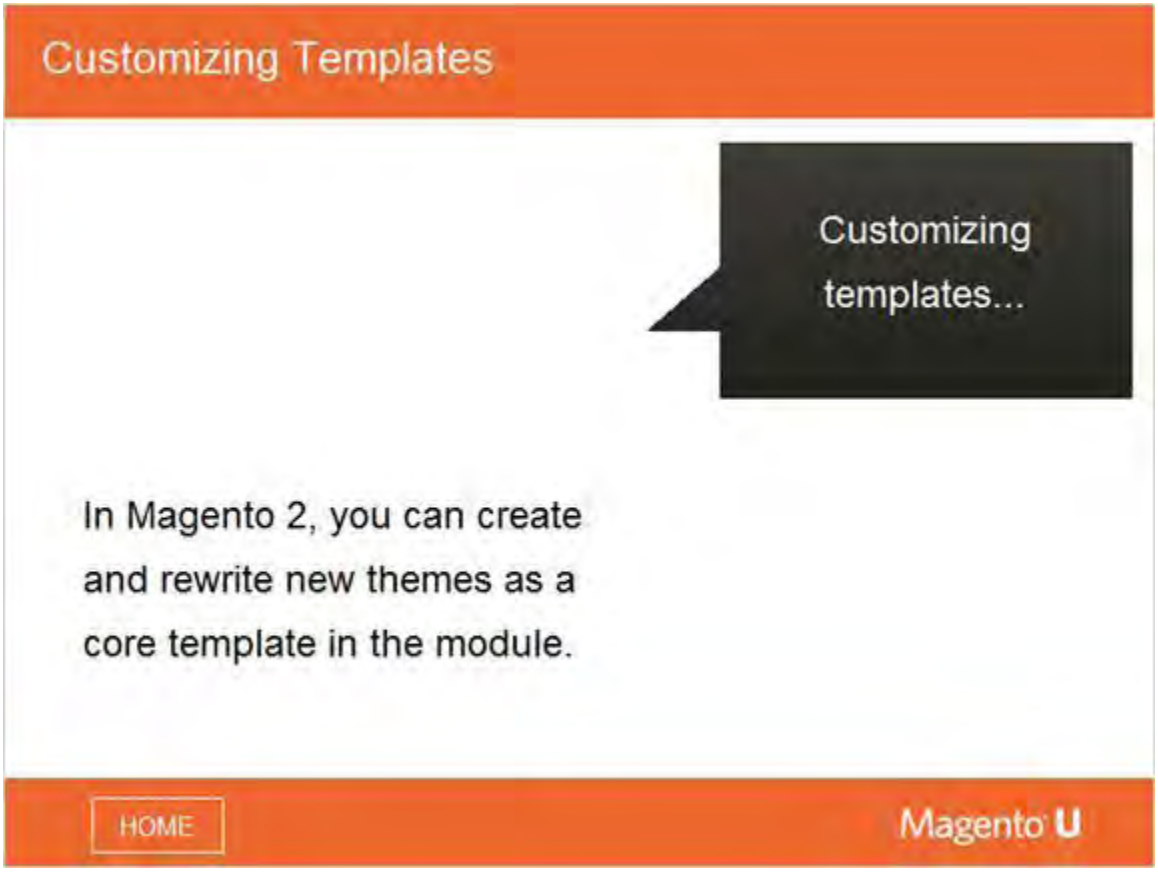
Magento U

Notes:

This code example shows the `Resolver::resolve()` method, which checks whether a filename exists in these folders, resolving the path to the template file.

The full path facilitates debugging and helps in the process of customizing templates.

6.16 Customizing Templates



Notes:

In Magento 2, you can create and rewrite new themes as a core template in the module.

6.17 Customizing Templates | Rewrite Core Template

Notes:

There are three general steps in rewriting the core template:

1. Create your module
2. Create a new template in your module
3. Set your template to the block that contains the core template to rewrite.

6.18 Exercise 3.6.1

Reinforcement Exercise (3.6.1)

The solution for this exercise can be found in the Magento 2 Fundamentals Exercise Solutions guide.

Define which template is used in:

```
Catalog\Block\Product\View\Attributes
```

HOME

Magento U

6.19 Exercise 3.6.2

Reinforcement Exercise (3.6.2)

The solution for this exercise can be found in the Magento 2 Fundamentals Exercise Solutions guide.

- Create a template block.
- Create a custom template file for it.
- Render the block in the controller.

HOME

Magento U

6.20 Exercise 3.6.3

Reinforcement Exercise (3.6.3)

The solution for this exercise can be found in the Magento 2 Fundamentals Exercise Solutions guide.

Customize the `Catalog\Block\Product\View\Description` block, and assign a custom template to it.

HOME

Magento U

7. Layout XML Structure

7.1 Layout XML Structure

Notes:

Layout XML is a tool to build the pages of the Magento application in a modular and flexible manner.

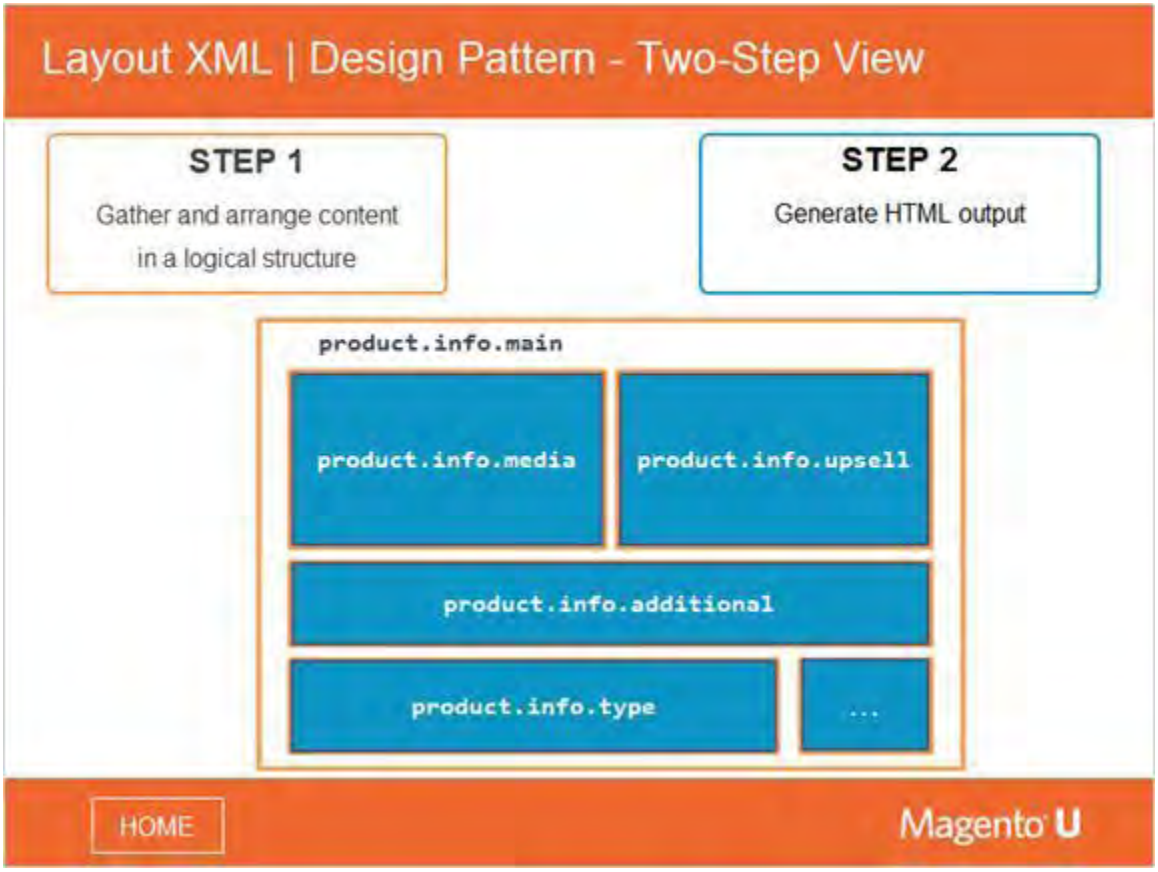
It allows frontend developers to describe the page layout and content placement, without regard to how each rendered content part will look.

7.2 Module Topics

Notes:

In this module, we will discuss: design patterns; page sections; containers, blocks, and layout files; merging files; directories and schemas.

7.3 Layout XML | Design Pattern - Two-Step View



Notes:

There are two main steps in working with the layout XML, as depicted in this two-step view:

1. Create the page content structure.
2. Render the structure.

The whole idea of the layout XML is to define the logical page structure.

The first step is to create a logical page content structure, and the second step is to generate the HTML output.

There are a few changes in Magento 2. The first change is that each update handle resides in its own file.

In addition, it is now a little more expressive and explicit. Each layout XML file is validated by the schema file, which should make xml syntax issues much easier to identify. This course focuses primarily on Step 1.

Note that a module can add content to existing pages without changing files in another module.

It can also move or remove existing content added by other modules. Once all the layout XML instructions from all the modules have been processed, the result is rendered and sent to the browser for display.

7.4 Layout XML Structure: Page Sections

Notes:

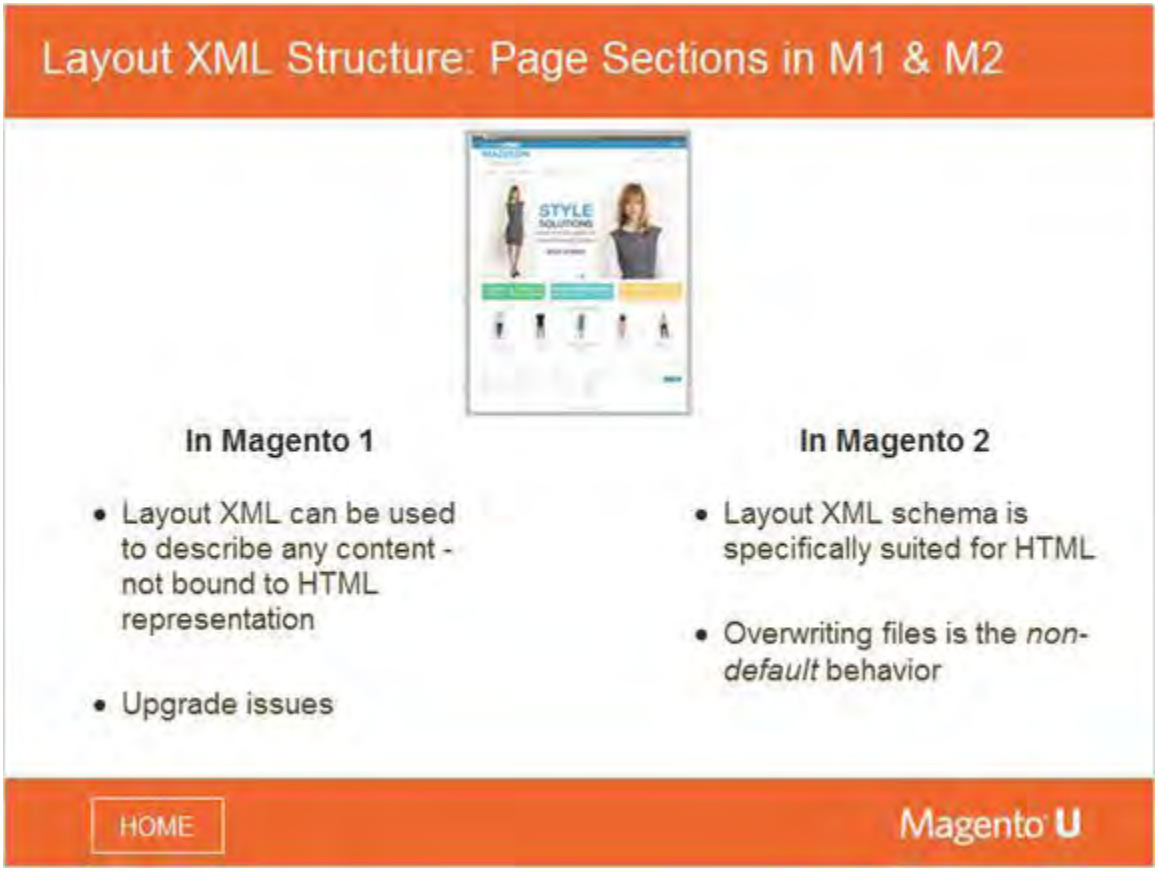
In Magento 2, the page has been split it into different parts:

- head
- body (*Most of the content visible to visitors is defined within this section)*
- html
- update

The first three Layout XML sections correspond with their html page tag equivalents. Each of Magento's Layout XML sections is used to manipulate the corresponding part of the html page.

Update is not really a section; it is used to include more page processing instructions in separate files, helping to avoid code duplication.

7.5 Layout XML Structure: Page Sections



Notes:

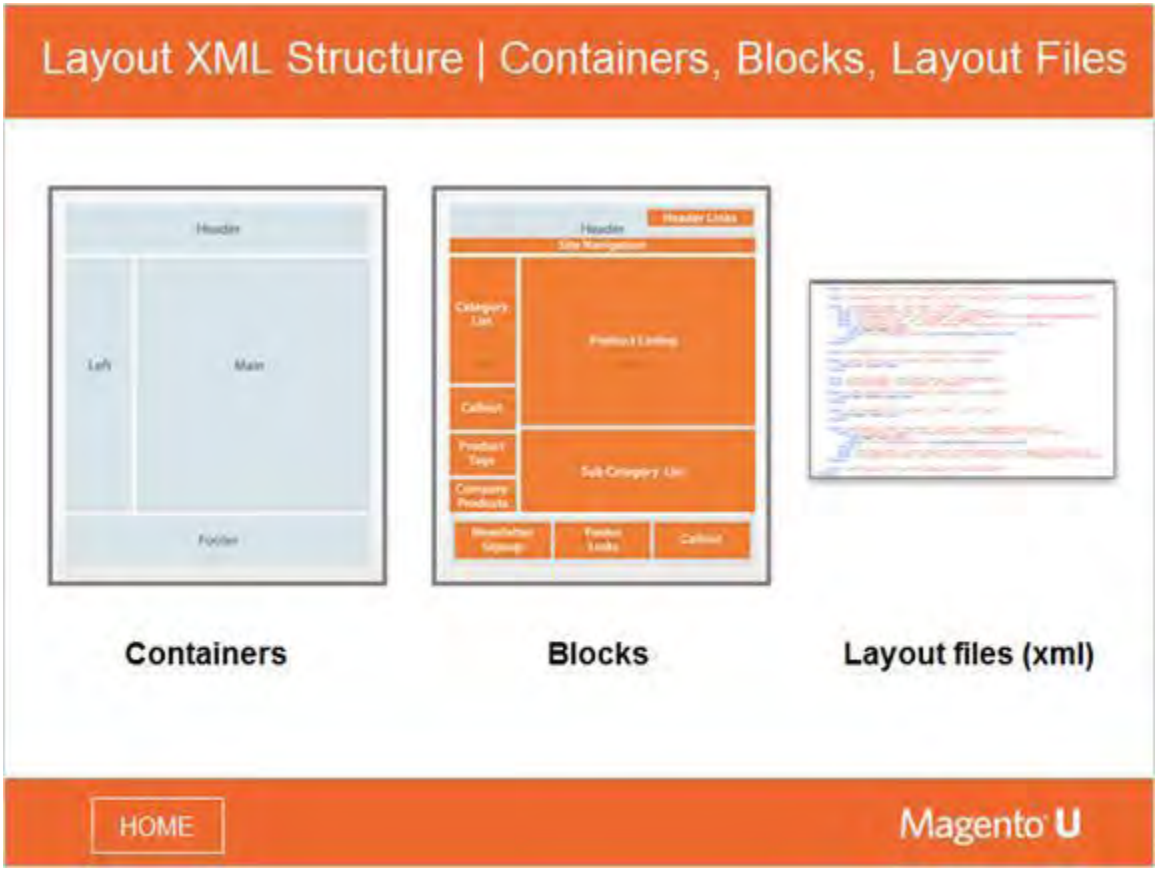
In Magento 1, the page structure is very generic.

Layout XML can be used to describe any kind of content -- it is not inherently bound to an HTML representation. You have a layout XML file and the theme fallback. If you create the same layout XML file earlier in the theme fallback, then it would not be inherently bound to an HTML representation. Usually, developers just copy and modify whatever is needed in a new version of the file. The big issue is that upgrades need to be made in the new version of the file, and this can represent a lot of work

In Magento 2, the Layout XML schema has been modified so it is suited for HTML specifically.

The default behavior with inheritance fallback means that it collects all the XML files and then merges them together. Note that it is still possible to overwrite files but that is now the *non-default* behavior - you have to use a special overwriting directory. It is not a recommended practice because that will introduce the same upgrade issues as with Magento 1.

7.6 Containers, Blocks, Layout Files



Notes:

This slide should look familiar, as you saw it in an earlier section.

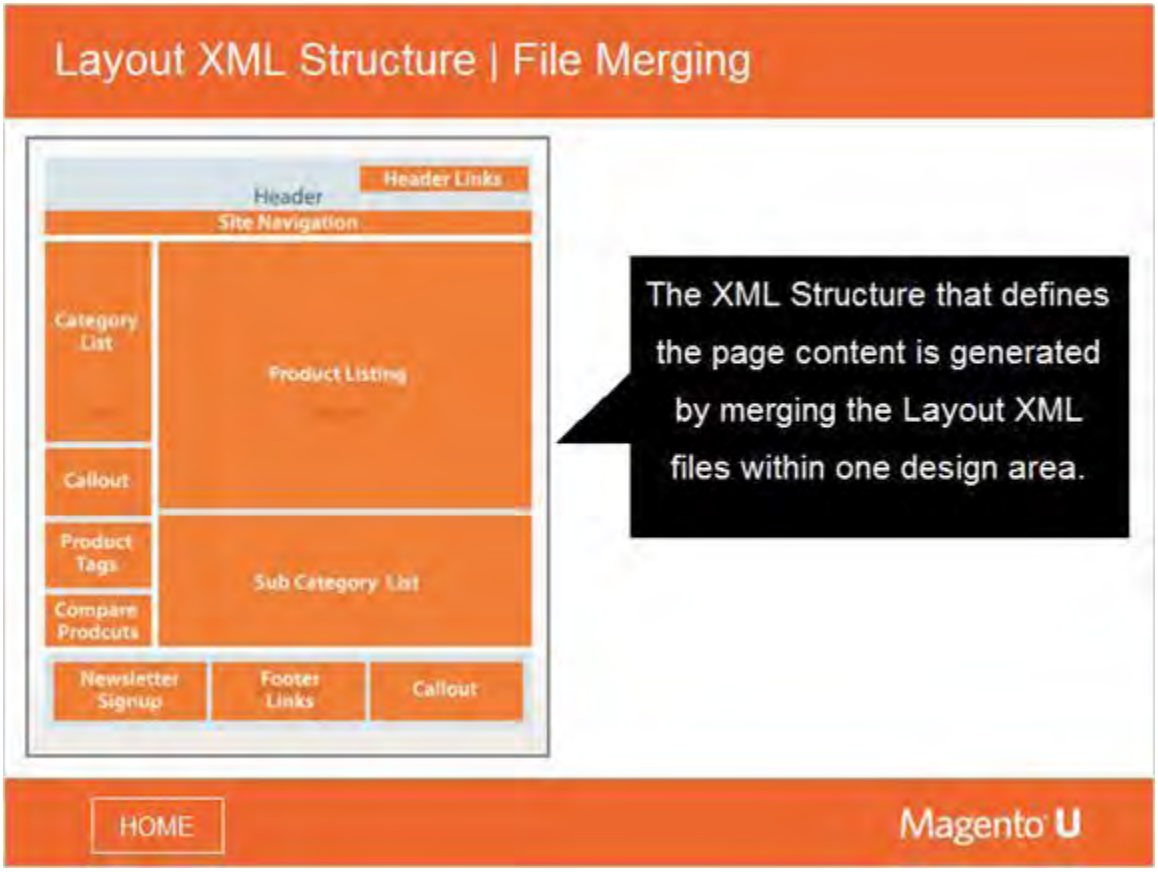
Recall that the layout of a page is determined by containers, which act as a framework and do not contain actual content, and that the content segments within a container are called blocks. Examples for content blocks might be a filter navigation, a page title or a product add-to-cart form.

Each web page within Magento is a hierarchy of containers comprised of blocks that, in turn, can contain any number of child content blocks or child containers. These provide handy extension points to other modules that place content on the page in the same area.

For this reason, the hierarchical page structure is sometimes also called a page tree, with the root referring to the parent containers, and the child block and container elements as the branches. The leaves would refer to content blocks that don't have any further children.

Blocks can also contain child containers, to provide extension points to other modules that place content on the page in that area.

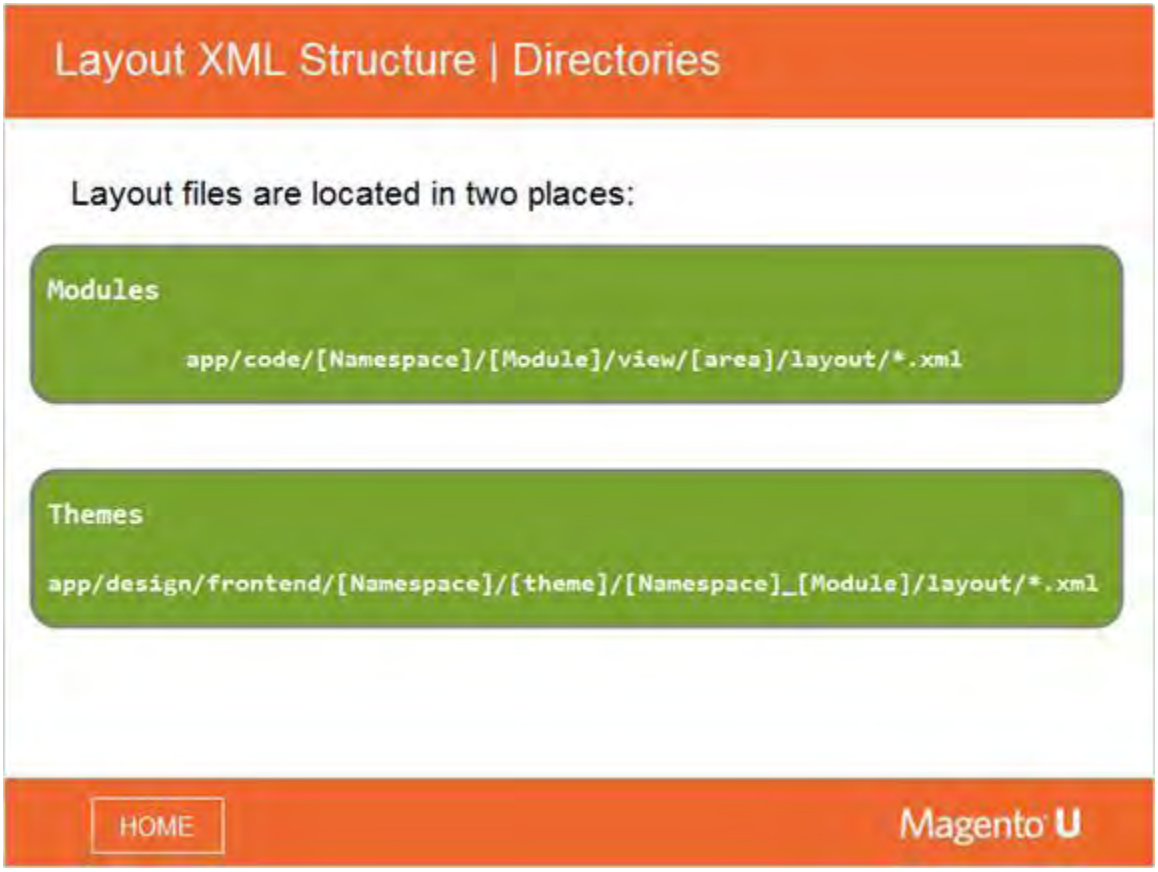
7.7 Layout XML: File Merging



Notes:

By creating Layout XML files as part of a module or theme, Magento will load and merge these files on the appropriate page.

7.8 Layout XML Structure | Directories



Notes:

There are two places where you can find layout XML files: modules and themes.

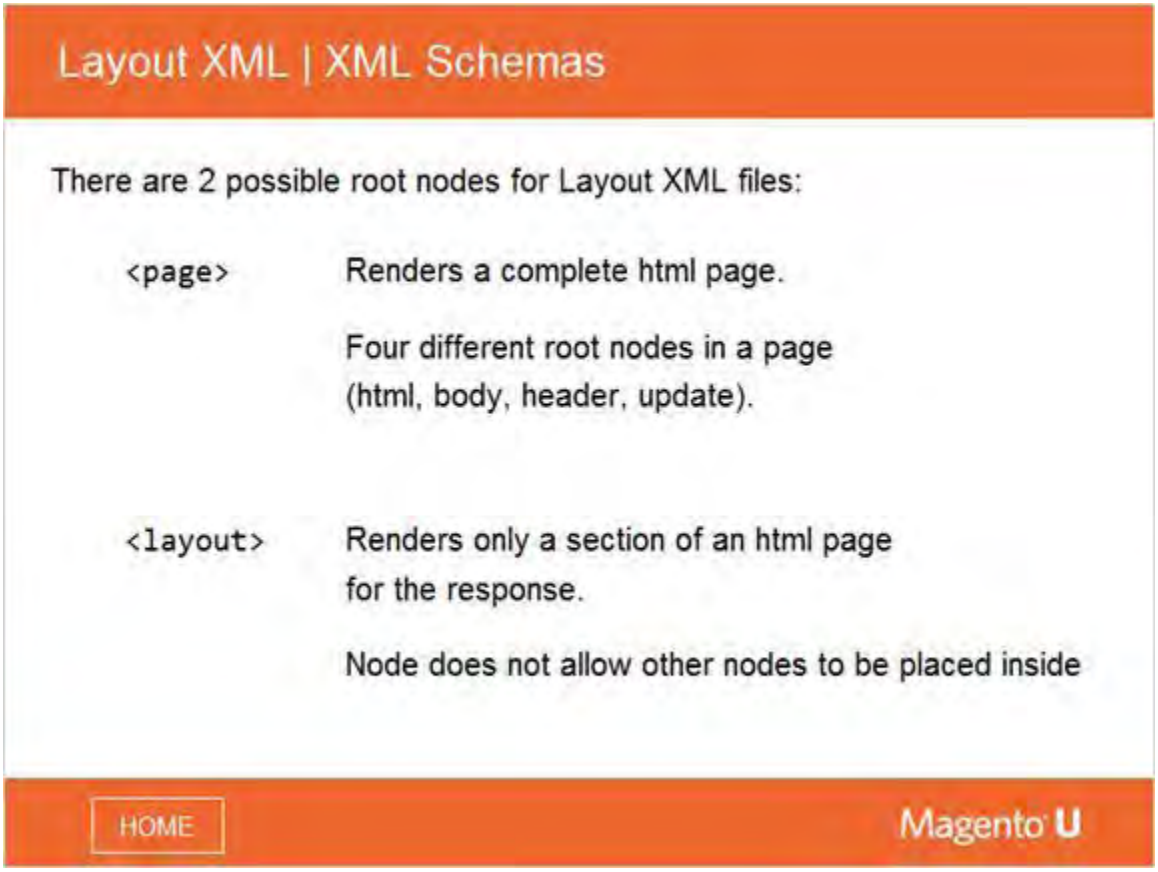
Modules: Layout xml files come from the module directories. Within each module are the view directory and the template files. Also within the directories are an area, which will be frontend or base or adminhtml, and the layout files.

Themes: In Magento 1, the base theme is base/default. In Magento 2, the module directory is the new base theme. Now, every layout file is clearly identified with a module. You can't have a layout that is not identified with a module.
The theme directory structure is `app/design/frontend` or `app/design/adminhtml`, then the `namespace/theme/etc`.

To allow separate modules to manipulate the same page, Magento uses XML merging, a very similar process to how configuration XML files are merged together.

With Layout XML, in addition to the merging of files from each module under `app/code/*/*/view/*/layout/*`, layout files from within module directories can also be extended or overridden by themes.

7.9 Layout XML: XML Schemas



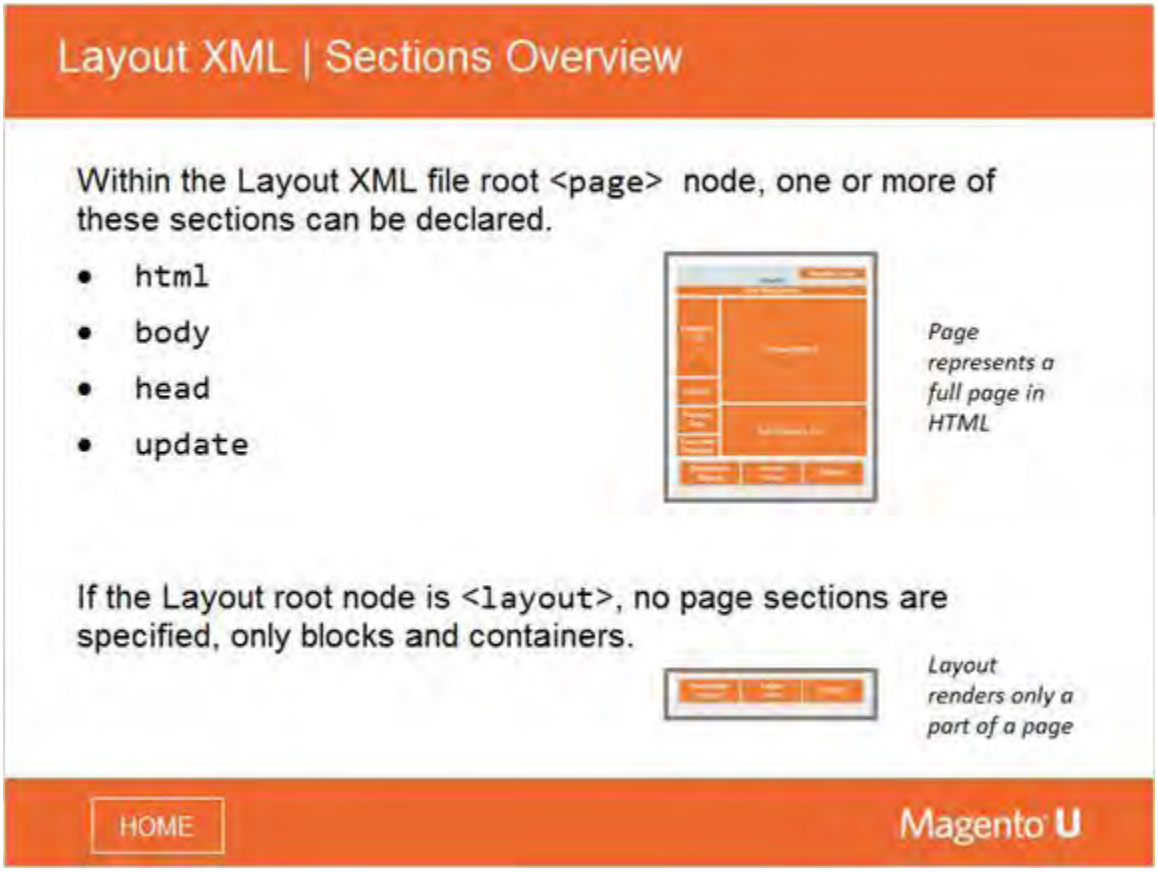
Notes:

In the layout file, there are two possible root nodes: `<page>` and `<layout>`.

`<page>`: There are four different root nodes (or page sections) in a page. The first is html, then body, header, and update.

`<layout>`: The <layout> root node does not allow other nodes to be placed inside it.

7.10 Layout XML: Sections Overview



Notes:

Why are there different root notes?

Page represents a full page in HTML. Layout represents a part of a page.

If you want to render only a section of an HTML page, then you want layout. If you want to render a container, or a block, then you want layout.

However, if you want to render a full page, then you want page. Page layout is a subset of page.

7.11 Layout XML: html Section

Layout XML | html Section Overview

```
<page>
    <html>
        <attribute name="lang" value="en"/>
        <attribute name="data-page-id" value="123"/>
    </html>
</page>
```

Rendered in root.phtml as attributes on the <html> tag.

```
<!-- Example -->
<html lang="en" data-page-id="123">
```

HOME

Magento U

Notes:

We only can define attributes. The attributes will be set in the HTML tag in the root template.

You can specify what you want here and it will be rendered as an attribute.

7.12 Layout XML: head Section

Layout XML | head Section Overview

```
<page>
    <head>
        <css    src="Namespace_Module::css/some.css"/>
        <script src="some/library.js"/>
        <link   src="Namespace_Module::js/some.js"/>
        <remove src="js/another.js"/>
        <title>Super Special Offer</title>
        <meta name ="x_ua_compatible"
              content ="IE=edge,chrome=1"/>
        <attribute name ="prefix"
            value ="product: http://ogp.me/ns/product#"/>
    </head>
</page>
```

HOME

Magento U

Notes:

Let's say you want to add a JavaScript file, or CSS file, as a resource for the page in the layout XML file.

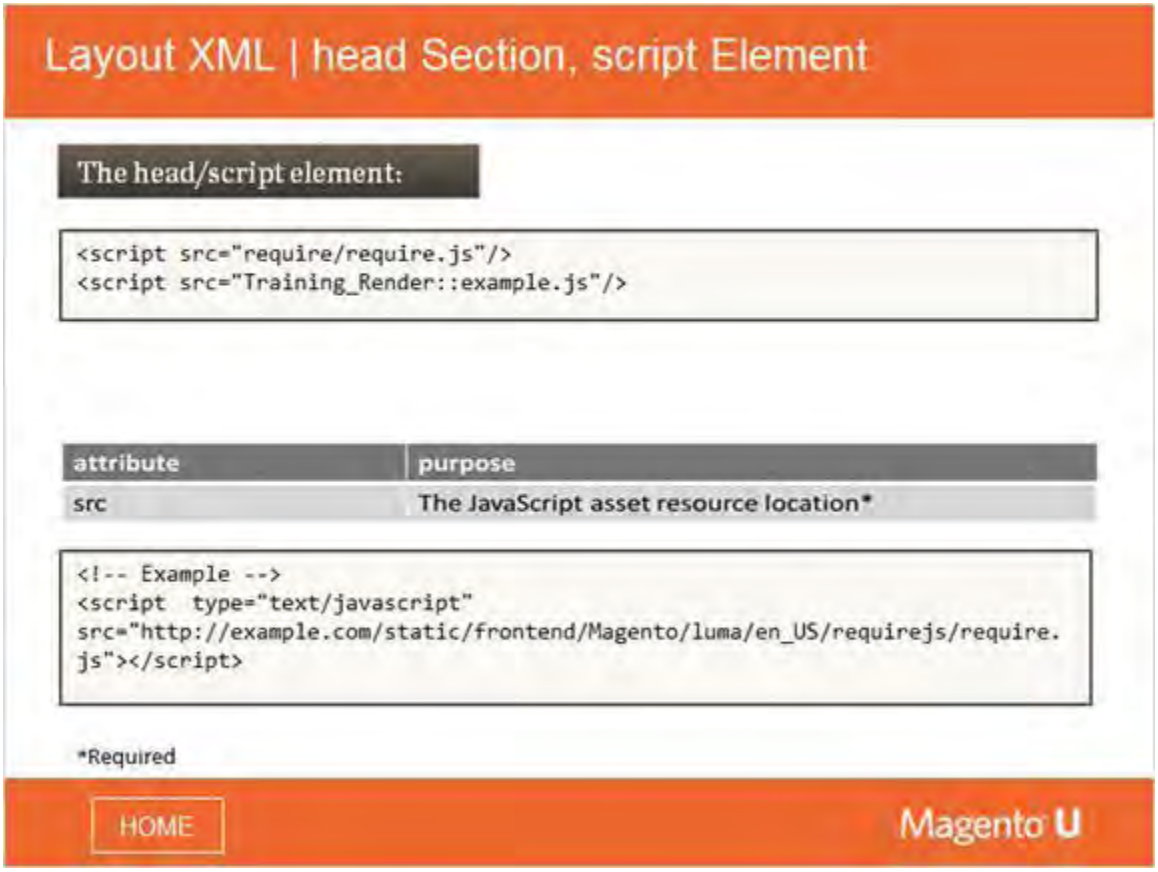
In Magento 1, there are different options available - reference name, actions, items, and so on.

With Magento 2, we now have explicit tags for each of those things. Internally, they all do exactly the same thing except for the attributes that we had in Magento 1.

All the CSS script links are adding assets to a page.

You may feel it is a lot more complicated now, but thanks to the auto-completion and schema tools, it turns out to be much easier.

7.13 Layout XML: head Section



Notes:

The head section is new element in Magento 2.

In Magento 1, there is a special “head” block, where we can have JavaScript and CSS files.

In Magento 2, the approach has changed a little. Recall that the process of rendering in Magento 2 uses `root.phtml`, which includes `$layoutContent` in the `<body>` tag, while the `<head>` section is rendered using head section.

So, the head section corresponds to the html `<head>` tag and allows you to manage everything that is placed there. For example, we can add JavaScript using the `head/script` element, as shown on the slide.

7.14 Layout XML: head Section

Layout XML | head Section, css Element

The head/css element:

```
<css src="media/gallery.css"/>
<css src="Magento_Core::prototype/magento.css"/>
```

attribute	purpose
src	The css asset resource location*

```
<!-- Example -->
<link  rel="stylesheet" type="text/css"  media="all"
href="http://m2.dev/static/frontend/Magento/luma/en_US/mage/gallery.css" />
```

*Required

HOME

Magento U

Notes:

The head section css element can either take a direct filename or module name.

7.15 Layout XML: head Section

Layout XML | head Section, link Element

The head/link element:

```
<link src="jquery/jquery-1.8.2.js"/>
<link src="Magento_Theme::example.js"/>
<link src="lib/ds-sleight.js" ie_condition="lt IE 7" defer="defer"/>
```

attribute	purpose
src	The JavaScript asset resource location*
ie_condition	Wraps the link in an IE conditional comment
defer	Defers script execution until the page is loaded

```
<!-- Example -->
<script  type="text/javascript"
src="http://example.com/static/adminhtml/Magento/backend/en_US/jquery.js">
</script>
```

*Required

HOME

Magento U

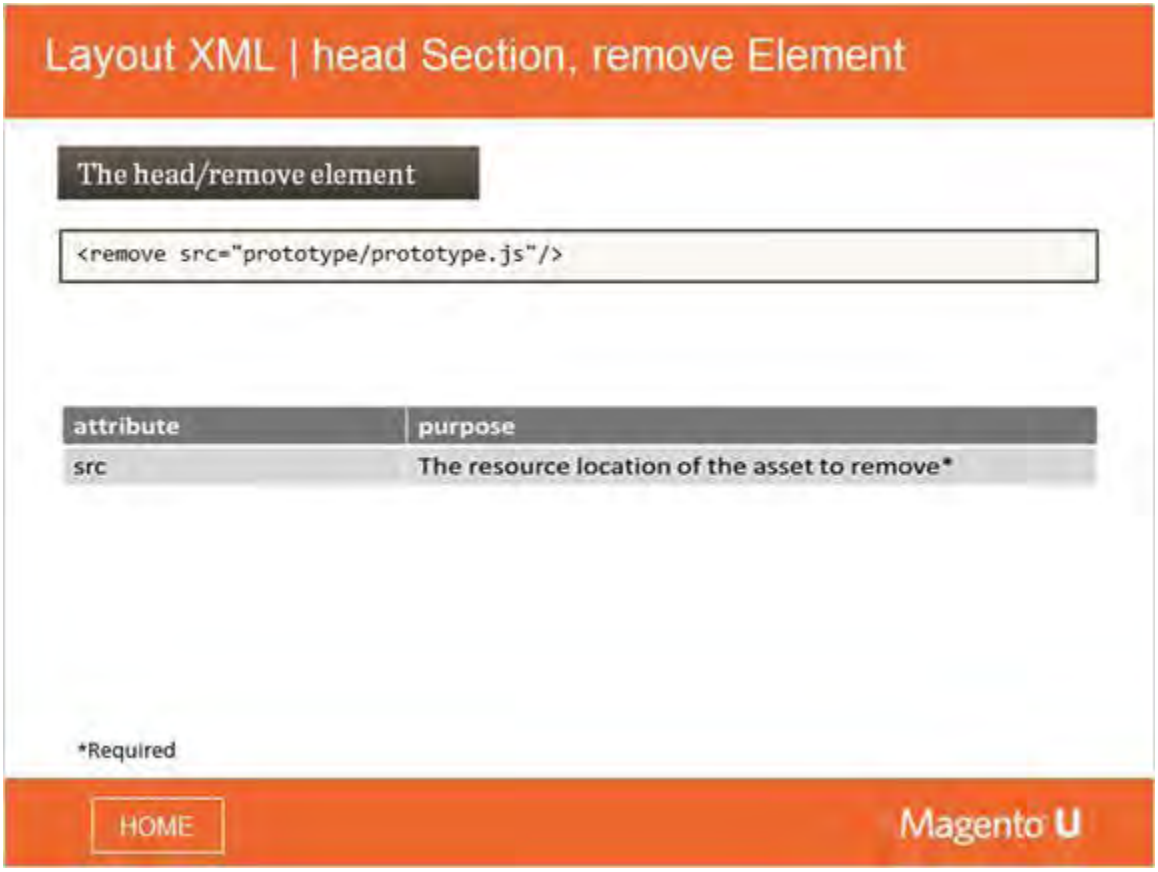
Notes:

This slide shows the `link` element of the head section.

As we can see on the slide there are three attributes available: `src, ie_condition, defer`.

Their meaning is explained on the slide.

7.16 Layout XML: head Section

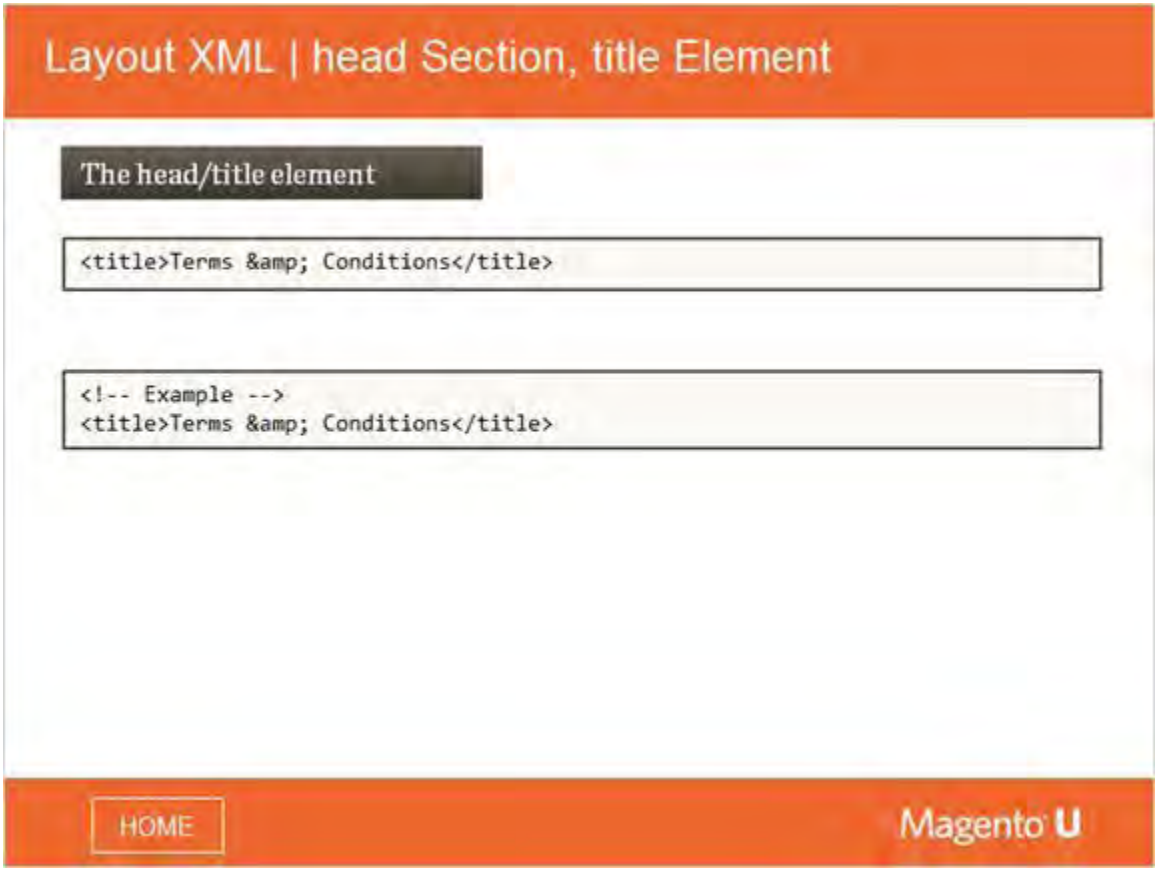


Notes:

This slide shows the `remove` element of the head section.

In the example, it allows the removal of a JavaScript library from a page.

7.17 Layout XML: head Section

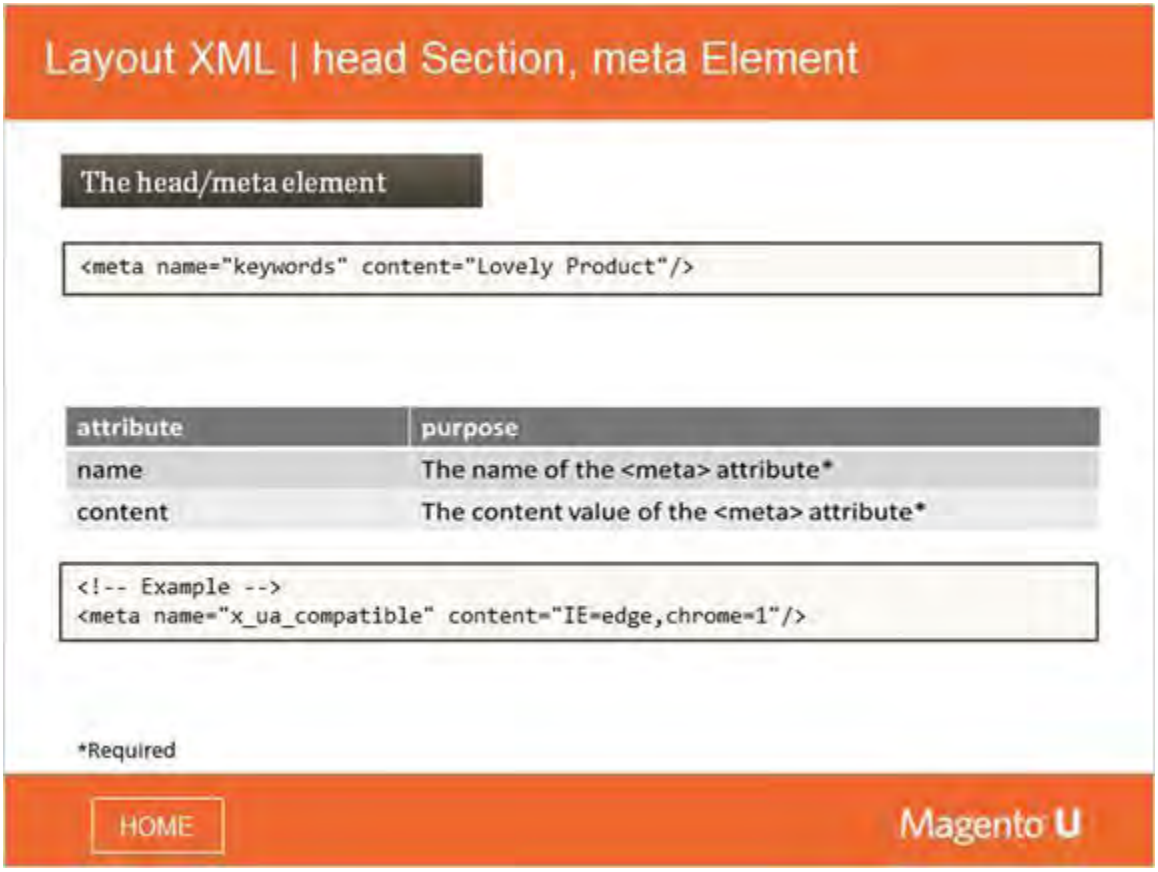


Notes:

This slide shows the `title` element of the head section.

It allows you to manage the content of the `<title>` tag within the `<head>` block of the html page.

7.18 Layout XML: head Section



Notes:

This slide shows the `meta` element of the head section.

It allows you to add the `<meta>` tag within the `<head>` block.

7.19 Layout XML: head Section

Layout XML | head Section, attribute Element

The head/attribute element

```
<attribute name="prefix"
           value="product: http://ogp.me/ns/product#"/>
```

attribute	purpose
name	The name of the <head> attribute*
value	The value of the <head> attribute*

```
<!-- Example -->
<head prefix="product: http://ogp.me/ns/product#">
```

*Required

HOME

Magento U

Notes:

This slide shows the `attributes` element of the head section.

It allows you to define attributes of the `<head>` tag. An example of why this might be useful is shown on the slide.

7.20 Layout XML: body Section

Layout XML | body Section Overview

```
<page>
    <body>
        <attribute name="id" value="example"/>
        <container name="example">…</container>
        <block class="Magento\Core\Block\RequireCookie"
            name="example">...</block>
        <move element="example"
            destination="bundled.options.container"
            before="-"/>
        <referenceBlock name="example">…</referenceBlock>
        <referenceContainer name="example">…
        </referenceContainer>
        <remove name="example"/>
        <ui_component name="example" component="paging"/>
    </body>
</page>
```

HOME

Magento U

Notes:

The next big section, and probably the most interesting one, is the body section. This is where you can add the actual content.

There are attributes inside the body that are set by the attributes in the body tag. You can declare the containers and can move elements with blocks.

The child - parent relationship is basically replaced by `<move>`.

When you reference blocks and containers, you have to explicitly state what is being referred to, like `referenceBlock` or `referenceContainer`. If you reference a block, you have the same children available as in a regular block.

Finally, there are the `remove` and `ui_components` within the body section.

Note that the container acts as a Page HTML wrapper instance because it can specify HTML tags.

"`name`" is an attribute of the block directive, which works in the same way as in Magento 1.

7.21 Layout XML: body Section

Layout XML | body Section, attribute Element

The body/attribute element

```
<attribute name="data-ng-app" value="myProductSelector"/>
```

attribute	purpose
name	The name of the <body> attribute*
value	The value of the <body> attribute*

```
<!-- Example -->
<body data-ng-app="myProductSelector">
```

*Required

HOME

Magento U

Notes:

The `<attribute>` element allows you to define the attributes of a body's tag.

7.22 Layout XML: body Section

Layout XML | body Section, container Element

The body/container element

```
<container name="additional.info" htmlTag="div"
    htmlClass="grey-box" htmlId="custom-content"
    label="Additional Content"/>
```

attribute	purpose
name	The internal name of the container*
htmlTag	HTML tag to wrap the content with
htmlClass	CSS classes to set on the container HTML tag
htmlId	HTML element ID
label	Label to use for frontend widget placement

```
<!-- Example -->
<div id="custom-content" class="grey-box">
```

*Required

HOME

Magento U

Notes:

The `container` element defines a Magento 2 container within the `<body>` section of a page.

Recall that a container allows modules to add content to that section of the page.

7.23 Layout XML: body Section

Layout XML | body Section, referenceBlock element

The body/referenceBlock element

```
<block class="Magento\Backend\Block\Template"
    name="training_render" as="example" after="system_messages"
    template="Training_Render::some/rendering/template.phtml"/>
```

attribute	purpose
name	The internal name of the target block*

*Required

HOME

Magento U

Notes:

In Magento 1, the `<reference>` layout directive is probably the most important tool for any layout updates.

In Magento 2, reference directives like `<referenceBlock>` and `<referenceContainer>` function in the same way as in Magento 1, and allow you, for example, to add a child to a block defined somewhere else.

7.24 Layout XML: body Section

Layout XML | body Section, referenceContainer element

The body/referenceContainer element

```
The body/referenceContainer element

<referenceContainer name="training.render.example">
</referenceContainer>
```

attribute	purpose
name	The internal name of the target container*

*Required

HOME

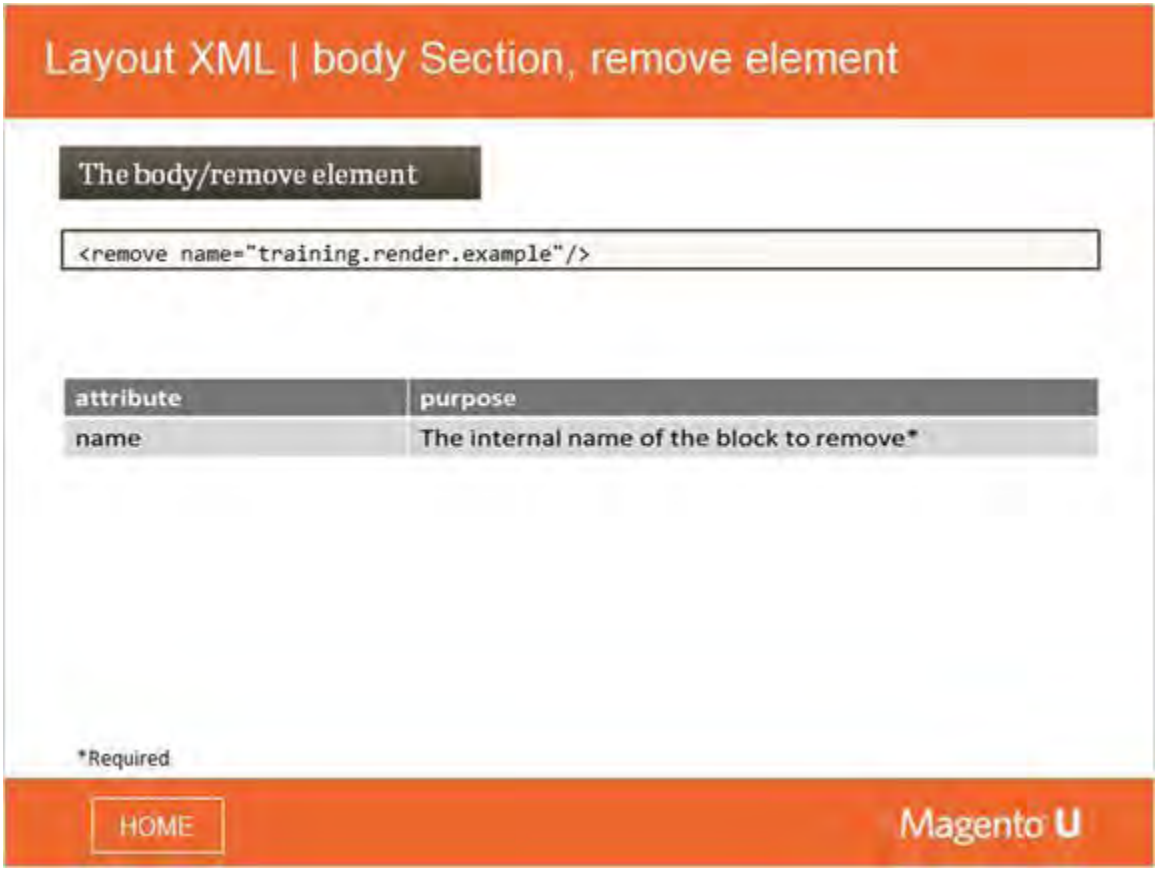
Magento U

Notes:

The `<referenceContainer>` is the “container” analogue of `<referenceBlock>`.

It functions the same way as `<referenceBlock>`, but for containers.

7.25 Layout XML: body Section



Notes:

The `<remove>` element, as implied by its name, allows you to remove any layout directive from the resulting xml on a page. It uses "`name`" to reference the node to remove.

7.26 Layout XML: body Section

Layout XML | body Section, ui_component element

The body/ui_component element

```
<ui_component name="example" component="paging"/>
```

attribute	purpose
name	The internal block name of this UI component block instance*
component	The name of the UI component to use*

```
<!-- Example -->
<div class="pager">
    <span class="pages-total-found">Total 2048 records found</span>
    <span id="productGrid-total-count" class="no-display">2048</span>
    <label class="view-pages">
    View <select name="limit" onchange="productGridJsObject.loadByElement(this)">
        <option value="20" selected="selected">20</option>
        <option value="50">50</option>
    </select> per page </label>
</div>
```

*Required

HOME

Magento U

Notes:

Ui_components are predefined generic block instances that can be configured using Layout XML when needed. The most common use case is `adminhtml` grids.

7.27 Layout XML: body Section

Layout XML | block Section, arguments/argument element

The block/arguments/argument element

```
<block class="..." name="...">
    <arguments>
        <argument name="title" translate="true"
                  xsi:type="string">Edit Account</argument>
    </arguments>
</block>
```

attribute	purpose
name	The name of the argument to be set on the block *
xsi:type	array, string, boolean, object, number, null *
translate	true or false (only allowed if xsi:type == string)

*Required

HOME

Magento U

Notes:

Each block can take arguments which are then passed to the constructor part of the data that is injected into the block through the object manager.

Each argument has a name and a type. String is the simplest type. The name is important because it has to be unique in the argument list.

If you have two different layout XML files, and both reference the same block, than the second one with the same name in the sequence will replace the first one. This is a very easy way to change blocks into arguments.

The `arguments/argument` node is used to populate the data array of the containing block.

Note that in Magento 1, to set data, you use `<action>` with magic setters or `setData()`.

7.28 Layout XML: body Section

Layout XML | block Section, action Element

The block/action element

```
<block class="..." name="...">
    <arguments>
        <action method="setTitle">
         <argument name="title" translate="true"
                  xsi:type="string">Edit Account Information</argument>
    </arguments>
</block>
```

attribute	purpose
name	The name of the argument to be set on the block *
xsi:type	array, string, boolean, object, number, null *
translate	true or false (only allowed if xsi:type == string)

*Required

HOME

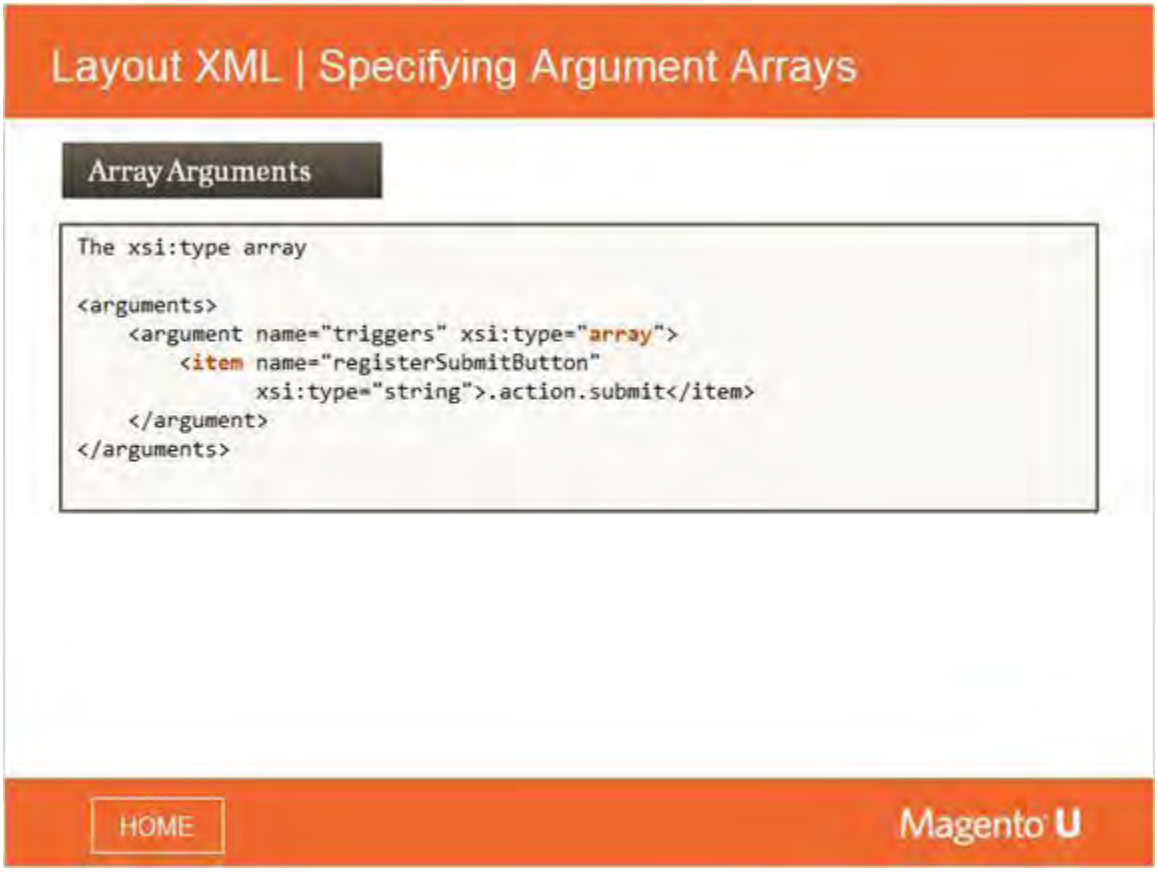
Magento U

Notes:

The `action` node is used to call methods on block instances.

Argument names have to be unique within the containing action.

7.29 Layout XML: body Section



Notes:

One of the most interesting argument types is the array. You can register as many items as you want within it. This can be both for arguments and for action arguments.

The `xsi:type array` can be used to construct argument arrays. The arrays may be nested to any depth. Each item in an array is declared with an `<item>` element, which, in turn, can be any of the argument types.

When merging, if two items within the same parent - the `<argument>` node - share the same name, the second one will overwrite the first.

8. Layout XML Loading & Rendering

8.1 Layout XML Loading & Rendering

Notes:

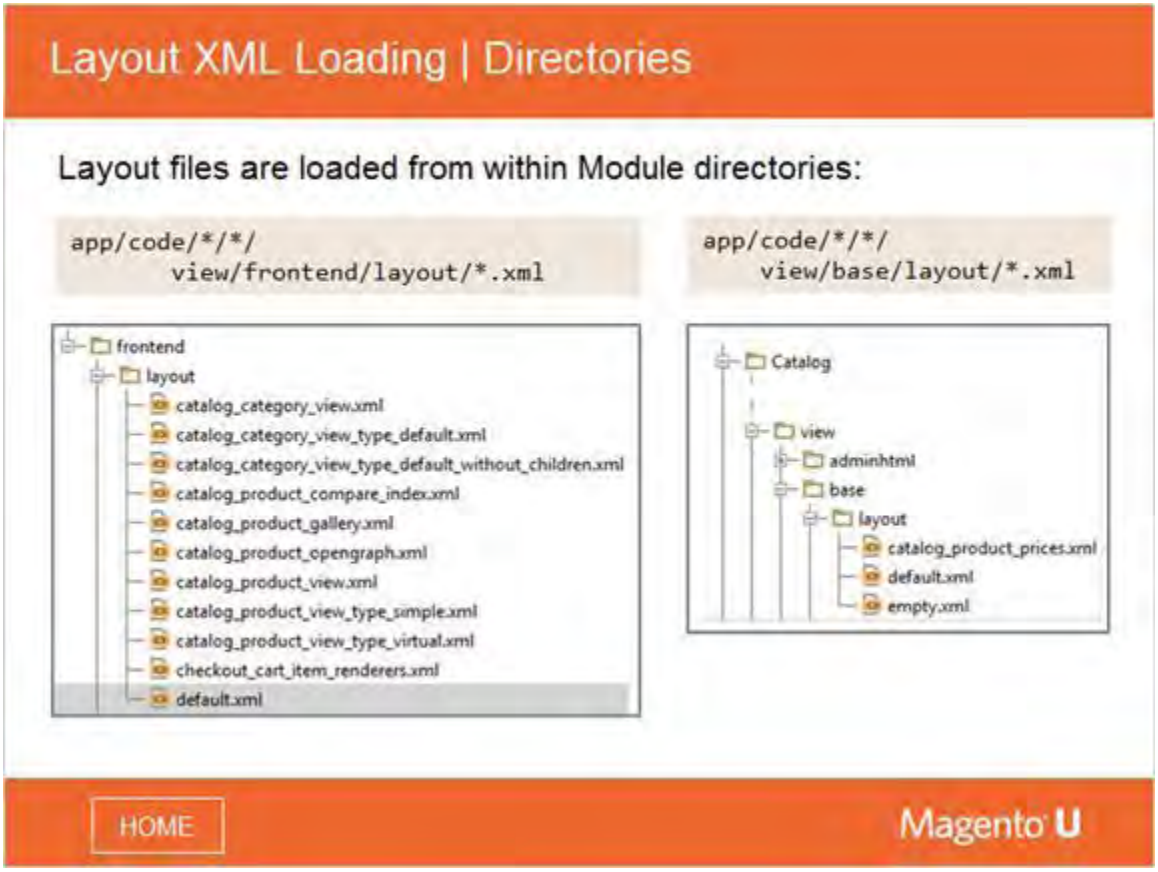
This module discusses the loading and rendering of Layout XML.

8.2 Module Topics

Notes:

In this module, we will discuss: directories; layout areas; theme inheritance; overriding layout; files; handles; page layout.

8.3 Layout Directories



Notes:

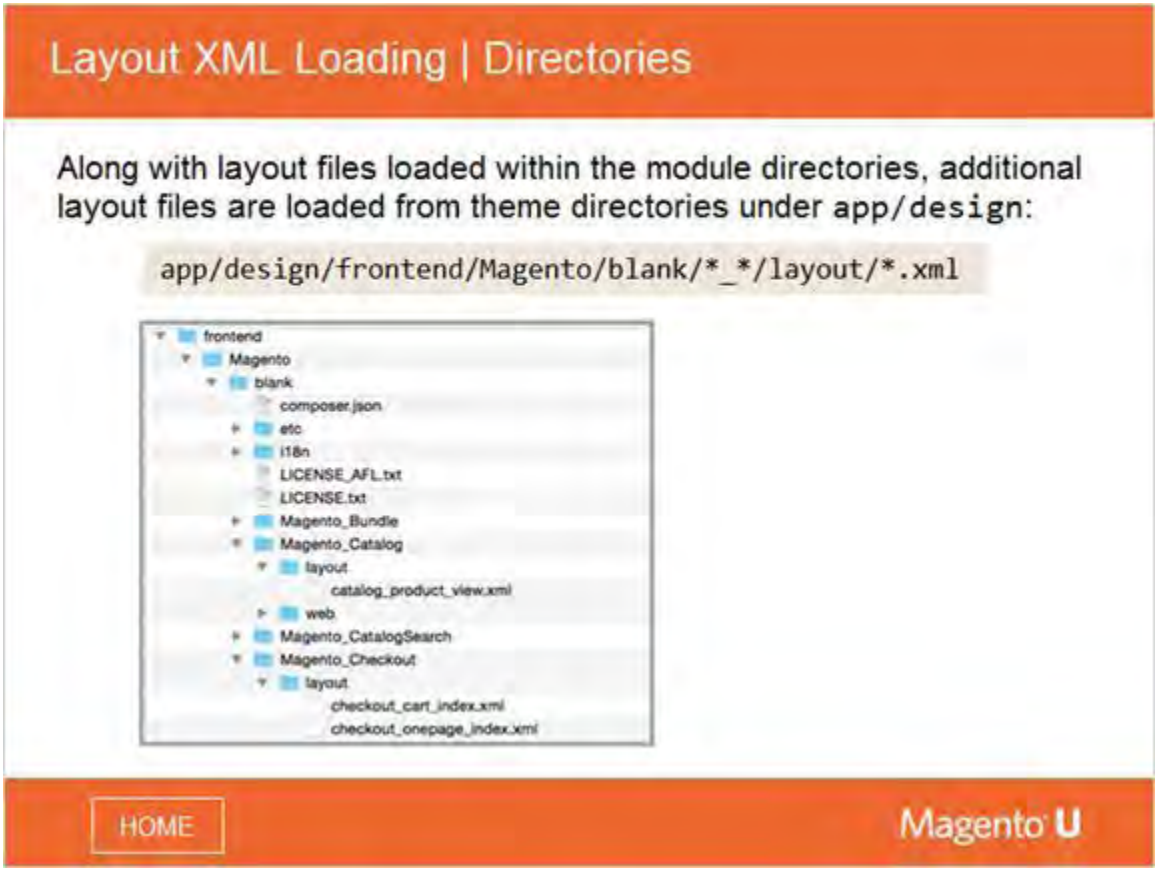
Reference: Locate the following file in your Magento installation:

```
app/code/Magento/Checkout/view/frontend/layout/checkout_cart_index.xml
```

These files are collected using `Magento\Framework\View\Layout\File\Collector\Aggregated`, which, in turn, delegates to the following collectors:

- `Magento\Framework\View\File\Collector\Base`
- `Magento\Framework\View\File\Collector\ThemeModular`
- `Magento\Framework\View\File\Collector\Override\Base`
- `Magento\Framework\View\File\Collector\Override\ThemeModular`

8.4 Layout XML Loading | Directories



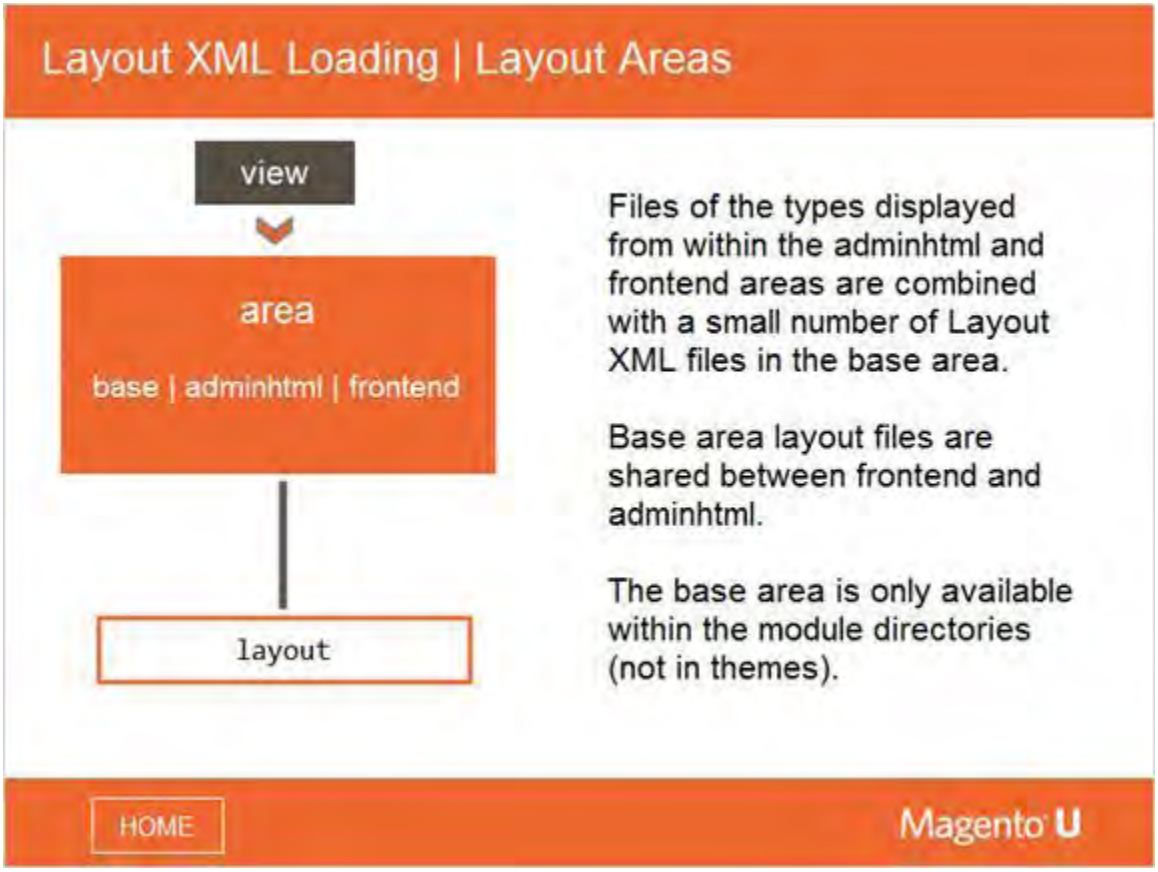
Notes:

Along with layout files loaded within the module directories, additional layout files are loaded from theme directories under `app/design`.

Reference: Locate the following xml file in your Magento installation:

```
app/design/frontend/Magento/blank/Magento_Catalog/layout/catalog_product_view.xml
```

8.5 Layout XML Loading | Layout Areas



Notes:

The system looks for these types of files to load. The view areas within the modules:

- base
- adminhtml
- frontend

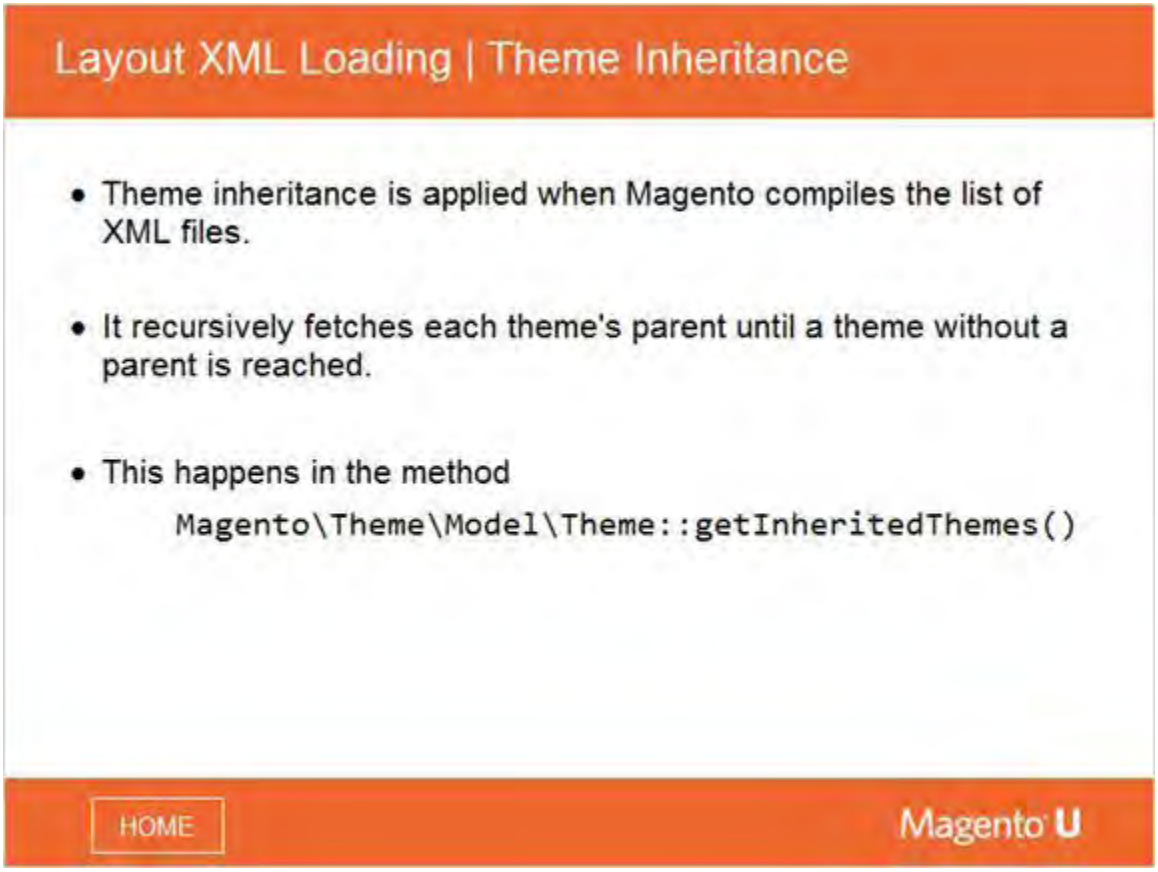
The view areas within the theme directories:

- adminhtml
- frontend

The files from within the adminhtml and frontend areas are combined with a small number of Layout XML files in the base area. The layout file list is always one of the following combinations:

- base + adminhtml
- base + frontend

8.6 Layout XML Loading | Theme Inheritance



Notes:

When the XML files are loaded, Magento applies an inheritance theme at the same time. You can apply a theme and it will look for the parent until a theme without a parent is reached.

How do you debug theme inheritance?

Look for an entry point so you can locate the method `getInheritedThemes()` and with that you can figure out the inheritance.

Themes in Magento can inherit other themes. This is configured in a `theme.xml` file or `composer.json` file within the theme. The list of theme Layout XML files is built by recursively looking within each theme in the inheritance list for a matching XML file.

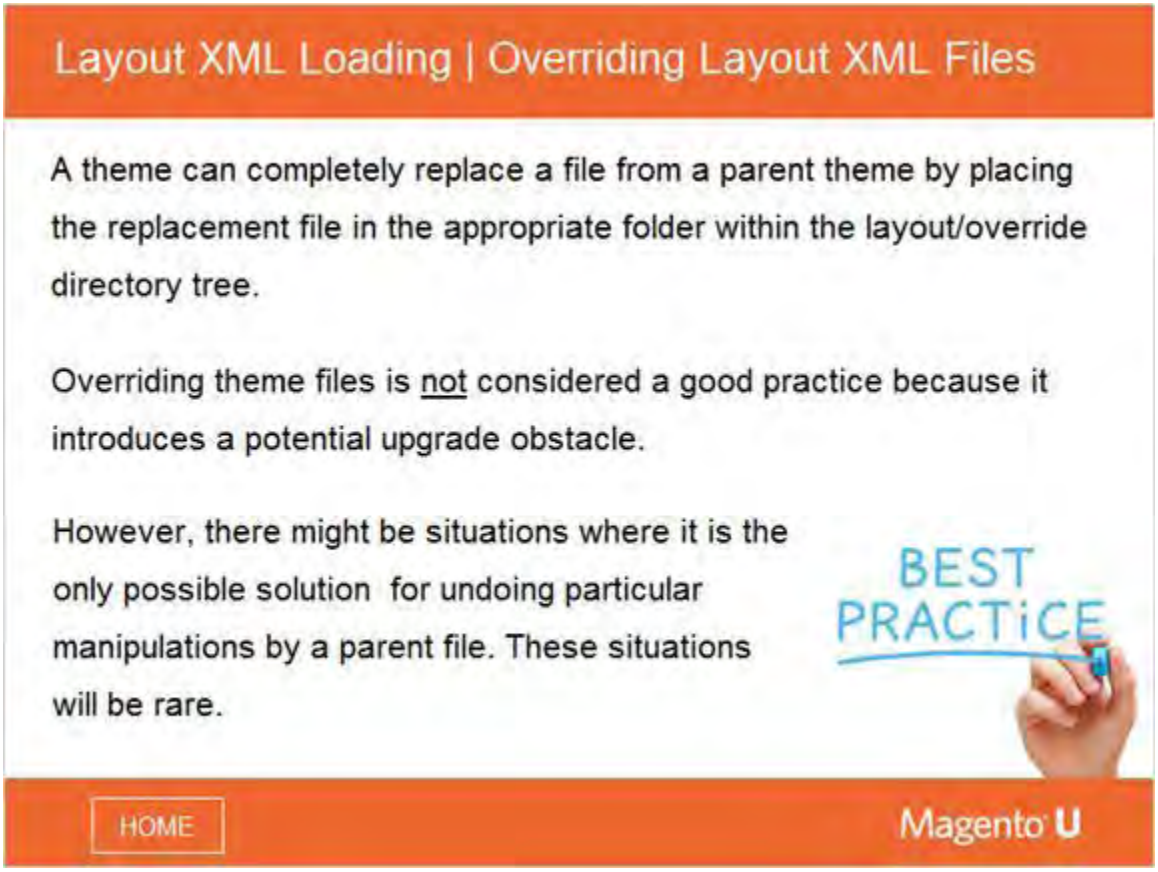
Usually, the last parent theme is `Magento/blank`.

Developers can create completely independent theme hierarchies.

Note: Magento 1 uses primarily a "theme fallback" concept, while in Magento 2 it has been completely replaced by a "theme inheritance" concept.

These concepts work similarly, but the Magento 2 version provides more flexibility with the option to put layout XML files into module directories. Starting with Magento 1.14 EE / 1.9 CE, inheritance is also supported in addition to the theme fallback, but not the placement of layout XML files in module directories.

8.7 Layout XML Loading: Overriding



Notes:

Overriding Layout XML is not a good practice, but it is sometimes necessary to use. Basically, it replaces a file in the fallback.

Note that Magento 2 is different from Magento 1. In Magento 2, you have to specify which file you want to override.

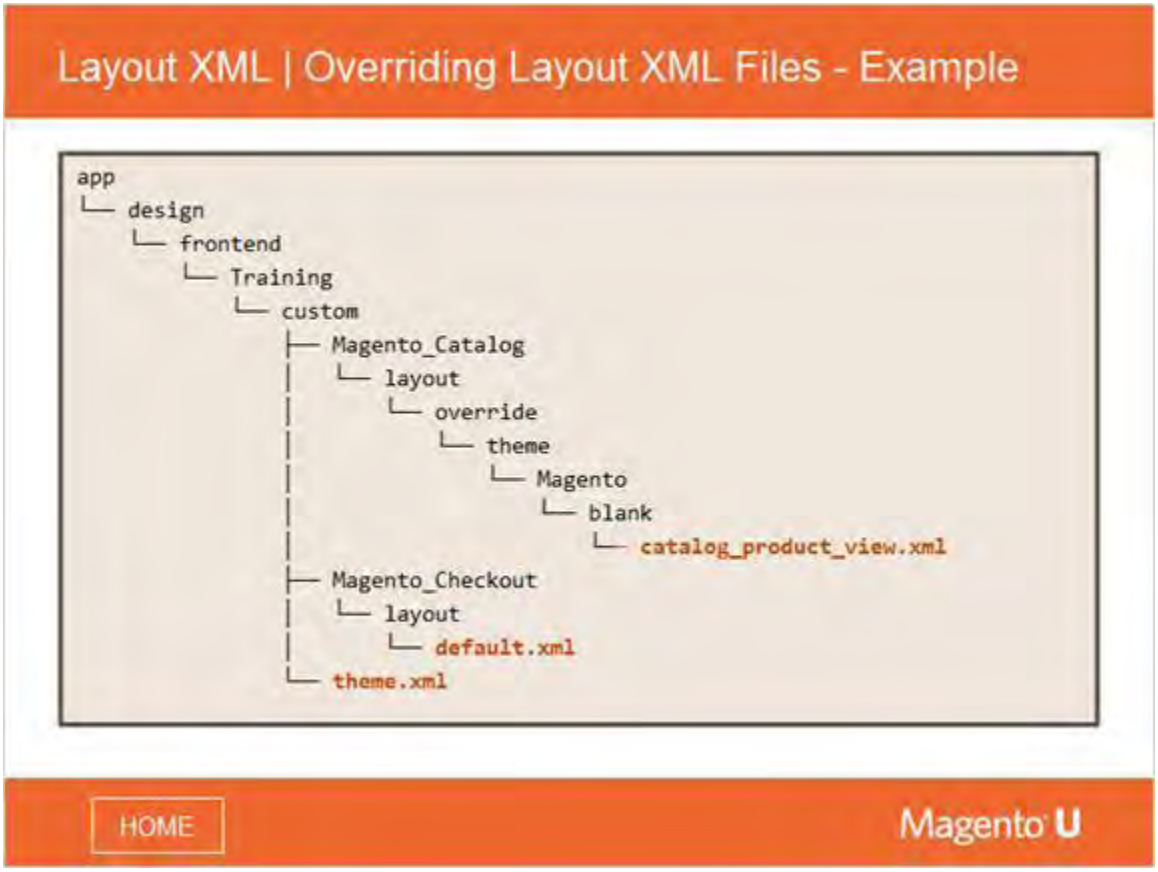
Base Override Files replace files from the **base** theme: `frontend/Magento/blank/*_*/layout/override/base/*.xml`

Theme Override Files are from a specific **parent** theme: `frontend/Magento/blank/*_*/layout/override/theme/*/*/*.xml`

Examples:

```
frontend/Training/custom/*_*/layout/override/Magento/blank/*.xml
frontend/Magento/blank/*_*/layout/override/base/*.xml
frontend/Magento/blank/*_*/page_layout/override/base/*.xml
```


8.8 Layout XML: File Structure Example



Notes:

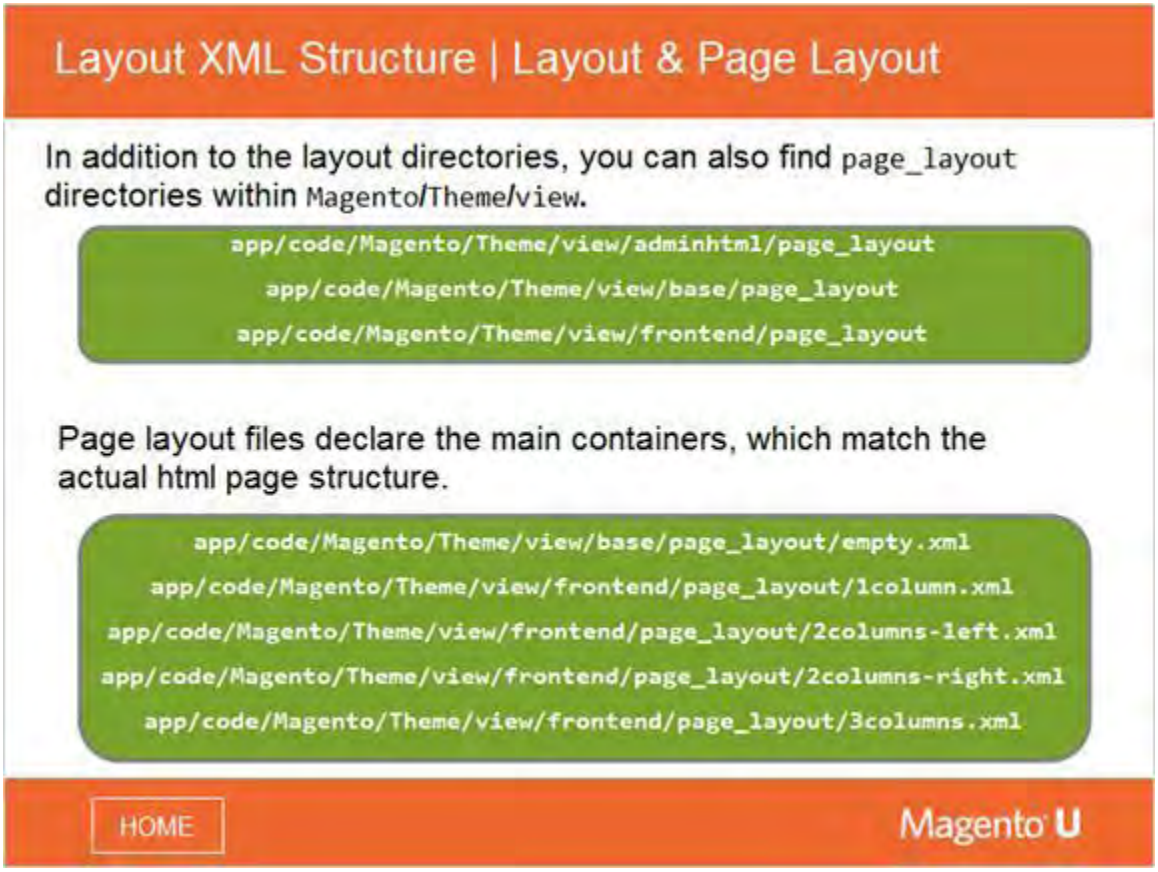
This is an example of the file system structure for a theme containing two Layout XML files.

The `Magento_Checkout/layout/default.xml` file will be merged with all other `default.xml` files from the parent themes and all modules.

The `Magento_Catalog/layout/override/theme/Magento/blank/catalog_product_view.xml` file will replace the file with the same name from within the `Magento/blank` theme during the merging process.

The file `theme.xml` is not a Layout XML file, it is a theme configuration file containing, for example, the theme title and preview image name.

8.9 Layout XML: Layout & Page Layout



Notes:

In Magento 2, there are not only layouts but also page layouts.

The layout XML files in the `page_layout` directories follow the same syntax rules as the layout XML files located in the usual layout directories. The XML files from the `page_layout` directories are combined with the list of files from the layout directories using `Magento\Framework\View\Model\Layout\Merge::_loadFileLayoutUpdatesXml()`, and are treated just like any other layout XML file thereafter.

Out of the box, you get three page layout directories, located in `Magento/theme`. Magento page layout files contain the base for the type of page layouts you want. For example: one-column, 2-column left, 2-column right, and so on.

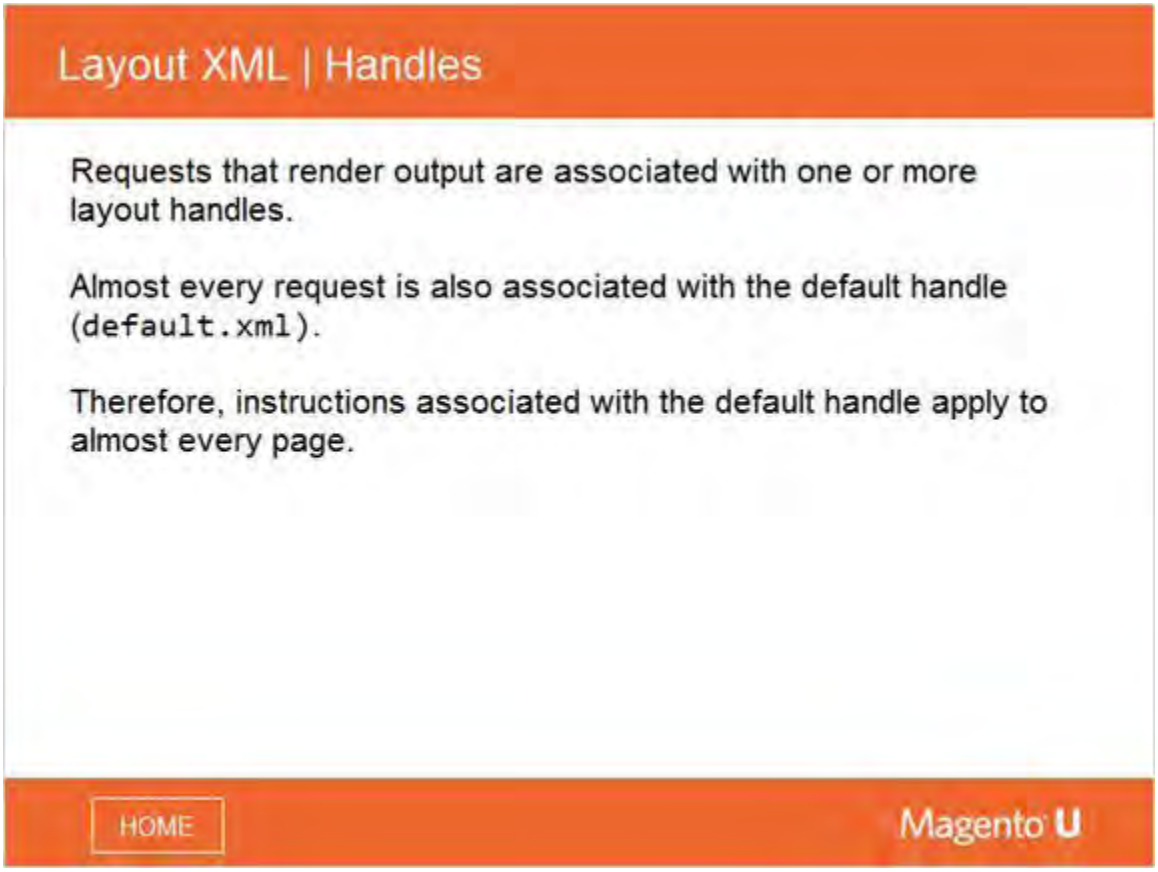
If you create the layout result object, it only includes the regular layout because it requires the root templates to be rendered by the page object.

8.10 Layout XML: Handles

Notes:

Layout handles or update handles are how the Magento view knows which layout XML instructions to process for a given request.

8.11 Layout XML: Handles



Notes:

Almost every page in Magento is associated with the **default** handle.

It is used to set up the main containers and content elements that are available on every page.

If more than one layout handle is associated with a request, the default handle is always processed first.

8.12 Layout XML: Handles

Layout XML | Handles

Page-specific content is associated with a page using the action handle specific to the request (ex: `catalog_product_view.xml`).

The action handle consists of the route, controller and action in lower case, separated by an underscore.

Examples:

- `catalog_product_view`
- `customer_account_login`
- `cms_index_index`
- `cms_page_view`

HOME

Magento U

Notes:

The page-specific action handle is used to add content to the generic page structure defined by the default handle.

The action handle is always processed after the default.

8.13 Layout XML: Additional Handles

Notes:

The contents of update handles are processed in the order in which the update handles are added.

8.14 Layout XML: Custom Handles

Notes:

To add custom update handles, the method `addHandle()` can be called on the result page object (`Magento\Framework\View\Result\Page` or `Magento\Framework\View\Result\Layout`) before the response is rendered.

Alternatively, custom handles can be added using the Layout XML `<update>` directive.

8.15 Layout XML: Page Layout

Layout XML | Page Layout

The basic page layout is selected with the `layout="..."` attribute on the `<page>` root node.

The allowed values match the files in the `Magento/Theme/view/page_layout/*` directories.

For example:

- 1column
- 2columns-left
- 2columns-right
- 3columns

```
<?xml version="1.0"?>
<page layout="2columns-left" ....>

    .... The Layout XML here ....

</page>
```

HOME

Magento U

Notes:

Not every page tag needs to specify the layout attribute. It is used only when creating a new page, or changing the layout for an existing page.

8.16 Exercise 3.8.1

Reinforcement Exercise (3.8.1)

The solution for this exercise can be found in the Magento 2 Fundamentals Exercise Solutions guide.

- Add a `default.xml` layout file to the `Training_Render` module.
- Reference the `content.top` container.
- Add a `Magento\Framework\View\Element\Template` block with a custom template.
- Create your custom template.
- Check that the template content is visible on every page.

HOME

Magento U

8.17 Exercise 3.8.2

Reinforcement Exercise (3.8.2)

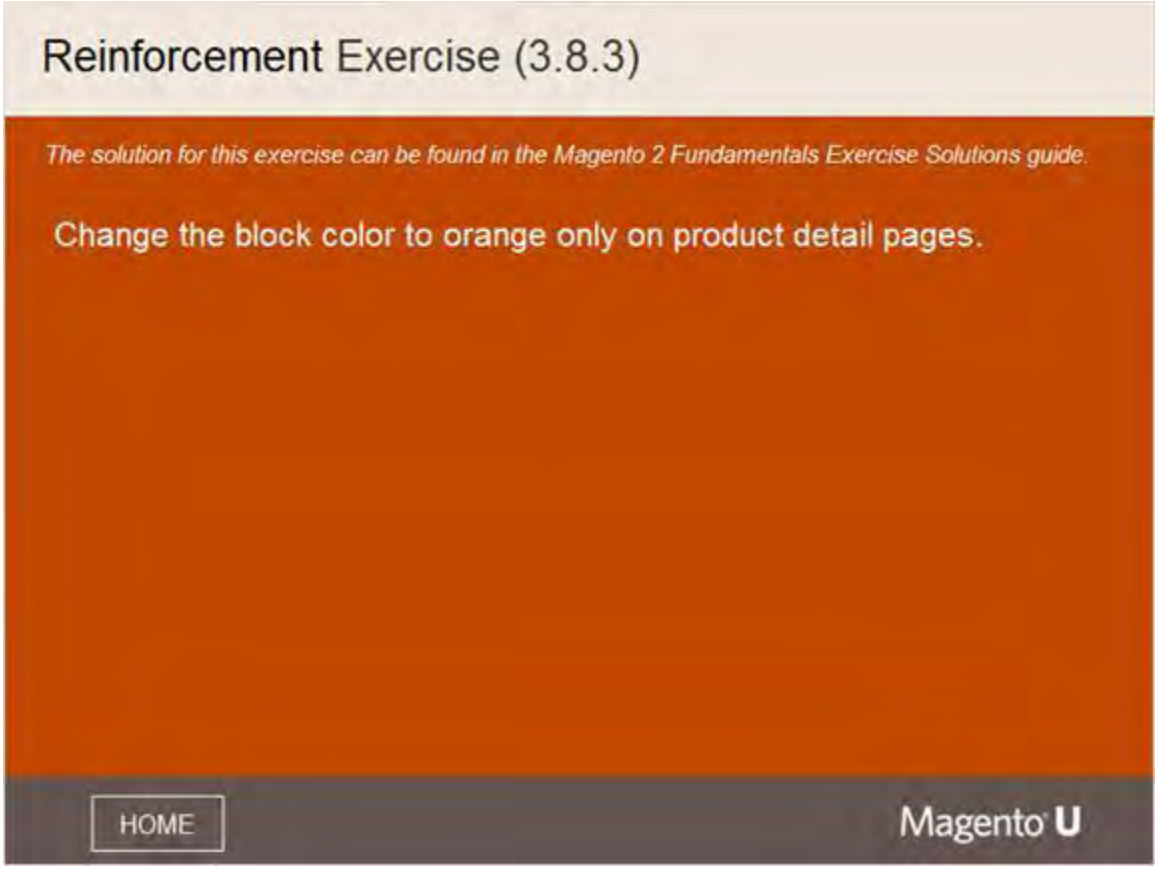
The solution for this exercise can be found in the Magento 2 Fundamentals Exercise Solutions guide.

- Add an `arguments/argument` node to the block.
- Set the argument name to `background_color`.
- Set the argument value to `lightskyblue`.
- In the template, add an inline style attribute to a `<div>` element:
  ```
  style="background-color:
  <?= $this->getData('background_color') ?>;"
  ```
- Confirm the background color is displayed.

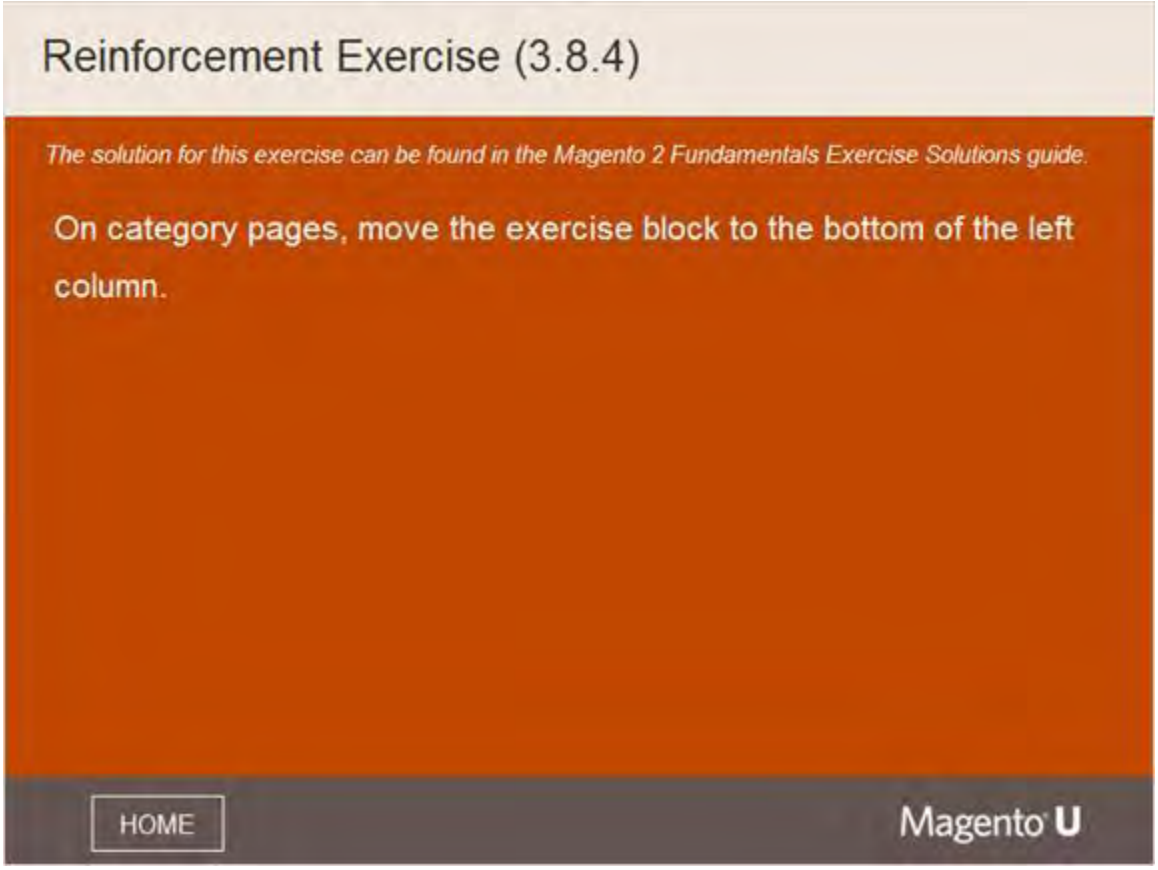
HOME

Magento U

8.18 Exercise 3.8.3



8.19 Exercise 3.8.4



8.20 Exercise 3.8.5

Reinforcement Exercise (3.8.5)

The solution for this exercise can be found in the Magento 2 Fundamentals Exercise Solutions guide.

- Create a new controller action (ex: `training_render/layout/onepage`).
- For that action, choose a single column page layout using layout XML.
- Set a page title using layout XML.

HOME

Magento U

8.21 Exercise 3.8.6

Reinforcement Exercise (3.8.6)

The solution for this exercise can be found in the Magento 2 Fundamentals Exercise Solutions guide.

On the custom action you just added, remove the custom block from the `content.top` container. (see exercise 3.8.1)

HOME

Magento U

8.22 Exercise 3.8.7

Reinforcement Exercise (3.8.7)

The solution for this exercise can be found in the Magento 2 Fundamentals Exercise Solutions guide.

Using Layout XML, add a new link for the custom page you just created to the set of existing links at the top on every page.

HOME

Magento U