No Frills Magento 2 Layout

Alan Storm

April 2018

IMPORTANT

Contents

Chapter 1

Introduction

The kindest thing I can say about writing this book is that it's been a challenge. The best way to introduce this book is to explain why it's been so challenging.

Writing the first edition of No Frills Magento Layout was no picnic. Even in 2011, the level of knowledge around Magento's HTML-generating domain-specific-language was scant. However, in the world of web development, there **was** a consensus on how we should be building web applications.

1. HTML delivered to the browser, possibly generated with server-side languages like PHP, ruby, python, java, etc.
2. CSS for layout and styles, with an occasional image or background-image if you wanted to get fancy
3. Javascript to add interactive elements to the page
4. AJAX requests if those interactive elements needed additional server resources/information

Even in 2018, these four principles remain a solid and proven way to build software that's delivered to users via a web browser.

Unfortunately, in the years since Magento's release and my writing the first edition of this book, **a lot** has changed in the world of web development.

The biggest change has been the rise of mobile and the stagnation of the mobile web. Chrome and Safari on Android and iOS based phones are wonders of software engineering, but the web browser remains a second class citizen in the battery constrained world of mobile devices. Where the aughties culminated with web-browsers recognized as the superior method of delivering cross platform software, the mobile computing world took a few steps backwards into device specific software and locked-in file formats. In this enviornment "the web" became a transport layer for data, which in turn led to the server side world becoming a specialized, UI-less island.

CSS has also seen its fair share of changes. Again, the aughties saw a wealth of clever CSS design and development techniques arise, often on a grassroots level. These techniques allowed web developers to tame the complicated world of cascading style sheets. In 2018, these techniques still exist, but they're locked into specific frameworks (Bootstrap) or preprocessing programs (LessCSS, Sass). When you consider a mobile web that has given up on a stable "CSS API" it's clear that writing your style sheets without one of these tools is a dicey proposition. By itself this might not be a problem, but these tools are often developed inside of agency culture, a culture that's more interested in short term results than they are in long term sustainability. This makes for complicated toolchains that change dramatically year-to-year.

All this bleeds over to the world of Javascript, where it's both the best of times and worst of times. The ubiquity of javascript in the browser has brought a **tremendous** amount of engineering resources to bear on the language, runtimes, and third party libraries. There's now a school of thought that says the **only** HTML you need to render on the server is the HTML that bootstraps your javascript runtime. However, despite (or because of?) all this attention, javascript based applications tend to be built-on hundreds of thinly sliced libraries with no dominant paradigm from application to application.

The PHP landscape has also seen its fair share of changes – the most important of which is Composer. While Composer modestly bills itself as a dependency manager, in reality it's taken off (and over) as PHP's de-facto package manager, autoloader bootstrap enviornment, and distribution channel for PHP code.

Another change has been PHP 7.0, 7.1, and 7.2's making a Zeno like march towards explicitly typed language semantics. PHP 7 lets developers familiar with java like semantics ply their wares like never before, and the tension between done-quick vs. done-correct has never been higher within the PHP community.

While some would chalk all this up as progress, it's balkanized the developer community. *The Right* way to develop a web application is much less clear. There's a tension as to how much attention we should pay to the web application vs. just building an API for mobile support.

Magento 2 jumped into a web development world that's dramatically changed, and it's not clear which direction (if any) Magento core is leaning with regards to to the chaos.

Magento 2

With Magento 2, Magento Inc.'s committed hard to PHP's trends. Composer's fully embraced, Magento 2's PHP semantics favor multi-level class based abstractions, and the XML configuration files have multiplied like guinea pigs. While all this does confer the Magento system with certain enterprise market

advantages, it's also created a system where getting started is a tad more complicated than dropping your PHP files into a `public_html` folder.

Magento's Layout XML files are a *domain specific language* that use PHP classes and templates to render HTML. In this way, Magento 2 is a traditional server side framework, similar to the version developed in the aughties.

However: Magento 2 has **also** revamped its API. Formerly a tool for data transport, Magento 2's new API puts REST services front and center, including browser based sessions and permissions. It's the sort of API you can call directly from javascript.

To support building javascript applications using this API, Magento 2 also features a RequireJS module system, a significant extension of Knockout.js for AJAX based template rendering, and a **new** domain specific language that enables pure javascript based UIs with **no server side rendered HTML**. In this way, Magento seems to be forward thinking.

However however: While Magento 2 has these newer systems in place, the Magento 2 application remains a *mix* of pure javascript UI, and the older server side rendered HTML *enhanced* by javascript.

Finally, in the realm of CSS, the Magento application ships with a heavily integrated LessCSS system and a grunt build enviornment. There's been community led efforts to enable both Sass and gulp for Magento 2 development workflows, but the core system itself has LessCSS as a hard dependency. This means developers targeting **all** Magento systems face some hard choices with regards to to their CSS build pipelines.

What We'll Cover

All of which is a preamble to saying: This book is not a top to bottom course in Magento's front end systems. Also, while you can get something out of this book if you're coming in fresh to Magento, you'll be best served if you already have an inkling of where Magento 2 came from. To help with this we've included a copy of the original No Frills Magento Layout with your purchase.

We'll cover using Magento's *layout handle XML files* to render HTML content at a module level. We'll also cover what *themes* are in Magento 2, how their handling of layout handle XML files has changed, and how you can use LessCSS via themes to style Magento pages. Finally, we'll take a brief survey of how Magento uses new front end technologies like RequireJS, Knockout.js, and LessCSS and show you how to get your front end files loaded into Magento's application context.

Conventions

There's a few last bits of intro business before we can get on with the book.

When you downloaded your copy of this book you also received a `Pulsestorm_Nofrillslayout` Magento module. You'll want to install this single module into your system, as many of the code samples included in this book start with this module as their base. If you're unsure how to install a stand alone module in Magento 2, read the "Installing the `Pulsestorm_Nofrillslayout` Module" appendix.

Once installed, you'll want to flip your system into `developer` mode. The simplest way to do this is to run the following command

```
1 $ php bin/magento deploy:mode:set developer
```

If you're not familiar with Magento's various modes, or not familiar with the command line, we have an appendix that covers each of those as well.

The appendixes are quick primers on topics that aren't *quite* related to Magento's layout system, but are still required knowledge for working with the system. We've broken them off into appendixes to avoid breaking flow in our tutorials.

Alright, let's get to it!

Chapter 2

Block Basics

The atomic unit of an HTML page in Magento 2 is a block. A block is a PHP object that renders a bit of HTML. To start with, we're going to use PHP to instantiate a block object, and then output a bit of HTML.

After installing the `Pulsestorm_Nofrills` module/extension, load the following URL in your browser (substituting the `magento.example.com` URL with your own)

```
1 http://magento.example.com/pulsestorm_nofrillslayout/
      chapter1
```

You should see a simple `Hello World` message displayed. The code that creates this page can be found here

```
1 #File: app/code/Pulsestorm/Nofrillslayout/Controller/
      Chapter1/Index.php
2 public function execute()
3 {
4     echo "Hello World";
5     exit;
6 }
```

This file is a *Magento Controller*. You don't need to understand the specifics here, but Magento implements a version of the popular Model, View, Controller programming pattern. For Magento, "MVC" means that every specific URL will call the `execute` method of a specific controller file. We're going to write our sample code inside these controller files. We're doing this because it's the easiest way to get a PHP enviornment that's bootstrapped with Magento's standard objects and libraries. If you're interested in learning more about Magento as an MVC framework, the *Magento 2 for PHP MVC developers series*

http://alanstorm.com/category/magento-2/#magento-2-mvc

is a good place to start.

Try changing the code above so it matches the following

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter1/Index.php
public function execute()
{
    $block = new \Pulsestorm\Nofrillslayout\Block\Chapter1\
        Hello;
    echo $block->toHtml();

    exit;
}
```

Above, we've changed the code so it

1. Creates a new PHP object from the class named `Pulsestorm\Nofrillslayout\Block\Chapter1\Hello`
2. Calls that object's `toHtml` method
3. Uses PHP's `exit` statement to halt normal Magento page rendering

If you reload the URL, you should see the following message

```
Hello World from a Block
```

Congratulations – you just instantiated your first block object, and used that block object to render a chunk of HTML.

What is a Block

OK, so what just happened up there? Lets take our code apart line-by-line.

```
$block = new \Pulsestorm\Nofrillslayout\Block\Chapter1\
    Hello;
```

This line uses PHP's `new` statement to create a new object from the PHP class `\Pulsestorm\Nofrillslayout\Block\Chapter1\Hello`. If you're confused by the backslashes, those are PHP namespaces. In PHP versions greater than 5.3, you can organize your classes into a tree of namespaces, sort of like a file system.

Many modern coding conventions dictate you should use a `use` statement in your class files, which will let you use a class's short name. This would look something like the following.

```
use Pulsestorm\Nofrillslayout\Block\Chapter1\Hello;

```

```
3  //...
4  function execute()
5  {
6      $block = new Hello;
7      //...
8  }
```

The `use` statement imports the PHP class into the current namespace under the short name `Hello`. While you'll need to understand the `use` statement to work with Magento 2 (and most modern PHP frameworks), we'll try to stick to fully namespaced classes in this book.

If we take a look at the next line

```
1  echo $block->toHtml();
```

we see a call to the `toHtml` method of our block object. Let's take a look at the definition file for the `Pulsestorm\Nofrillslayout\Block\Chapter1\Hello` class, which should contain the definition for the `toHtml` method. You're probably wondering where you can find that source file.

Most PHP frameworks, Magento included, use PHP's autoloader feature to automatically load class definition files. Most modern PHP frameworks use one of the PSR autoloading standards (PSR-0 or PSR-4).

So where's our class file? Magento converts a class name like `Pulsestorm\Nofrillslayout\Block\Chapter1\Hello` into a file path like

```
1  Pulsestorm/Nofrillslayout/Block/Chapter1/Hello.php
```

and then searches a set number of base directories for your class definition file. If you're interested in learning more, checkout the appendix on PHP autoloaders. In our case, that's the following file.

```
1  #File: app/code/Pulsestorm/Nofrillslayout/Block/Chapter1/
       Hello.php
2  namespace Pulsestorm\Nofrillslayout\Block\Chapter1;
3  class Hello implements \Magento\Framework\View\Element\
       BlockInterface
4  {
5      public function toHtml()
6      {
7          return '<p>Hello World from a Block</p>';
8      }
9  }
```

Here we see the definition of our block class. All an object needs to do to be a block is

1. Have a method named `toHtml`.

2. Have the `toHtml` method return a rendered HTML string.

Our class does both these things. You'll also notice our class implements an interface. If you're not sure what an interface is for, checkout the interfaces appendix.

```
1 implements \Magento\Framework\View\Element\BlockInterface
```

If we take a look at this interface

```
1 <?php
2 #File: vendor/magento/module-cms/Api/Data/BlockInterface.
      php
3 namespace Magento\Framework\View\Element;
4
5 interface BlockInterface
6 {
7     public function toHtml();
8 }
```

we see a single `toHtml` method. In addition to supporting the *convention* that block objects need a `toHtml` method, Magento 2 enforces a `toHtml` method by having all blocks implement this `Magento\Framework\View\Element\BlockInterface` interface.

Creating your Own Block Class.

Now that we know what a block class is, lets try creating one of our own. We'll name our block class `Pulsestorm\Nofrillslayout\Block\Chapter1\Hello2`. So, to start, let's change our execute method to use our new block class.

```
1 #File: app/code/Pulsestorm/Nofrillslayout/Controller/
      Chapter1/Index.php
2 public function execute()
3 {
4     $block = new \Pulsestorm\Nofrillslayout\Block\Chapter1\
          Hello2;
5     echo $block->toHtml();
6 }
```

If we reload our URL with the above in place, we'll see an error that looks something like this

> Fatal error: Class 'Pulsestorm12' not found in /path/to/magento/app/code/Pulsestorm/Nofrillslayout/Controller/Chapter1/Index.php on line 17

This is PHP telling us we tried to use a class (`Pulsestorm\Nofrillslayout\Block\Chapter1\Hello2`) with no definition file. Let's fix that by defining our class. Create the following file in the following location

```
1  #File: app/code/Pulsestorm/Nofrillslayout/Block/Chapter1/
       Hello2.php
2  <?php
3  namespace Pulsestorm\Nofrillslayout\Block\Chapter1;
4  class Hello2 implements \Magento\Framework\View\Element\
       BlockInterface
5  {
6      public function toHtml()
7      {
8          return '<p>Hello World from <b>our own</b> Block</p
               >';
9      }
10 }
```

This is almost exactly the same definition file as the `Pulsestorm\Nofrillslayout\Block\Chapter1\Hello` class – the only exceptions are

1. The class's short name is `Hello2`
2. We're outputting a different HTML string

After you've done the above, reload the page and you should see your new Hello World message.

The Magento Object Manager

Before we move on, we need to discuss another object oriented programming feature of Magento 2 – something called the Object Manager. The Object Manager is a special object Magento uses to replace PHP's `new` keyword.

In normal PHP code, you instantiate an object like this

```
1  $block = new \Pulsestorm\Nofrillslayout\Block\Chapter1\
       Hello2;
```

When Magento 2 instantiates an object, the code looks more like this

```
1  $block = $objectManager->create('Pulsestorm\Nofrillslayout\
       Block\Chapter1\Hello2');
```

The reasons for this are myriad, but from a high level point of view, by requiring system developers to use an object manager, Magento gives their objects a number of special properties that PHP objects don't have. One *negative* consequence of this is many of the Magento core classes are difficult and/or impossible to use **without** the object manager. For example, if you tried to create a new class that extends a base Magento class like this

```
<?php
namespace Pulsestorm\Nofrillslayout\Block;
class SomeClass extends \Magento\Framework\View\Element\
    Template
{
}
```

and then instantiated that class with the `new` keyword

```
$block = new \Pulsestorm\Nofrillslayout\Block\SomeClass;
```

You'd end up with a PHP error that looked something like this

> 1 exception(s): Exception #0 (Exception): Recoverable Error: Argument 1 passed to Magento::__construct() must be an instance of Magento, none given

The specific reasons for this are a bit beyond the scope of this book, but if you're interested in learning more *The Magento 2 Object System*

http://alanstorm.com/category/magento#magento_2_object_system

is a great place to get started. There's also a Dependency Injection appendix in the back of this book that covers adjacent topics.

For our immediate purposes here's all you need to know. The base controller file in the `Pulsestorm_Nofrillslayout` module has a special `getObjectManager` method. This method will return an instance of the Magento object manager for you. From this point forward, all our code is going to use this object manager to create objects. In a normal Magento project you'll rarely need to use the object manager – but in order to teach you why that's true, we'll need to use the object manager. Programming is crazy like that sometimes.

So, when you see code like this

```
$objectManager = $this->getObjectManager();
$block         =
    $objectManager->create('Pulsestorm\Nofrillslayout\Block
        \SomeClass');
```

all you need to know is this code instantiates a `Pulsestorm\Nofrillslayout\Block\SomeClass` object.

Creating Text and Template Blocks

Lets get back to Magento blocks. Hard coding HTML or text values in a `toHtml` method isn't a great best practice. Fortunately, Magento 2 ships with two block types that can help us work around this: Text blocks, and Template blocks.

First, let's talk about text blocks, and by talk about, we mean use! Change the Chapter 1 controller so it matches the following

```
1  #File: app/code/Pulsestorm/Nofrillslayout/Controller/
       Chapter1/Index.php
2  public function execute()
3  {
4      $objectManager = $this->getObjectManager();
5      $block         = $objectManager->get('Magento\Framework
           \View\Element\Text');
6      $block->setText('Hello text block.');
7      echo $block->toHtml();
8  }
```

With the above code in place, reload the page and you should see a page that contains the `Hello text Block.` text.

What we've done above is use Magento's Object Manager to create a `Magento\Framework\View\Element\Text` object, use that object's `setText` method to set the text on the block, and then called that block's `toHtml` method. When a `Magento\Framework\View\Element\Text` object renders itself, it looks for the value set with `setText`, and renders that value as HTML text. While the above example is simple, a text block gives you the ability to create a block that renders **any** text you'd like, from any source.

Template Blocks

While Text Blocks are powerful, the real workhorse of Magento 2 layouts are template blocks. Template blocks are similar to text blocks in that they allow you to render any HTML you want – however, rather than render arbitrary strings set via PHP code, a template block will use a `phtml` PHP template file for that rendering. Lets give template blocks a try.

First, we'll create our `.phtml` template file.

```
1  #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
       templates/chapter1/user/hello.phtml
2  <h1>Hello Template!</h1>
```

Then, we'll change our controller action file to instantiate a template object, assign that template object a path, and then render that template object

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter1/Index.php
public function execute()
{
    $objectManager = $this->getObjectManager();
    $block         = $objectManager->get('Magento\Framework
        \View\Element\Template');
    $block->setTemplate('Pulsestorm_Nofrillslayout::
        chapter1/user/hello.phtml');
    echo $block->toHtml();
}
```

Reload the URL, and you should see the rendered text from your `.phtml` template.

Congratulations, you just rendered your first Magento 2 template block! There's a little more to unpack here that in our previous code samples, so lets get to it.

First off, there's our `.phtml` template file.

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    templates/chapter1/user/hello.phtml
<h1>Hello Template!</h1>
```

In Magento 2, a module's `phtml` template files ship **with the module**. Every Magento 2 module can have a `view` folder

```
app/code/Pulsestorm/Nofrillslayout/view
```

This folder contains assets related to rendering a module's front end HTML, CSS, and Javascript. The next portion of a view's path

```
frontend
```

should be the *Magento Area* you want to use the view in. Areas are a tricky topic to cover in full, but for the purposes of this book, the `frontend` area means the user facing Shopping Cart application, and the `adminhtml` area means the backend Magento Admin Console. You can also checkout the areas appendix in the back of this book.

The third portion of a template's path is the word `templates`. The folder indicates the type of view asset inside the folder.

Finally, within the `templates` folder, you can use any directory hierarchy you'd like to organize your `.phtml` templates. We've chosen to place our template in the `chapter1/user` folder – you'd probably choose something else for a real module.

Next is the code that instantiates and renders the template block.

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter1/Index.php

$objectManager = $this->getObjectManager();
$block         = $objectManager->get('Magento\Framework\
    View\Element\Template');
$block->setTemplate('Pulsestorm_Nofrillslayout::chapter1/
    user/hello.phtml');
echo $block->toHtml();
```

The `Magento\Framework\View\Element\Template` class is Magento's base template class. All Magento 2 blocks inherit from this base template class. The bit of code we're most interested in though is this one

```
$block->setTemplate('Pulsestorm_Nofrillslayout::user/hello.
    phtml');
```

A Magento 2 programmer uses the `setTemplate` method to tell Magento where a block's template is located – the `Pulsestorm_Nofrillslayout::chapter1/user/hello.phtml` syntax (A "Template URN") contains two parts – the first is the module where your template is. Behind the scenes Magento will

1. Expand this module name into your module's `view` folder (`app/code/Pulsestorm/Nofrillslayout/view`)
2. Append the current area (`frontend`)
3. Append the word template (`template`)
4. Append the file path on the *right* side of the `::`.

So, with `Pulsestorm_Nofrillslayout::user/chapter1/hello.phtml`, Magento uses those four parts

- `app/code/Pulsestorm/Nofrillslayout/view`
- `frontend`
- `template`
- `Pulsestorm_Nofrillslayout::chapter1/user/hello.phtml`

crete a full template path of app/code/Pulsestorm/Nofrillslayout/view/frontend/template/chapter1/user/hello.phtml

Creating your Own Template Blocks

The examples in the last section used Text and Template blocks directly. While this is possible in Magento 2, it's far more common (and useful) for a developer to create their own block classes that *extend* these base block classes.

For example, if you create the following PHP class definition file

```
#File: app/code/Pulsestorm/Nofrillslayout/Block/Chapter1/
    User/Template.php
<?php
namespace Pulsestorm\Nofrillslayout\Block\Chapter1\User;
class Template extends \Magento\Framework\View\Element\
    Template
{

}
```

you'll be able to use the `Pulsestorm\Nofrillslayout\Block\Chapter1\User\Template` block class in your own code.

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter1/Index.php
public function execute()
{
    $objectManager = $this->getObjectManager();
    $block         = $objectManager->get('Pulsestorm\
        Nofrillslayout\Block\Chapter1\User\Template');
    $block->setTemplate('Pulsestorm_Nofrillslayout::
        chapter1/user/hello.phtml');
    echo $block->toHtml();
}
```

This probably seems like an unneeded complication for our simple hello world output, but its the real power of this approach becomes apparent when we add a method definition to our block object.

```
#File: app/code/Pulsestorm/Nofrillslayout/Block/Chapter1/
    User/Template.php
<?php
namespace Pulsestorm\Nofrillslayout\Block\Chapter1\User;
class Template extends Magento\Framework\View\Element\
    Template
{
    public function getFish()
    {
        return ['one fish', 'two fish', 'red fish', 'blue
            fish'];
    }
}
```

and then **use that method from our template**.

```
<?php #app/code/Pulsestorm/Nofrillslayout/view/frontend/
    templates/chapter1/user/hello.phtml ?>
```

```
2
3  <?php
4      $all_fish = $block->getFish();
5  ?>
6  <h1>Hello Fish!</h1>
7  <ul>
8  <?php foreach($all_fish as $fish):?>
9      <li><?php   echo $block->escapeHtml($fish); ?></li>
10 <?php endforeach; ?>
11 </ul>
```

If you make the above change to your `Template.php` and `hello.phtml` and then reload the page, you should see the following content rendered.

```
1  Hello Fish!
2
3  - one fish
4  - two fish
5  - red fish
6  - blue fish
```

The above code introduces a few new concepts, so lets cover them one by one. First off, we added a PHP method to our block class

```
1  #File: app/code/Pulsestorm/Nofrillslayout/Block/Chapter1/
       User/Template.php
2  public function getFish()
3  {
4      return ['one fish', 'two fish', 'red fish', 'blue fish'
           ];
5  }
```

This method returns an array that contains four strings. Next, in our template file, we **called this method** to fetch the list of fish

```
1  <?php
2      $all_fish = $block->getFish();
3  ?>
```

A `.phtml` template file is just a PHP file – you can write **any** sort of PHP code you'd like in these templates. All Magento `.phtml` files used by the layout system are tied to a specific block class. When you say `$block->getFish()`, you're calling the `getFish` method on the template's block class.

Once we've fetch the array of fish, we use standard PHP to loop through the array, and echo out each string element as an unordered list.

```
1  <ul>
2  <?php foreach($all_fish as $fish):?>
```

```
3       <li><?php   echo $block->escapeHtml($fish); ?></li>
4   <?php endforeach; ?>
5   </ul>
```

The only non-core PHP code in this block is the call to the block's `escapeHtml` method. All untrusted output should be passed through the `$block->escapeHtml` method when you're rendering a `.phtml` template. This will take care of escaping HTML such that you're safe from XSS style attacks.

The `escapeHtml` method is similar to our `getFish` method, in that it's a method defined on our block object. However, this method is defined on the base template block's parent class (a `Magento\Framework\View\Element\AbstractBlock`).

```
1   #File: vendor/magento/framework/View/Element/AbstractBlock.
        php
2   public function escapeHtml($data, $allowedTags = null)
3   {
4       return $this->_escaper->escapeHtml($data, $allowedTags)
            ;
5   }
```

Generally speaking, the code in a Block class has two purposes – first, it's used to fetch data that a template might want to display (using Magento 2's model layer, or however else you might want to fetch data). Second, they provide helper methods (like `escapeHtml`) so that `phtml` template programmers don't need to write overly complicated template code. While you *can* write anything in a `phtml` template, the idea is to keep it simple: Call block methods to fetch data, loop over it, and add the occasional conditional `if` before `echo`ing the content out.

Blocks can make all the difference in the world when/if you need to refactor some a complicated bit of template logic, or you need to hand off the templates to a great front end developer who may not understand the underlying implementation details of where the data used by the template comes from.

Magento 2 vs. Magento 1: $this and $block

There's one last thing to mention about `phtml` templates before we move on. Magento 2 introduces a **new** `$block` variable into every template. This variable is *almost* interchangeable with `$this` – i.e. you use it to call your block's methods

```
1   <?php #app/code/Pulsestorm/Nofrillslayout/view/frontend/
        templates/chapter1/user/hello.phtml ?>
2
3   <?php
4       $all_fish = $block->getFish();
5       $all_fish = $this->getFish();
```

```
6 ?>
```

For the remainder of this book we're going to stick to the `$block` convention.

The technical reasons for this are a bit beyond the scope of this book, but at a high level Magento 2 introduced a **new** object (`Magento\Framework\View\TemplateEngine\Php`) that does the actual HTML rendering – i.e the `include` statement for a block is no longer in the `AbstractBlock` class – it's in the `Magento\Framework\View\TemplateEngine\Php` class. If you didn't follow the last paragraph, don't worry, it's mainly intended for folks familiar with Magento 1's internals to help explain the `$block` vs `$this` difference.

Parent/Child Blocks

Now that we have a better understanding of what a block object is, we need to discuss how Magento typically uses block objects on a page. Whenever you want Magento to return an HTML page from a controller, Magento's core code will instantiate a *layout* object. This object keeps track of which blocks are needed for a particular page, and is the object Magento uses to render the blocks. The layout object also organizes blocks into a tree structure. A simplified example of that might look something like this

```
1 - root
2     - sidebar block
3     - content block
4         - main section section
5     - footer block
```

There's a root block at the top – this block has three child blocks (sidebar, content, footer). The content block has a main section block. Real Magento page requests may have dozens, sometimes hundreds, of individual blocks with parent/child hierarchies that go many layers deep. This allows Magento core developers to isolate a specific bit of HTML and template logic from the rest of the page.

If that didn't make sense, some code should make it clearer. Replace our controller action with the following code.

```
1 #File: app/code/Pulsestorm/Nofrillslayout/Controller/
      Chapter1/Index.php
2 public function execute()
3 {
4     $objectManager      = $this->getObjectManager();
5     $layout             = $objectManager->get('Magento\
          Framework\View\Layout');
6     $block              = $layout->createBlock('Magento\
          Framework\View\Element\Text');
```

```
7       $block->setText('<h1>Hello Layout!</h1>');
8       echo $block->toHtml();
9
10  }
```

If you reload our URL with the above code in place, you should see the "Hello Layout!" from our text block rendered.

This code introduces two new concepts. First, there's the Layout object itself, and second, there's the layout object's `createBlock` factory method.

The Layout object (instantiated from a `Magento\Framework\View\Layout` class) is the main object Magento's core code uses to create blocks. For Magento 1 developers, this object is analogous to the old `Mage::getSingleton(core/layout)` object. This object has a `createBlock` method. The system usually uses the `createBlock` method (rather than direct use of the object manager) to instantiate block objects. It's also necessary to use the `createBlock` method for your block if you want them to have parent/child relationships.

Next, we're going to create two blocks with a parent child relationship. First, let's create the blocks. Replace your controller action code with the following

```
1   #File: app/code/Pulsestorm/Nofrillslayout/Controller/
        Chapter1/Index.php
2   public function execute()
3   {
4       $objectManager = $this->getObjectManager();
5       $layout        = $objectManager->get('Magento\Framework
            \View\Layout');
6       $blockParent   = $layout->createBlock('Magento\
            Framework\View\Element\Template');
7       $blockParent->setTemplate('Pulsestorm_Nofrillslayout::
            chapter1/parent.phtml');
8
9       $blockChild    = $layout->createBlock('Magento\
            Framework\View\Element\Template');
10      $blockChild->setTemplate('Pulsestorm_Nofrillslayout::
            chapter1/child.phtml');
11
12      echo $blockParent->toHtml();
13      echo $blockChild->toHtml();
14  }
```

So far, there's nothing new here. All we're doing is instantiating two blocks, assigning them templates, and then rendering each block individually. To make one block a child of another block you need to **append** that block to the first. Give the following code a try

```
1   #File: app/code/Pulsestorm/Nofrillslayout/Controller/
        Chapter1/Index.php
```

```
2  public function execute()
3  {
4      $objectManager = $this->getObjectManager();
5      $layout        = $objectManager->get('Magento\Framework
           \View\Layout');
6      $blockParent   = $layout->createBlock('Magento\
           Framework\View\Element\Template');
7      $blockParent->setTemplate('Pulsestorm_Nofrillslayout::
           chapter1/parent.phtml');
8
9      $blockChild    = $layout->createBlock('Magento\
           Framework\View\Element\Template');
10     $blockChild->setTemplate('Pulsestorm_Nofrillslayout::
           chapter1/child.phtml');
11
12     $blockParent->append($blockChild);
13     echo $blockParent->toHtml();
14     #echo $blockChild->toHtml();
15 }
```

The code above is almost identical to the previous example, except we've

1. Commented out the `echo` for the child block
2. Used the parent block's `append` method to add the child block to the parent block

If you reload the page, **both** blocks are still printed out, even though we're only echoing the first block.

Rendering a Child Block

The above examples used templates we prepared for this book. Let's try building a parent/child block relationship from scratch. First, we'll create a parent template and two children.

```
1  #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
       templates/chapter1/user/parent.phtml
2  <h1>Our Own Parent Block</h1>
3
4  #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
       templates/chapter1/user/child1.phtml
5  <p>Some people think having one child is enough</p>
6
7  #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
       templates/chapter1/user/child2.phtml
8  <p>Other people think a second child can keep the first
       company.</p>
```

Then, let's change our controller code to create the blocks, append the children, and `echo` the parent.

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter1/Index.php
public function execute()
{
    $objectManager = $this->getObjectManager();
    $layout        = $objectManager->get('Magento\Framework
        \View\Layout');
    $blockParent   = $layout->createBlock('Magento\
        Framework\View\Element\Template');
    $blockParent->setTemplate('Pulsestorm_Nofrillslayout::
        chapter1/user/parent.phtml');

    $blockChild1    = $layout->createBlock('Magento\
        Framework\View\Element\Template');
    $blockChild1->setTemplate('Pulsestorm_Nofrillslayout::
        chapter1/user/child1.phtml');

    $blockChild2    = $layout->createBlock('Magento\
        Framework\View\Element\Template');
    $blockChild2->setTemplate('Pulsestorm_Nofrillslayout::
        chapter1/user/child2.phtml');

    $blockParent->append($blockChild1);
    $blockParent->append($blockChild2);
    echo $blockParent->toHtml();
}
```

With the above code in place, if we reload our URL we'll see

```
Our Own Parent Block
```

Huh. That's unexpected. **Only** the parent block rendered, even though we added the two child blocks. What gives?

A parent block **does not** echo out its children automatically. There's one additional thing we'll need to do to make this happen. Find your parent template file, and add the following code to it.

```
<h1>Our Own Parent Block</h1>
<?php
    echo $this->getChildHtml();
?>
```

The `getChildHtml` method will render **all** of a parent's children blocks. Reload the URL with the above code in place, and you should see both your children rendered.

```
Our Own Parent Block

Some people think having one child is enough

Other people think a second child can keep the first
    company.
```

Once you've digested the above, you may be wondering if it's possible to *selectively* render a specific child block. This is possible, but we **need to assign a name to our blocks** when we create them. Let's change our controller code so it matches the following

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter1/Index.php
public function execute()
{
    $objectManager = $this->getObjectManager();
    $layout        = $objectManager->get('Magento\Framework
        \View\Layout');
    $blockParent   = $layout->createBlock(
        'Magento\Framework\View\Element\Template',
        'pulsestorm_nofrills_parent'
    );
    $blockParent->setTemplate('Pulsestorm_Nofrillslayout::
        chapter1/user/parent.phtml');

    $blockChild1    = $layout->createBlock(
        'Magento\Framework\View\Element\Template',
        'pulsestorm_nofrills_child1'
    );
    $blockChild1->setTemplate('Pulsestorm_Nofrillslayout::
        chapter1/user/child1.phtml');

    $blockChild2    = $layout->createBlock(
        'Magento\Framework\View\Element\Template',
        'pulsestorm_nofrills_child2'
    );
    $blockChild2->setTemplate('Pulsestorm_Nofrillslayout::
        chapter1/user/child2.phtml');

    $blockParent->append($blockChild1);
    $blockParent->append($blockChild2);
    echo $blockParent->toHtml();
    #echo $blockChild->toHtml();
}
```

The big change to the above code is we're passing a second argument to the `createBlock` method.

```
$blockChild2    = $layout->createBlock(
    'Magento\Framework\View\Element\Template',
    'pulsestorm_nofrills_child2'
);
```

This second argument (`pulsestorm_nofrills_child2` above) gives our block a **name** in the layout. Once our blocks have names, we can tell the `getChildHtml` method which block we want to render. If we edit our `parent.phtml` file so it matches the following

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    templates/chapter1/user/parent.phtml
<h1>Our Own Parent Block</h1>
<?php
    echo $this->getChildHtml('pulsestorm_nofrills_child1');
?>
```

and reload our page, we'll see that only the first child rendered.

Once your blocks are named, you can add special conditional logic or rules around which child blocks (if any) render for a particular template.

Magento 2: Block Containers and the Layout Structure

So far, much of what we've talked about has been a retread of Magento 1 features and concepts. Magento 2's new object system may have changed the syntax of what we're doing, but conceptually it's very similar to Magento 1. In this section we're going to cover two new features of the Magento 2 layout system – containers and the structure.

A container is conceptually similar to a block, in that it's a node in the layout object. However, (unlike a block), a container

1. Has no associated class file
2. Has no template
3. Has no content

A container's *only* job in Magento 2 is to hold a reference to other block objects, and then render each and every block added to it. For Magento 1 developers, a container performs the same job as the old `text/list` blocks.

With the restrictions above, programmatically interacting with containers becomes a bit trickier than interacting with blocks. How can we append a container to another block if we can't instantiate a container? Also – how do we add the *root* container that sits at the top of the layout tree?

In an attempt to improve the layout system in Magento 2, the core developers inadvertently introduced a new layer of complexity into the rendering of a layout object. In order to deal with this complexity, the layout object uses a special structure object to keep track of how blocks, containers, and other elements are related.

Normally, you'll never need to interact with this structure object. In fact, Magento's built the system in such a way that it's very *difficult* to deal directly with a structure object. The `Pulsestorm_Nofrillslayout` module that ships with this book contains a special class preference (see the Dependency Injection appendix for more information on class preferences) that will let us get at and use the structure object. We don't recommend using this in production code, but as a teaching aid it will be invaluable.

OK, so that's a lot of new concepts all at once. Let's see if we can clarify things a bit with some code.

First, let's swap out our controller action with the following.

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter1/Index.php
public function execute()
{
    $objectManager = $this->getObjectManager();
    $layout        = $objectManager->get('Magento\Framework
        \View\Layout');

    $blockParent   = $layout->createBlock(
        'Magento\Framework\View\Element\Template',
        'pulsestorm_nofrills_parent'
    );
    $blockParent->setTemplate('Pulsestorm_Nofrillslayout::
        chapter1/user/parent.phtml');
    $blockChild1    = $layout->createBlock(
        'Magento\Framework\View\Element\Template',
        'pulsestorm_nofrills_child1'
    );
    $blockChild1->setTemplate('Pulsestorm_Nofrillslayout::
        chapter1/user/child1.phtml');
    $blockParent->append($blockChild1);

    echo $blockParent->toHtml();
}
```

This is a rehash of our code from above. We've created a parent block, added a child block to the parent, and then `echo`d the content. Once you've confirmed the above code works, let's replace the `echo $blockParent->toHtml();` so our action looks like this

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter1/Index.php
public function execute()
{
    $objectManager = $this->getObjectManager();
    $layout        = $objectManager->get('Magento\Framework
        \View\Layout');

    $blockParent   = $layout->createBlock(
        'Magento\Framework\View\Element\Template',
        'pulsestorm_nofrills_parent'
    );
    $blockParent->setTemplate('Pulsestorm_Nofrillslayout::
        chapter1/user/parent.phtml');
    $blockChild1    = $layout->createBlock(
        'Magento\Framework\View\Element\Template',
        'pulsestorm_nofrills_child1'
    );
    $blockChild1->setTemplate('Pulsestorm_Nofrillslayout::
        chapter1/user/child1.phtml');
    $blockParent->append($blockChild1);

    $layout->addContainer('top', 'The top level container')
        ;

    // Magento\Framework\View\Layout\Data\Structure
    $structure = $layout->getStructure(); //note: not
        standard magento
    $structure->setAsChild('pulsestorm_nofrills_parent', '
        top');

    $layout->generateElements();
    echo $layout->getOutput();
}
```

If you reload with the above code in place, you should see your blocks rendered out correctly, despite the fact we never called `toHtml` on the parent block. The new lines we added

1. Added a root level container named `top`
2. Fetched the layout's structure manager object (`Magento\Framework\View\Layout\Data\Structure`)
3. Used the structure manager to set our parent block as a child of the `top` container
4. Used the layout object's `getOutput` method to render the entire layout

Conceptually, that's a lot to take in. Let's review each of the above in a little more detail.

First off, the following line

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter1/Index.php

$layout->addContainer('top', 'The top level container');
```

Uses the Layout Object's `addContainer` method to add a "root level" container to the layout object. At this point, our layout looks like the following

```
- top (a container)
```

That's it – just a top level container. If we tried rendering our layout at this point, we'd just get a blank string since our layout is empty. Next, we need to **add a block** to our container.

Unfortunately – the Magento layout object doesn't have a publicly accessible method for adding a block to a container. This may make sense for the core team's goals – but it gets in the way of our goals: teaching you how the layout system works. That's why this next line

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter1/Index.php

// Magento\Framework\View\Layout\Data\Structure
$structure = $layout->getStructure(); //note: not standard
    magento
```

has the "not standard Magento" warning. We've used Magento's flexible object system to give the layout object a `getStructure` method. This will return the (standard Magento) `Magento\Framework\View\Layout\Data\Structure` object. The structure object is the object that keeps track of the parent/child relationships between blocks and containers in a Magento layout.

The structure object has a `setAsChild` method. This method allows you to tell Magento

> Make this already-created block a child of a particular container

So when we said

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter1/Index.php

$structure->setAsChild('pulsestorm_nofrills_parent', 'top')
    ;
```

This was us telling Magento

> Make the already created `pulsestorm_nofrills_parent` block a child of the container named `top`

At this point, our layout looks like this

```
- top (a container)
    - pulsestorm_nofrills_parent (a block)
        - pulsestorm_nofrills_child1 (a block)
```

We've added a root level container named `top` and added the block named `pulsestorm_nofrills_parent` to this container. The `pulsestorm_nofrills_child1` was already a child of `pulsestorm_nofrills_parent` from our previous layout manipulations.

Our final step is outputting the layout. This is two lines of code

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter1/Index.php

$layout->generateElements();
echo $layout->getOutput();
```

First, we call `generateElements` on the layout object. We'll talk more about this method in later chapters. For now, just think of it as a initialization method for the layout object. This is us telling Magento

> Hey Magento, get ready, we're about to output the layout

The second line calls the layout object's `getOutput` method. This is us telling Magento

> Hey Magento, please give us a final rendered string for this layout object.

When you call `getOutput` on the layout object, Magento will

1. Find every root level container added with `addContainer`
2. Render *every top level block* inside that container.

So, in our specific case, Magento finds a root level container named `top`. Inside of top it finds a single block named `pulsestorm_nofrills_parent`, and then calls this block's `toHtml` method.

Wrap Up

OK! Those were lots of words, and some of you may be thinking this is a *crazy* way to build page layouts. Fortunately, someone who worked at Magento at one point in time agrees with you. While building layouts in PHP makes it easier

to see their structure, for day-to-day HTML work, Magento created its Layout XML system.

In our next chapter, we're going to take a look at using this layout XML system to create block and page structures identical to what we did above – all without writing a single line of new PHP code.

Chapter 3

Building Layouts via XML

In Chapter 1, we spent a lot of time using PHP code to create block objects. Towards the end, you may have noticed the PHP code involved was growing in both size and complexity. Depending on your comfort level and expertise with PHP, this was somewhere between a "mild annoyance" and an "occasion to go into the stairwell and cry".

The need to expose non-expert programmers to this much systems level code is a common problem in programmatic systems. One industry solution to this problem is to use something called a *Domain Specific Language*, or DSL. Domain Specific Languages are still programming languages, but they

1. Offer reduced syntax and functionality when compared to a regular programming language
2. Are designed to solve a specific problem, sometimes called the problem domain
3. Are often simple enough to implement in a configuration based language (YAML, JSON, XML)

While you *can* create Magento page layouts directly with PHP code, Magento offers an XML based DSL they'd prefer you use. This XML based language allows a front end developer to create and control their HTML layouts without writing a single line of PHP code. In this chapter, we'll start to explore this XML based language and how it compares to the PHP based code we've used so far.

Starting with Containers

Like chapter 1, chapter 2 has a pre-built controller file that's ready to go. It should have an execute method that looks like the following

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter2/Index.php
public function execute()
{
    $objectManager = $this->getObjectManager();
    $layout        = $objectManager->get('Magento\Framework
        \View\Layout');
    $layout->addContainer('top', 'The top level container')
        ;

    $blockOne    = $layout->createBlock(
        'Magento\Framework\View\Element\Template',
        'pulsestorm_nofrills_chapter2_block1'
    );
    $blockOne->setTemplate('Pulsestorm_Nofrillslayout::
        chapter2/block1.phtml');

    $blockTwo    = $layout->createBlock(
        'Magento\Framework\View\Element\Template',
        'pulsestorm_nofrills_chapter2_block2'
    );
    $blockTwo->setTemplate('Pulsestorm_Nofrillslayout::
        chapter2/block2.phtml');

    $structure = $layout->getStructure(); //note: not
        standard magento
    $structure->setAsChild('
        pulsestorm_nofrills_chapter2_block1', 'top');
    $structure->setAsChild('
        pulsestorm_nofrills_chapter2_block2', 'top');

    $layout->generateElements();
    echo $layout->getOutput();
}
```

Everything in the above `execute` was covered in chapter 1. If you load the following URL in your Magento system

```
http://magento.example.com/pulsestorm_nofrillslayout/
    chapter2
```

you should see output that looks similar to the following

```
## First!

This is the first block. It's the loneliest block.

## Second!

```

```
7 This is the second block. It's the loneliest block since
      the
8 block pulsestorm_nofrills_chapter2_block1.
```

Let's replace our `execute` method with the following code.

```
1  #File: app/code/Pulsestorm/Nofrillslayout/Controller/
       Chapter2/Index.php
2  public function execute()
3  {
4      $objectManager = $this->getObjectManager();
5      $layout        = $objectManager->get('Magento\Framework
           \View\Layout');
6
7      //START new code
8      $updateManager = $layout->getUpdate();
9      $updateManager->addUpdate(
10         '<container name="top"></container>'
11     );
12     //END   new code
13
14     $blockOne   = $layout->createBlock(
15         'Magento\Framework\View\Element\Template',
16         'pulsestorm_nofrills_chapter2_block1'
17     );
18     $blockOne->setTemplate('Pulsestorm_Nofrillslayout::
           chapter2/block1.phtml');
19
20     $blockTwo    = $layout->createBlock(
21         'Magento\Framework\View\Element\Template',
22         'pulsestorm_nofrills_chapter2_block2'
23     );
24     $blockTwo->setTemplate('Pulsestorm_Nofrillslayout::
           chapter2/block2.phtml');
25
26     $structure = $layout->getStructure(); //note: not
           standard magento
27     $structure->setAsChild('
           pulsestorm_nofrills_chapter2_block1', 'top');
28     $structure->setAsChild('
           pulsestorm_nofrills_chapter2_block2', 'top');
29
30     //START new code (commenting out structure)
31     //$layout->generateElements();
32     //END   new code
33     echo $layout->getOutput();
34 }
```

The biggest change you'll notice here is we've removed the line which creates

our container,

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter2/Index.php

$layout->addContainer('top', 'The top level container');
```

and replaced it with the following.

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter2/Index.php

//START new code
$updateManager = $layout->getUpdate();
$updateManager->addUpdate(
    '<container name="top"></container>'
);
//END   new code
```

This is our first example of Magento's domain specific language for creating layouts. Clear your cache, reload the page – and everything should still look the same, despite our change!

Updating the Layout

Before we can explain the new XML, we'll need to talk a little more PHP.

The following code fetches a special object called the "Update Manager" from the layout object.

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter2/Index.php

/**
 * @var Magento\Framework\View\Model\Layout\Merge
 */
$updateManager = $layout->getUpdate();
```

This object is responsible for keeping track of the changes a client programmer wants to make to the layout. i.e. It manages the XML updates.

Next, we used the update manager's `addUpdate` method to update the layout.

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter2/Index.php

$updateManager->addUpdate(
```

```
4       '<container name="top" label="The top level container
            "></container>'
5   );
```

A *Layout Update* is a chunk of XML. Unless you have a deep software engineering background, you probably think of XML as a configuration language. A layout update isn't, strictly speaking, configuration. Instead, it is code written in a Domain Specific programming Language (DSL). When you say

```
1   #File: app/code/Pulsestorm/Nofrillslayout/Controller/
        Chapter2/Index.php
2
3   <container name="top" label="The top level container"></
        container>
```

Magento will, behind the scenes, transform this XML into PHP code that looks like (or, for the sticklers, behaves like) the following

```
1   $objectManager = $this->getObjectManager();
2   $layout        = $objectManager->get('Magento\Framework\
        View\Layout');
3   $layout->addContainer('top', 'The top level container');
```

Domain specific programming languages exist to limit the number of things you can do with a system, reduce the complexity of a system, and to allow non-or-new programmers the ability to to write code that does simple-but-important things.

Ideally, you shouldn't need to think about the PHP code generated by the DSL – however, understanding the basics of how a DSL works is useful when debugging DSL code.

Before we move on, you may be wondering why we commented out `generateElements`.

```
1   //START new code (commenting out structure)
2   //$layout->generateElements();
3   //END   new code
```

We will need to beg off discussing `generateElements` for the time being. We'll discuss this method more in chapter six.

Creating Blocks with Updates

Layout updates can do more than create containers. Change your controller `execute` method so it matches the following.

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter2/Index.php
public function execute()
{
    $objectManager = $this->getObjectManager();
    $layout        = $objectManager->get('Magento\Framework
        \View\Layout');

    $updateManager = $layout->getUpdate();

    $updateManager->addUpdate(
        '<container name="top"></container>'
    );

    $updateManager->addUpdate(
        '<referenceContainer name="top">
            <block
                class="Magento\Framework\View\Element\
                    Template"
                name="pulsestorm_nofrills_chapter2_block1"
                template="Pulsestorm_Nofrillslayout::
                    chapter2/block1.phtml">
            </block>
        </referenceContainer>'
    );

    echo $layout->getOutput();
}
```

You'll notice we've removed the PHP code that instantiated the two blocks and added those blocks to the `top` container. Instead, we've added a second chunk of XML to the update manager via its `addUpdate` method.

If you **clear Magento's cache** and reload the page, you should see a single rendered block

```
#First!

This is the first block. It's the loneliest block.
```

If we take a closer look at our layout update XML

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter2/Index.php

<referenceContainer name="top">
    <block
        class="Magento\Framework\View\Element\Template"
        name="pulsestorm_nofrills_chapter2_block1"
```

```
7           template="Pulsestorm_Nofrillslayout::chapter2/
                block1.phtml">
8       </block>
9   </referenceContainer>
```

we'll see two new tags – `referenceContainer` and `block`. In plain english, this XML tells Magento

> For each block tag inside the container tag, instantiate a new block object, and then add those blocks to the container named `top`

Or, in PHP pseudo code

```
1  $container = $layout->getContainer('top');
2  $block     = new Magento\Framework\View\Element\Template;
3  $block  ->setNameInLayout('
       pulsestorm_nofrills_chapter2_block1')
4          ->setTemplate('Pulsestorm_Nofrillslayout::chapter2/
               block1.phtml');
5  $container->addBlock($block);
```

The `pulsestorm_nofrills_chapter2_block1` block gets added to the container top because it's **inside** the `referenceContainer` tag.

Taking a closer look at the block XML itself

```
1  #File: app/code/Pulsestorm/Nofrillslayout/Controller/
       Chapter2/Index.php
2
3  <block
4      class="Magento\Framework\View\Element\Template"
5      name="pulsestorm_nofrills_chapter2_block1"
6      template="Pulsestorm_Nofrillslayout::chapter2/block1.
           phtml">
7  </block>
```

The `class` attribute tells magento which PHP class to use when instantiating the block (same as the class we used in the object manager earlier). The `name` attribute gives the block a unique identifier in the layout (same as the second argument to `createBlock`), and the `template` attribute sets a template URN (same as the `setTemplate` method we used earlier).

Before we continue, there's a bit of pedagogical infrastructure we need to address: Namely – let's get our XML strings into stand-alone files.

Moving to Files

For readers with programming experience, you're probably a little weirded out by our use of XML in PHP string literals. This is, generally speaking, a bad default practice. Let's remove those strings and move our XML updates to files.

First, we'll create two XML files with the following contents for our updates

```
#File: app/code/Pulsestorm/Nofrillslayout/sample-xml/
    chapter2/user/top-container.xml
<container name="top"></container>

#File: app/code/Pulsestorm/Nofrillslayout/sample-xml/
    chapter2/user/blocks.xml
<referenceContainer name="top">
    <block
        class="Magento\Framework\View\Element\Template"
        name="pulsestorm_nofrills_chapter2_block1"
        template="Pulsestorm_Nofrillslayout::chapter2/
            block1.phtml">
    </block>
</referenceContainer>
```

Next, we'll replace our controller's execute method with the following.

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter2/Index.php

public function execute()
{
    $objectManager = $this->getObjectManager();
    $layout        = $objectManager->get('Magento\Framework
        \View\Layout');
    //$layout->addContainer('top', 'The top level container
        ');
    $updateManager = $layout->getUpdate();

    $container_xml = $this->loadXmlFromSampleXmlFolder('
        chapter2/user/top-container.xml');
    $updateManager->addUpdate($container_xml);

    $block_xml     = $this->loadXmlFromSampleXmlFolder('
        chapter2/user/blocks.xml');
    $updateManager->addUpdate($block_xml);

    // $layout->generateElements();
    echo $layout->getOutput();
}
```

This code should be functionally identical to our previous code. What we've done is replace the XML string literals with calls to the `loadXmlFromSampleXmlFolder` method.

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter2/Index.php

    $container_xml = $this->loadXmlFromSampleXmlFolder('
        chapter2/user/top-container.xml');
    $updateManager->addUpdate($container_xml);
```

The `loadXmlFromSampleXmlFolder` method is not part of stock Magento. It's something we've written to make working through these tutorials a bit easier. Being able to store XML in files means we get all the syntax highlighting, formatting, and searching abilities most text editors and IDEs bring to XML files. Using "raw" PHP string literals makes it too easy to introduce errors that are hard to spot. If you're curious in the implementation of `loadXmlFromSampleXmlFolder`, you can find it here

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/Base.
    php
protected function loadXmlFromSampleXmlFolder($path)
{
    $path = realpath(__DIR__) . '/../sample-xml/'  . $path;
    //using the hated error surpression operator
    //to avoid xsi:type warnings from simple XML
    @$xml = simplexml_load_file($path);
    if(!$xml)
    {
        throw new \Exception("Could not load valid XML from
            $path");
    }
    return str_replace('<?xml version="1.0"?>', '', $xml->
        asXml());
}
```

Alright! With our XML in files, lets return to our programming!

Multiple Blocks

XML updates aren't limited to a single block. Try changing your `blocks.xml` so it looks like this.

```
#File: app/code/Pulsestorm/Nofrillslayout/sample-xml/
    chapter2/user/blocks.xml

<referenceContainer name="top">
```

```
4       <block
5           class="Magento\Framework\View\Element\Template"
6           name="pulsestorm_nofrills_chapter2_block1"
7           template="Pulsestorm_Nofrillslayout::chapter2/
                block1.phtml">
8       </block>
9
10      <!-- START: added this XML -->
11      <block
12          class="Magento\Framework\View\Element\Template"
13          name="pulsestorm_nofrills_chapter2_block2"
14          template="Pulsestorm_Nofrillslayout::chapter2/
                block2.phtml">
15      </block>
16      <!-- END:   added this XML -->
17
18  </referenceContainer>
```

Clear your cache, and reload the page. You should see the second block rendered!

Parent/Child Blocks

You'll remember back in chapter 1 we showed you you parent/child blocks worked.

```
1  $blockParent   = $layout->createBlock('Magento\Framework\
       View\Element\Template');
2  $blockParent->setTemplate('Pulsestorm_Nofrillslayout::
       chapter1/parent.phtml');
3
4  $blockChild    = $layout->createBlock('Magento\Framework\
       View\Element\Template');
5  $blockChild->setTemplate('Pulsestorm_Nofrillslayout::
       chapter1/child.phtml');
6
7  $blockParent->append($blockChild);
```

Parent/child blocks are **also** possible with XML Updates. Give the following a try in `blocks.xml`.

```
1  #File: app/code/Pulsestorm/Nofrillslayout/sample-xml/
       chapter2/user/blocks.xml
2
3  <referenceContainer name="top">
4      <block
5          class="Magento\Framework\View\Element\Template"
6          name="pulsestorm_nofrills_chapter2_parent"
```

```
7           template="Pulsestorm_Nofrillslayout::chapter2/
                parent.phtml">
8               <block
9                   class="Magento\Framework\View\Element\
                        Template"
10                  name="pulsestorm_nofrills_chapter2_child"
11                  template="Pulsestorm_Nofrillslayout::
                        chapter2/child.phtml">
12              </block>
13      </block>
14  </referenceContainer>
```

Notice we've configured the `pulsestorm_nofrills_chapter2_block1` block **as a child node** of the `pulsestorm_nofrills_chapter2_parent` block node. When we nest blocks like this, Magento knows to add the inner blocks as children of the outer blocks.

Introducing Arguments

Another neat feature of the layout DSL is the ability to pass in data arguments to your templates. For example, replace your `blocks.xml` file with the following

```
1   #File: app/code/Pulsestorm/Nofrillslayout/sample-xml/
        chapter2/user/blocks.xml
2   <referenceContainer name="top">
3       <block
4           class="Magento\Framework\View\Element\Template"
5           name="pulsestorm_nofrills_chapter2_hello_argument"
6           template="Pulsestorm_Nofrillslayout::chapter2/user/
                hello-argument.phtml">
7               <arguments>
8                   <argument name="the_message" xsi:type="
                        string">Hello World</argument>
9               </arguments>
10      </block>
11  </referenceContainer>
```

Notice that we've created new `<argument/>` nodes under a new `<arguments/>` node. Next, lets create a template for our new block.

```
1   #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
        templates/chapter2/user/hello-argument.phtml
2   <p>
3       Printed with getData: <?php echo $block->getData('
            the_message'); ?>
4   </p>
5
```

```
<p>
    Printed with magic methods: <?php echo $block->
        getTheMessage(); ?>
</p>
```

Reload the page (after clearing your cache) and you should see output similar to the following

```
Printed with getData: Hello World

Printed with magic methods: Hello World
```

The `<arguments/>` node is deceptively named – its real purpose is to set *data properties* on a block object. Any named `<argument/>` will be available to the block Magento instantiates. Eagle eyed readers may have noticed the `xsi:type` attribute that was part of the argument node.

```
#File: app/code/Pulsestorm/Nofrillslayout/sample-xml/
    chapter2/user/blocks.xml
<argument name="the_message" xsi:type="string">
```

The `xsi:type` node allows you to pass in arguments that are more complex than simple scaler values. For example, you could pass in an array of values using `xsi:type="array"`

```
#File: app/code/Pulsestorm/Nofrillslayout/sample-xml/
    chapter2/user/blocks.xml

<arguments>
    <argument name="the_message" xsi:type="string">Hello
        World</argument>
    <argument name="the_array" xsi:type="array">
        <item name="key" xsi:type="string">value</item>
        <item name="foo" xsi:type="string">var</item>
    </argument>
</arguments>
```

You can even tell Magento to instantiate a new object with `xsi:type="object"`

```
@highlightsyntax@php
#File: app/code/Pulsestorm/Nofrillslayout/sample-xml/
    chapter2/user/blocks.xml

<arguments>
    <!-- ... -->
    <argument name="the_object" xsi:type="object">
        Pulsestorm\Nofrillslayout\Chapter2\Example</
        argument>
```

```
7  </arguments>
```

Try accessing both the array and the object in the `phtml` template files via the `getData` method, or via the block's magic methods (`getTheArray`, `getTheObject`) on your own.

Pulling it All Together

Before we move on, we've prepared one last example for you. In our controller file, let's try loading the prepared `page.xml` file instead of the `user/blocks.xml` file.

```
1  //$block_xml     = $this->loadXmlFromSampleXmlFolder('
       chapter2/user/blocks.xml');
2  $block_xml     = $this->loadXmlFromSampleXmlFolder('
       chapter2/page.xml');
```

Clear your cache, reload the page, and you'll find a crude example of the skeleton for an HTML page layout.

```
1  | Home | About | Etc |
2
3  This is where the content goes.
4
5  © 2017 Acme Widgets
```

If you open up `sample-xml/chapter2/page.xml`, you'll see

1. A root block with a root.phtml template
2. Three child blocks, `navigation`, `content`, and `footer`, each with their own templates
3. A custom template block class (that extends the base Magento template block)
4. An argument to set the content
5. An argument to add links
6. Templates that use custom PHP logic to output their arguments

While overly simple, this same sort of system is how Magento 2 builds the HTML for its own pages. In the coming chapters we'll explore how Magento scales this simple concept to its own, very complex, HTML page layouts.

Chapter 4

Layout Handles

So far in this book, we've been directly echoing output from a controller action. While this works, and is a great way to learn, it's **not** how client programmers are meant to use Magento's system.

Building an entire HTML page *from scratch* does not make for an easy to reuse system. It forces each individual developer to decide how the basic <html/> page skeleton will be stored, rendered, and edited. It forces each individual to decide how common page elements (left sidebar, footer, etc.) should be added to.

When you're working with Magento 2's "according to hoyle" development rules, you'll want to use a *Results Page Object* to create the base HTML page layout. In this chapter we'll explore using a *Results Page Object*, as well as using a system called *Layout Handles* to formally share Magento 2 layout elements.

For Magento 1 developers, result page objects are the end-result of Magento 2's refactoring of the old layout XML system. You'll see many familiar concepts that are implemented quite differently in Magento 2 – i.e. you know just enough to be dangerous, so step carefully!

Returning a Result Page Object

We're going to jump right in. Load the following URL in your browser

```
1 http://magento.example.com/pulsestorm_nofrillslayout/
      chapter3
```

Just like our previous chapters, you'll see output similar to the following.

```
1 string 'Pulsestorm\Nofrillslayout\Controller\Chapter3\Index
      ::execute' (length=60)
```

Lets take a look at that URL's controller file

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter3/Index.php
<?php
namespace Pulsestorm\Nofrillslayout\Controller\Chapter3;
use Pulsestorm\Nofrillslayout\Controller\Base;

class Index extends Base
{
    protected $resultPageFactory;
    public function __construct(
        \Magento\Framework\App\Action\Context $context,
        \Magento\Framework\View\Result\PageFactory
            $resultPageFactory
    )
    {
        $this->resultPageFactory = $resultPageFactory;
        return parent::__construct($context);
    }
    public function execute()
    {
        var_dump(__METHOD__);
    }
}
```

In the `execute` method we can see the simple `var_dump` statement that led to our output, but there's also something new. The `__construct` method uses Magento's automatic constructor dependency injection system to inject a result page factory object (i.e. `Magento\Framework\View\Result\PageFactory`). If you're not familiar with Magento's dependency injection system, checkout the Magento 2 Dependency Injection appendix at the end of this book.

We're going to use the `resultPageFactory` object to create a response object for our controller. Change your controller's execute method so it looks like the following

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter3/Index.php
public function execute()
{
    $pageObject = $this->resultPageFactory->create();
    return $pageObject;
}
```

If you reload with the above page we'll see – another blank page? However, if we view the **source** of that page using our browser's `View Source` feature, or a CLI program like `curl`

```
$ curl 'http://magento.example.com/
    pulsestorm_nofrillslayout/chapter3'
<!doctype html>
<html lang="en-US">
    <head >
        <script>
    var require = {
        "baseUrl": "http://magento.example.com/static/
            frontend/Magento/luma/en_US"
    };
</script>
        <meta charset="utf-8"/>
<meta name="description" content="Default Description"/>
<meta name="keywords" content="Magento, Varien, E-commerce"
    />
<meta name="robots" content="INDEX,FOLLOW"/>
<meta name="viewport" content="width=device-width, initial-
    scale=1, maximum-scale=1.0, user-scalable=no"/>
<title></title>
<link  rel="stylesheet" type="text/css"  media="all" href="
    http://magento.example.com/static/frontend/Magento/luma
    /en_US/mage/calendar.css" />
<link  rel="stylesheet" type="text/css"  media="all" href="
    http://magento.example.com/static/frontend/Magento/luma
    /en_US/css/styles-m.css" />
<link  rel="stylesheet" type="text/css"  media="screen and
    (min-width: 768px)" href="http://magento.example.com/
    static/frontend/Magento/luma/en_US/css/styles-l.css" />
<link  rel="stylesheet" type="text/css"  media="print" href
    ="http://magento.example.com/static/frontend/Magento/
    luma/en_US/css/print.css" />
<link  rel="icon" type="image/x-icon" href="http://magento.
    example.com/static/frontend/Magento/luma/en_US/
    Magento_Theme/favicon.ico" />
<link  rel="shortcut icon" type="image/x-icon" href="http
    ://magento.example.com/static/frontend/Magento/luma/
    en_US/Magento_Theme/favicon.ico" />
<script  type="text/javascript"  src="http://magento.
    example.com/static/frontend/Magento/luma/en_US/
    requirejs/require.js"></script>
<script  type="text/javascript"  src="http://magento.
    example.com/static/frontend/Magento/luma/en_US/
    Pulsestorm_Nofrillslayout/user/example-script.js"></
    script>
<link  rel="stylesheet" type="text/css"  media="all" href="
    http://magento.example.com/media/styles.css" />
                </head>
    <body data-container="body" data-mage-init='{"
        loaderAjax": {}, "loader": { "icon": "http://
        magento.example.com/static/frontend/Magento/luma/
```

```
            en_US/images/loader-2.gif"}}' class="pulsestorm-
            nofrillslayout-chapter3-index page-layout-admin-1
            column">
26              </body>
27  </html>
```

we'll see Magento has added a skeleton HTML page. So, right off the bat, we already have a shared HTML skeleton for all developers to use. This skeleton pulls in Magento's base javascript and CSS, sets up default `<body/>` tag attributes, and makes sure any boilerplate needed for **all** Magento 2 pages is ready for us to use.

With this skeleton created, our next trick will be pulling in the default Magento page with a store's layout, branding, and navigation already created. We'll do this with something called layout handles.

Layout Handles

This is where things get a little weird. We're going to

1. Add a "handle" to our page object
2. Setup a "Layout Handle XML file" for that handle
3. Use that "Layout Handle XML file" to specify a default layout

If you don't know what a *handle* or *Layout Update XML file* is, don't worry. These are things Magento has invented. We're going to walk through the process of creating each, which should help you understand what they are.

First, we'll add a handle (`our_custom_handle`) to our page layout object.

```
1  #File: app/code/Pulsestorm/Nofrillslayout/Controller/
       Chapter3/Index.php
2  public function execute()
3  {
4      $pageObject = $this->resultPageFactory->create();
5      $pageObject->addHandle('our_custom_handle');
6      return $pageObject;
7  }
```

Next, we'll create the Layout Handle XML file

```
1  #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
       layout/our_custom_handle.xml
2  <page xsi:noNamespaceSchemaLocation="urn:magento:framework:
       View/Layout/etc/page_configuration.xsd" layout="1column
       ">
3  </page>
```

Finally, we'll clear our cache and reload the page. We should see a fully rendered one column Magento page with no content.

Congratulations – you just created your first Magento layout handle.

What are Handles

The best way to think about handles is as a layout specific event system. When it comes time for Magento to render a results page object, Magento will look at all the configured handles, and then ask the system

> Hey, here are my handles – does anyone have XML files for me?

Then, as module developers, we can create an XML file with our handle name (the `our_custom_handle.xml` file above). This is module developers responding with

> Hello layout system! Thank you for the handle list. Yes! We have a `our_custom_handle.xml` file with some layout updates inside.

The XML file itself contains further instructions. Our file was relatively simple.

```
1 #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
      layout/our_custom_handle.xml
2
3 <?xml version="1.0"?>
4 <page xsi:noNamespaceSchemaLocation="urn:magento:framework:
      View/Layout/etc/page_configuration.xsd" layout="1column
      ">
5 </page>
```

The root node of all layout handle XML files is `<page/>`, and the `xsi:noNamespaceSchemaLocation` attribute points to the Magento provided XSD schema validation file. The one piece of non-boilerplate code in our layout handle XML file is the following

```
1 layout="1column"
```

This tells Magento we want to use the "1column" default layout. We'll get to what using a different *page layout* means in the next chapter. Before we can talk about page layouts though, we need to talk a bit about default handles.

Default Handles

During normal run-of-the-mill development, you'll rarely add a custom handle like we did.

```
1 $pageObject->addHandle('our_custom_handle');
```

Like a lot of what we do in this book, this was a pedagogical shortcut to help you understand the system. You *might* include custom handles if your features are complex enough to warrant them, but by and large the default Magento handles should be enough to get you by.

What are the default Magento handles? Unfortunately – there's no simple code snippet you can use to view the default handles in Magento 2. Debugging tools like Commerce Bug

http://store.pulsestorm.net/products/commerce-bug-3)

will display the specific handles for any page in Magento.

That said, there are always two handles you can rely on being present.

The first is the `default` handle. This handle is fired on **every** page load. You can use `default` to add layout instructions you want to run everywhere in Magento. Magento uses the default node to build up much of a page's standard HTML structure and needed javascript/css code. There are plenty of examples where Magento's core modules use the `default` handle.

```
1  $ find vendor/magento -wholename '*/module-*/default.xml'
2  vendor/magento/module-admin-notification/view/adminhtml/
       layout/default.xml
3  vendor/magento/module-backend/view/adminhtml/layout/default
       .xml
4  vendor/magento/module-bundle/view/frontend/layout/default.
       xml
5  vendor/magento/module-captcha/view/frontend/layout/default.
       xml
6  vendor/magento/module-catalog/view/base/layout/default.xml
7  vendor/magento/module-catalog/view/frontend/layout/default.
       xml
8  vendor/magento/module-catalog-search/view/frontend/layout/
       default.xml
9  vendor/magento/module-checkout/view/frontend/layout/default
       .xml
10 vendor/magento/module-cms/view/frontend/layout/default.xml
11 vendor/magento/module-contact/view/frontend/layout/default.
       xml
12 vendor/magento/module-cookie/view/frontend/layout/default.
       xml
13 vendor/magento/module-customer/view/frontend/layout/default
       .xml
14 vendor/magento/module-directory/view/frontend/layout/
       default.xml
15 vendor/magento/module-google-analytics/view/frontend/layout
       /default.xml
```

```
16 vendor/magento/module-newsletter/view/frontend/layout/
       default.xml
17 vendor/magento/module-page-cache/view/frontend/layout/
       default.xml
18 vendor/magento/module-paypal/view/frontend/layout/default.
       xml
19 vendor/magento/module-reports/view/frontend/layout/default.
       xml
20 vendor/magento/module-rss/view/frontend/layout/default.xml
21 vendor/magento/module-sales/view/frontend/layout/default.
       xml
22 vendor/magento/module-search/view/frontend/layout/default.
       xml
23 vendor/magento/module-security/view/adminhtml/layout/
       default.xml
24 vendor/magento/module-theme/view/frontend/layout/default.
       xml
25 vendor/magento/module-ui/view/base/layout/default.xml
26 vendor/magento/module-weee/view/frontend/layout/default.xml
27 vendor/magento/module-widget/view/frontend/layout/default.
       xml
28 vendor/magento/module-wishlist/view/frontend/layout/default
       .xml
```

The second handle is also issued on every page – but has a *different name* on every page. This handle is known as the "full action name" handle. Every request made to Magento is identifiable by a full action name – this name combines the three portions of a standard Magento URL

```
1 /pulsestorm_nofrillslayout/chapter3/index
```

into an underscore separated string.

```
1 pulsestorm_nofrillslayout_chapter3_index
```

Magento will always add this underscore separated string as a layout handle name. If you're unsure of a page's full action name, you can peek at it with the following code in your controller's `execute` method.

```
1 #File: app/code/Pulsestorm/Nofrillslayout/Controller/
      Chapter3/Index.php
2 public function execute()
3 {
4     var_dump($this->getRequest()->getFullActionName());
5 }
```

Another example should clarify things further. First, lets remove the custom layout handle from our controller

```
#File: app/code/Pulsestorm/Nofrillslayout/Controller/
    Chapter3/Index.php
public function execute()
{
    $pageObject = $this->resultPageFactory->create();
    //$pageObject->addHandle('our_custom_handle');
    return $pageObject;
}
```

If we reload our page with the above in place, we'll be back to a blank browser window. Without our custom handle, Magento doesn't know which page layout it should load.

However, we now know Magento will always issue a `pulsestorm_nofrillslayout_chapter3_index` handle on this page. So, if we rename our layout handle XML file from `our_custom_handle.xml` to `pulsestorm_nofrillslayout_chapter3_index.xml`, clear our cache, and refresh the page we'll see our layout restored.

Adding Content via Layout Update XML Files

Layout handles are for more than just creating a default layout. They can also contain any bit of layout update XML – the sort of things we were doing manually in Chapter 2. Give the following a try in our new `pulsestorm_nofrillslayout_chapter3_index.xml` file.

```
<!-- File: app/code/Pulsestorm/Nofrillslayout/view/frontend
    /layout//pulsestorm_nofrillslayout_chapter3_index.xml
    -->
<page xsi:noNamespaceSchemaLocation="urn:magento:framework:
    View/Layout/etc/page_configuration.xsd" layout="1column
    ">
    <body>
        <referenceContainer name="content">
            <block  name="pulsestorm_nofrills_chapter3_text
                "
                    class="Magento\Framework\View\Element\
                        Text">
                <arguments>
                    <argument name="text" xsi:type="string"
                        >This is a test.</argument>
                </arguments>
            </block>
        </referenceContainer>
    </body>
</page>
```

Clear your cache, reload the page, and you should see the phrase

> This is a test

appended to the `content` container.

The layout update XML should be familiar to you

```
<!-- File: app/code/Pulsestorm/Nofrillslayout/view/frontend
    /layout//pulsestorm_nofrillslayout_chapter3_index.xml
    -->

<referenceContainer name="content">
    <block  name="pulsestorm_nofrills_chapter3_text"
            class="Magento\Framework\View\Element\Text">
        <arguments>
            <argument name="text" xsi:type="string">This is
                a test.</argument>
        </arguments>
    </block>
</referenceContainer>
```

This code gets a reference to a preexisting container (added by Magento) named `content`. Then, it instantiates a text block object, using an `<argument>` node to set the block's text property. i.e. This code tells Magento to instantiate a text block and add that block to the content container.

More important for us though is *where* this block sits

```
<!-- File: app/code/Pulsestorm/Nofrillslayout/view/frontend
    /layout//pulsestorm_nofrillslayout_chapter3_index.xml
    -->
<page xsi:noNamespaceSchemaLocation="urn:magento:framework:
    View/Layout/etc/page_configuration.xsd" layout="1column
    ">
    <body>
        <!-- ... -->
    </body>
</page>
```

In a page layout handle XML file, any layout update XML should go in the `<body/>` section of the document. This is where Magento will look for layout update blocks to apply. In addition to being behaviorally enforced (i.e. put those blocks somewhere else and Magento ignores them), the following XML schema helps enforce these rules

```
urn:magento:framework:View/Layout/etc/page_configuration.
    xsd
```

You can find this schema in the following file

```
1 vendor/magento/framework/View/Layout/etc/page_configuration
      .xsd
```

While beyond the scope of this particular book, it's worth spending some time looking at these XSD files. In addition to helping developers keep their XML files correct, they also serve as the de-facto documentation on how a certain type of XML file is meant to be structured.

Although a simple example, that's layout handles in a nutshell. You may be wondering where the `content` block we referenced came from – in our next chapter we'll take a look at what's averrable in a default page layout for Magento developers, as well as how Magento renders the overall page skeleton.

Chapter 5

Page Layouts in Magento 2

When it comes to the HTML page layout for a page, so far we've spent a lot of time talking about

1. Individual components of Magento's layout system and using those components to build up simple examples "from scratch"
2. Starting with a fully rendered Magento page and slightly altering it.

In this chapter, we're going to dive a bit deeper and take a look at how **Magento** creates its fully designed layouts from scratch. We'll do this by stripping a Magento MVC endpoint down to a naked, blank, HTML page.

Getting to a Blank Page

To start with, load the following URL in your browser

```
1 http://magento.example.com/pulsestorm_nofrillslayout/blank/
      index
```

You should see a one column Magento page with the message *Hello Blank Page*. If you're curious, the page's controller file looks like this

```
1 #File: app/code/Pulsestorm/Nofrillslayout/Controller/Blank/
      Index.php
2 <?php
3 namespace Pulsestorm\Nofrillslayout\Controller\Blank;
4 use Pulsestorm\Nofrillslayout\Controller\BaseController;
5 class Index extends \Magento\Framework\App\Action\Action
6 {
7     protected $resultPageFactory;
8     public function __construct(
```

```
9           \Magento\Framework\App\Action\Context $context,
10          \Magento\Framework\View\Result\PageFactory
                $resultPageFactory
11      )
12      {
13          $this->resultPageFactory = $resultPageFactory;
14          return parent::__construct($context);
15      }
16      public function execute()
17      {
18          $pageObject = $this->resultPageFactory->create();
19          return $pageObject;
20      }
21  }
```

and its layout handle XML file looks like this

```
1   #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
        layout/pulsestorm_nofrillslayout_blank_index.xml
2   <?xml version="1.0"?>
3   <page   xsi:noNamespaceSchemaLocation="urn:magento:
        framework:View/Layout/etc/page_configuration.xsd"
4           layout="1column">
5       <body>
6           <referenceContainer name="content">
7               <block
8                   template="Pulsestorm_Nofrillslayout::blank/
                        hello.phtml"
9                   class="Magento\Framework\View\Element\
                        Template"
10                  name="pulsestorm_nofrillslayout_blank_hello
                        "/>
11          </referenceContainer>
12      </body>
13  </page>
```

Our goal is to strip away Magento's outer shell and render as simple an HTML page as possible. Magento's layout system actually has a feature for rendering a empty page. If we change the `layout="1column"` to 'layout=“empty”

```
1   #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
        layout/pulsestorm_nofrillslayout_blank_index.xml
2   <?xml version="1.0"?>
3   <page   xsi:noNamespaceSchemaLocation="urn:magento:
        framework:View/Layout/etc/page_configuration.xsd"
4           layout="empty">
5           <!-- ... -->
6   </page>
```

clear our cache, and then reload the page, we should see an "empty" page with our content.

Observant viewers may know why we put "empty" in skeptical quotes. While the Magento page branding and navigation is gone, our page itself is still rendered with a non-default type-face, and some non-default indenting. If we view the source of the document returned from the server, we'll see something that looks similar to the following

```
<!doctype html>
<html lang="en-US">
    <head>
        <script>
            var require = {
                "baseUrl": "http://magento.example.com/
                    static/frontend/Magento/luma/en_US"
            };
        </script>
        <meta charset="utf-8" />
        <meta name="description" content="Default
            Description" />
        <meta name="keywords" content="Magento, Varien, E-
            commerce" />
        <meta name="robots" content="INDEX,FOLLOW" />
        <meta name="viewport" content="width=device-width,
            initial-scale=1, maximum-scale=1.0, user-
            scalable=no" />
        <title></title>
        <link rel="stylesheet" type="text/css" media="all"
            href="http://magento.example.com/static/
            frontend/Magento/luma/en_US/mage/calendar.css"
            />
        <link rel="stylesheet" type="text/css" media="all"
            href="http://magento.example.com/static/
            frontend/Magento/luma/en_US/css/styles-m.css"
            />
        <link rel="stylesheet" type="text/css" media="
            screen and (min-width: 768px)" href="http://
            magento.example.com/static/frontend/Magento/
            luma/en_US/css/styles-l.css" />
        <link rel="stylesheet" type="text/css" media="print
            " href="http://magento.example.com/static/
            frontend/Magento/luma/en_US/css/print.css" />
        <link rel="icon" type="image/x-icon" href="http://
            magento.example.com/static/frontend/Magento/
            luma/en_US/Magento_Theme/favicon.ico" />
        <link rel="shortcut icon" type="image/x-icon" href=
            "http://magento.example.com/static/frontend/
            Magento/luma/en_US/Magento_Theme/favicon.ico"
            />
```

```
21          <script type="text/javascript" src="http://magento.
                example.com/static/frontend/Magento/luma/en_US/
                requirejs/require.js">
22          </script>
23          <script type="text/javascript" src="http://magento.
                example.com/static/frontend/Magento/luma/en_US/
                mage/requirejs/mixins.js">
24          </script>
25          <script type="text/javascript" src="http://magento.
                example.com/static/_requirejs/frontend/Magento/
                luma/en_US/requirejs-config.js">
26          </script>
27          <script type="text/javascript" src="http://magento.
                example.com/static/frontend/Magento/luma/en_US/
                Pulsestorm_Nofrillslayout/user/example-script.
                js">
28          </script>
29          <link rel="stylesheet" type="text/css" media="all"
                href="http://magento.example.com/media/styles.
                css" />
30      </head>
31      <body data-container="body" data-mage-init='{"
            loaderAjax": {}, "loader": { "icon": "http://
            magento.example.com/static/frontend/Magento/luma/
            en_US/images/loader-2.gif"}}' class="pulsestorm-
            nofrillslayout-blank-index page-layout-empty">
32          <script>
33              require.config({
34                  deps: [
35                      'jquery',
36                      'mage/translate',
37                      'jquery/jquery-storageapi'
38                  ],
39                  callback: function ($) {
40                      'use strict';
41
42                      var dependencies = [],
43                          versionObj;
44
45                      $.initNamespaceStorage('mage-
                            translation-storage');
46                      $.initNamespaceStorage('mage-
                            translation-file-version');
47                      versionObj = $.localStorage.get('mage-
                            translation-file-version');
48
49                      if (versionObj.version !== '91
                            c3baf5050b0c1cb0285ace5c5cb45e3f973149
                            ') {
50                          dependencies.push(
```

```
51                              'text!js-translation.json'
52                          );
53
54                      }
55
56                      require.config({
57                          deps: dependencies,
58                          callback: function (string) {
59                              if (typeof string === 'string')
                                    {
60                                  $.mage.translate.add(JSON.
                                        parse(string));
61                                  $.localStorage.set('mage-
                                        translation-storage',
                                        string);
62                                  $.localStorage.set(
63                                      'mage-translation-file-
                                            version',
64                                      {
65                                          version: '91
                                                c3baf5050b0c1cb0285ace5c5cb45e3f973149
                                                '
66                                      }
67                                  );
68                              } else {
69                                  $.mage.translate.add($.
                                        localStorage.get('mage-
                                        translation-storage'));
70                              }
71                          }
72                      });
73                  }
74              });
75          </script>
76          <script type="text/x-magento-init">
77              {
78                  "*": {
79                      "mage/cookies": {
80                          "expires": null,
81                          "path": "/",
82                          "domain": ".magento.example.com",
83                          "secure": false,
84                          "lifetime": "3600"
85                      }
86                  }
87              }
88          </script>
89          <noscript>
90              <div class="message global noscript">
91                  <div class="content">
```

```
92                      <p>
93                          <strong>
94                              JavaScript seems to be disabled
                                    in your browser.
95                          </strong>
96                          <span>
97                              For the best experience on our
                                    site, be sure to turn on
                                    Javascript in your browser.
98                          </span>
99                      </p>
100                 </div>
101             </div>
102         </noscript>
103         <div class="page-wrapper">
104             <main id="maincontent" class="page-main">
105                 <a id="contentarea" tabindex="-1">
106                 </a>
107                 <div class="page messages">
108                     <div data-placeholder="messages">
109                     </div>
110                     <div data-bind="scope: 'messages'">
111                         <div data-bind="foreach: { data:
                                cookieMessages, as: 'message' }
                                " class="messages">
112                             <div data-bind="attr: {
113             class: 'message-' + message.type + ' ' +
                    message.type + ' message',
114             'data-ui-id': 'message-' + message.type
115         }">
116                                 <div data-bind="html:
                                        message.text">
117                                 </div>
118                             </div>
119                         </div>
120                         <div data-bind="foreach: { data:
                                messages().messages, as: '
                                message' }" class="messages">
121                             <div data-bind="attr: {
122             class: 'message-' + message.type + ' ' +
                    message.type + ' message',
123             'data-ui-id': 'message-' + message.type
124         }">
125                                 <div data-bind="html:
                                        message.text">
126                                 </div>
127                             </div>
128                         </div>
129                     </div>
130                     <script type="text/x-magento-init">
```

```
131                             {
132                                 "*": {
133                                     "Magento_Ui/js/core/app": {
134                                         "components": {
135                                                 "messages": {
136                                                     "component"
                                                        : "
                                                        Magento_Theme
                                                        /js/
                                                        view/
                                                        messages
                                                        "
137                                                 }
138                                             }
139                                         }
140                                     }
141                             }
142                         </script>
143                 </div>
144                 <div class="columns">
145                     <div class="column main">
146                         <input name="form_key" type="hidden
                            " value="QK7qYF0ueevnC66R" />
147                         <div id="authenticationPopup" data-
                            bind="scope:'
                            authenticationPopup'" style="
                            display: none;">
148                             <script>
149                                 window.authenticationPopup
                                    = {"customerRegisterUrl
                                    ":"http:\/\/magento.
                                    example.com\/customer\/
                                    account\/create\/","
                                    customerForgotPasswordUrl
                                    ":"http:\/\/magento.
                                    example.com\/customer\/
                                    account\/forgotpassword
                                    \/","baseUrl":"http
                                    :\/\/magento.example.
                                    com\/"};
150                             </script>
151 <!-- ko template: getTemplate() -->
152 <!-- /ko -->
153                             <script type="text/x-magento-
                                init">
154                                 {
155                                     "#authenticationPopup":
                                        {
156                                         "Magento_Ui/js/core
                                            /app": {"
```

```
                                        components":{"
                                        authenticationPopup
                                        ":{"component":
                                        "
                                        Magento_Customer
                                        \/js\/view\/
                                        authentication-
                                        popup","
                                        children":{"
                                        messages":{"
                                        component":"
                                        Magento_Ui\/js
                                        \/view\/
                                        messages","
                                        displayArea":"
                                        messages"},"
                                        captcha":{"
                                        component":"
                                        Magento_Captcha
                                        \/js\/view\/
                                        checkout\/
                                        loginCaptcha","
                                        displayArea":"
                                        additional-
                                        login-form-
                                        fields","formId
                                        ":"user_login",
                                        "configSource":
                                        "checkout"}}}}}
                                                    },
157                                 "*": {
158                                     "Magento_Ui/js/
                                            block-loader":
                                            "http://magento
                                            .example.com/
                                            static/frontend
                                            /Magento/luma/
                                            en_US/images/
                                            loader-1.gif"
159                                 }
160                             }
161                         </script>
162                     </div>
163                     <script type="text/x-magento-init">
164                         {"*":{"Magento_Customer\/js\/
                            section-config":{"sections"
                            :{"stores\/store\/switch":"
                            *","directory\/currency\/
                            switch":"*","*":["messages"
                            ],"customer\/account\/
```

```
logout":"*","customer\/
account\/loginpost":"*","
customer\/account\/
createpost":"*","customer\/
ajax\/login":["checkout-
data","cart"],"catalog\/
product_compare\/add":["
compare-products"],"catalog
\/product_compare\/remove"
:["compare-products"],"
catalog\/product_compare\/
clear":["compare-products"
],"sales\/guest\/reorder":[
"cart"],"sales\/order\/
reorder":["cart"],"checkout
\/cart\/add":["cart"],"
checkout\/cart\/delete":["
cart"],"checkout\/cart\/
updatepost":["cart"],"
checkout\/cart\/
updateitemoptions":["cart"
],"checkout\/cart\/
couponpost":["cart"],"
checkout\/cart\/
estimatepost":["cart"],"
checkout\/cart\/
estimateupdatepost":["cart"
],"checkout\/onepage\/
saveorder":["cart","
checkout-data","last-
ordered-items"],"checkout\/
sidebar\/removeitem":["cart
"],"checkout\/sidebar\/
updateitemqty":["cart"],"
rest\/*\/v1\/carts\/*\/
payment-information":["cart
","checkout-data","last-
ordered-items"],"rest\/*\/
v1\/guest-carts\/*\/payment
-information":["cart","
checkout-data"],"rest\/*\/
v1\/guest-carts\/*\/
selected-payment-method":["
cart","checkout-data"],"
rest\/*\/v1\/carts\/*\/
selected-payment-method":["
cart","checkout-data"],"
multishipping\/checkout\/
overviewpost":["cart"],"
paypal\/express\/placeorder
```

```
                                    ":["cart","checkout-data"],
                                    "paypal\/payflowexpress\/
                                    placeorder":["cart","
                                    checkout-data"],"review\/
                                    product\/post":["review"],"
                                    authorizenet\/
                                    directpost_payment\/place"
                                    :["cart","checkout-data"],"
                                    braintree\/paypal\/
                                    placeorder":["cart","
                                    checkout-data"],"wishlist\/
                                    index\/add":["wishlist"],"
                                    wishlist\/index\/remove":["
                                    wishlist"],"wishlist\/index
                                    \/updateitemoptions":["
                                    wishlist"],"wishlist\/index
                                    \/update":["wishlist"],"
                                    wishlist\/index\/cart":["
                                    wishlist","cart"],"wishlist
                                    \/index\/fromcart":["
                                    wishlist","cart"],"wishlist
                                    \/index\/allcart":["
                                    wishlist","cart"],"wishlist
                                    \/shared\/allcart":["
                                    wishlist","cart"],"wishlist
                                    \/shared\/cart":["cart"]},"
                                    clientSideSections":["
                                    checkout-data"],"baseUrls"
                                    :["http:\/\/magento.example
                                    .com\/"]}}}
165                             </script>
166                             <script type="text/x-magento-init">
167                                 {"*":{"Magento_Customer\/js\/
                                    customer-data":{"
                                    sectionLoadUrl":"http:\/\/
                                    magento.example.com\/
                                    customer\/section\/load\/",
                                    "cookieLifeTime":"3600","
                                    updateSessionUrl":"http
                                    :\/\/magento.example.com\/
                                    customer\/account\/
                                    updateSession\/"}}}
168                             </script>
169                             <script type="text/x-magento-init">
170                                 {
171                                     "body": {
172                                         "pageCache": {"url":"
                                            http:\/\/magento.
                                            example.com\/
                                            page_cache\/block\/
```

```
                                                render\/","handles"
                                                :["default","
                                                pulsestorm_nofrillslayout_blank_index
                                                "],"originalRequest
                                                ":{"route":"
                                                pulsestorm_nofrillslayout
                                                ","controller":"
                                                blank","action":"
                                                index","uri":"\/
                                                pulsestorm_nofrillslayout
                                                \/blank\/index"},"
                                                versionCookieName":
                                                "
                                                private_content_version
                                                "}        }
173                                 }
174                             </script>
175                             <p>
176                                 Hello Blank Page.
177                             </p>
178                         </div>
179                     </div>
180                 </main>
181                 <small class="copyright">
182                     <span>
183                         Copyright © 2016 Magento. All rights
                                reserved.
184                     </span>
185                 </small>
186             </div>
187         </body>
188     </html>
```

In other words, we'll see a document that's **far** from *empty*. The copyright notice, the default styles and javascript, the many javascript blocks Magento relies on for base functionality, the `page-wrapper` `<div/>`, the `<main/>` section, etc. While an end user may look at this page and consider it empty, a developer knows better.

More Empty

There's a way we can get a page that's "more" empty. If we remove the `layout="empty"` tag, or set it to a value that Magento doesn't recognize, we'll get something that's even **more** empty. We're going to opt for setting it to `layout="more-empty"`, as some versions of Magento 2 set a default value for this attribute in other layout handles.

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    layout/pulsestorm_nofrillslayout_blank_index.xml
<?xml version="1.0"?>
<page   xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
        xsi:noNamespaceSchemaLocation="
            ../../../../../../../lib/internal/Magento/
            Framework/View/Layout/etc/page_configuration.
            xsd"
        layout="more-empty">
        <!-- ... -->
</page>
```

If you clear your cache and reload the browser page with the above in place, we'll see a page that's much more blank.

In fact, this page is **so** blank, that even our content is gone! We'll explain that side effect in a bit, but let's take a look at the HTML source of the page

```
<!doctype html>
<html lang="en-US">
    <head>
        <script>
            var require = {
                "baseUrl": "http://magento.example.com/
                    static/frontend/Magento/luma/en_US"
            };
        </script>
        <meta charset="utf-8" />
        <meta name="description" content="Default
            Description" />
        <meta name="keywords" content="Magento, Varien, E-
            commerce" />
        <meta name="robots" content="INDEX,FOLLOW" />
        <meta name="viewport" content="width=device-width,
            initial-scale=1, maximum-scale=1.0, user-
            scalable=no" />
        <title>
        </title>
        <link rel="stylesheet" type="text/css" media="all"
            href="http://magento.example.com/static/
            frontend/Magento/luma/en_US/mage/calendar.css"
            />
        <link rel="stylesheet" type="text/css" media="all"
            href="http://magento.example.com/static/
            frontend/Magento/luma/en_US/css/styles-m.css"
            />
        <link rel="stylesheet" type="text/css" media="
            screen and (min-width: 768px)" href="http://
```

```
            magento.example.com/static/frontend/Magento/
            luma/en_US/css/styles-l.css" />
19          <link rel="stylesheet" type="text/css" media="print
            " href="http://magento.example.com/static/
            frontend/Magento/luma/en_US/css/print.css" />
20          <link rel="icon" type="image/x-icon" href="http://
            magento.example.com/static/frontend/Magento/
            luma/en_US/Magento_Theme/favicon.ico" />
21          <link rel="shortcut icon" type="image/x-icon" href=
            "http://magento.example.com/static/frontend/
            Magento/luma/en_US/Magento_Theme/favicon.ico"
            />
22          <script type="text/javascript" src="http://magento.
            example.com/static/frontend/Magento/luma/en_US/
            requirejs/require.js">
23          </script>
24          <script type="text/javascript" src="http://magento.
            example.com/static/frontend/Magento/luma/en_US/
            Pulsestorm_Nofrillslayout/user/example-script.
            js">
25          </script>
26          <link rel="stylesheet" type="text/css" media="all"
            href="http://magento.example.com/media/styles.
            css" />
27      </head>
28      <body data-container="body" data-mage-init='{"
            loaderAjax": {}, "loader": { "icon": "http://
            magento.example.com/static/frontend/Magento/luma/
            en_US/images/loader-2.gif"}}' class="pulsestorm-
            nofrillslayout-blank-index page-layout-more-empty">
29      </body>
30  </html>
```

While there's less here, it's still hard to call this page empty. The `baseUrl` configuration for RequireJS, the `<meta/>` tags, the default styles, the `<script/>` tag pulling in RequireJS, and the attributes of the `<body/>` tag all point to more code archaeology work to be done at deeper levels of the Magento system.

Magento 2's root.phtml Template

If you're familiar with Magento 1, you know Magento had a number of default "root" templates for a store.

```
1  $ ls -1 /path/to/magento-1/app/design/frontend/base/default
       /template/page/
2  1column.phtml
3  2columns-left.phtml
```

```
2columns-right.phtml
3columns.phtml
empty.phtml
//...
```

Each of these did pretty much what the file name said. A one column page, a two column page with a left column, a two column page with a right column, etc. Each of these templates set the base skeleton for a Magento page, and began rendering child blocks for each section and column of the page.

While Magento 2's layout system is very similar to Magento 1's, this is one area where things have changed substantially. Magento 2 has **one** `phtml` root template file, and this file **cannot** be changed by the layout system.

This file is located at the following path, and contains the following content.

```
#File: vendor/magento/module-theme/view/base/templates/root.phtml
<!doctype html>
<html <?php /* @escapeNotVerified */ echo $htmlAttributes ?>>
    <head <?php /* @escapeNotVerified */ echo $headAttributes ?>>
        <?php /* @escapeNotVerified */ echo $requireJs ?>
        <?php /* @escapeNotVerified */ echo $headContent ?>
        <?php /* @escapeNotVerified */ echo $headAdditional ?>
    </head>
    <body data-container="body" data-mage-init='{"loaderAjax": {}, "loader": { "icon": "<?php /* @escapeNotVerified */ echo $loaderIcon; ?>"}}' <?php /* @escapeNotVerified */ echo $bodyAttributes ?>>
        <?php /* @escapeNotVerified */ echo $layoutContent ?>
    </body>
</html>
```

This `phtml` template is responsible for rendering every Magento MVC page. The `/* @escapeNotVerified /` contents are a flag for Magento's test suite that say "we know these variables aren't escaped, they're probably OK as is, but we haven't verified that". Let's remove those comments so the file's a little easier to read, as well as jigger the formatting a bit.

```
#File: vendor/magento/module-theme/view/base/templates/root.phtml
<!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>
```

The first big change for a Magento 1 developer is – there's no calls to `getChildHtml`. Instead, Magento `echo`s out eight different variables

```
$htmlAttributes
$headAttributes
$requireJs
$headContent
$headAdditional
$loaderIcon
$bodyAttributes
$layoutContent
```

Each of these variables contains default page elements, and explains why our *more-empty* layout was still not really empty. Our next logical question is: Where to these variables come from?

The View Results Page Class

In Magento 2, the `Magento\Framework\View\Result\Page` class is ultimately responsible for rendering an MVC response. The reasons for this are, unfortunately, too complicated to go into right now. The more adventurous among you will need to take that journey on your own.

For now, we're interested in the `renderPage` method of `Magento\Framework\View\Result\Page`.

```
#File: vendor/magento/framework/View/Result/Page.php
protected function renderPage()
{
    $fileName = $this->viewFileSystem->getTemplateFileName(
        $this->template);
    if (!$fileName) {
```

```
6           throw new \InvalidArgumentException('Template "' .
                $this->template . '" is not found');
7       }
8
9       ob_start();
10      try {
11          extract($this->viewVars, EXTR_SKIP);
12          include $fileName;
13      } catch (\Exception $exception) {
14          ob_end_clean();
15          throw $exception;
16      }
17      $output = ob_get_clean();
18      return $output;
19  }
```

This method creates a file path from the `$this->template` variable.

```
1  #File: vendor/magento/framework/View/Result/Page.php
2
3  $fileName = $this->viewFileSystem->getTemplateFileName(
       $this->template);
```

Then, it uses PHP's `extract` function to pull all the `$this->viewVars` array keys into the local scope as variables.

```
1  #File: vendor/magento/framework/View/Result/Page.php
2
3  extract($this->viewVars, EXTR_SKIP);
```

This means they'll also be available in `$fileName` when Magento uses PHP's `include` function to load `$fileName`.

```
1  #File: vendor/magento/framework/View/Result/Page.php
2
3  include $fileName;
```

This code is also wrapped in output buffering functions and a try/catch block. If you've ever wondered why Magento 2 exceptions *only* print an error and not any content – this is why.

The `$fileName` variable contains the full path to our `root.phtml` template. Magento populates the `$this->template` variable via automatic constructor dependency injection

```
1  #File: vendor/magento/framework/View/Result/Page.php
2  public function __construct(
3      /* ... */
4      $template,
```

```
    /* ... */
) {
    /* ... */
    $this->template = $template;
    /* ... */
}
```

The value of this variable is `Magento_Theme::root.phtml`, set in the following core `di.xml` file.

```
#File: vendor/magento/magento2-base/app/etc/di.xml
<type name="Magento\Framework\View\Result\Page">
    <arguments>
        <!-- ... -->
        <argument name="template" xsi:type="string">
            Magento_Theme::root.phtml
        </argument>
    </arguments>
</type>
```

Don't worry if you didn't follow all of that. The key take away is Magento renders `root.phtml` via a plain old `include`, and the variables `echo`ed inside that template come from the `viewVars` array.

Our next question? What populates `viewVars`?

View Variables for the View Results Page Class

The `Magento\Framework\View\Result\Page` class has an `assign` method.

```
#File: vendor/magento/framework/View/Result/Page.php
protected function assign($key, $value = null)
{
    if (is_array($key)) {
        foreach ($key as $subKey => $subValue) {
            $this->assign($subKey, $subValue);
        }
    } else {
        $this->viewVars[$key] = $value;
    }
    return $this;
}
```

This method is pretty simple: Pass it a key and a value, and the method sets a value on the `viewVars` array object property. We mention `assign`, because if we look at the `render` method (the method that itself calls `renderPage` from earlier in the chapter)

```
#File: vendor/magento/framework/View/Result/Page.php
protected function render(ResponseInterface $response)
{
    /* ... */
    $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);
    /* ... */
    return $this;
}
```

We see **this** is where, (via `assign`), that Magento sets the variables rendered in `root.phtml`. In other words `$layoutContent` here

```
#File: vendor/magento/module-theme/view/base/templates/root
    .phtml
<?php /* @escapeNotVerified */ echo $layoutContent ?>
```

comes from this call here

```
#File: vendor/magento/framework/View/Result/Page.php

$output = $this->getLayout()->getOutput();
$this->assign('layoutContent', $output);
```

Each of these view variables is set in a different way. The `$output = $this->getLayout()->getOutput();` looks for output generated after Magento has processed all the layout handle XML files in the system for a particular request.

The `$loaderIcon` variable, on the other hand, is a simple rendering of a URL using the `getViewFileUrl` method.

```
#File: vendor/magento/framework/View/Result/Page.php

'loaderIcon' => $this->getViewFileUrl('images/loader-2.gif'),
```

Magento populates the `$headAdditional` and `$requireJs` variables by rendering a **specific** block from the layout

```
#File: vendor/magento/framework/View/Result/Page.php

$addBlock = $this->getLayout()->getBlock('head.additional'); // todo
$requireJs = $this->getLayout()->getBlock('require.js');
//...
'requireJs' => $requireJs ? $requireJs->toHtml() : null,
'headAdditional' => $addBlock ? $addBlock->toHtml() : null,
```

But values for `$headContent`, `$htmlAttributes`, `$headAttributes`, and `$bodyAttributes` come from calls to the `pageConfigRenderer` property

```
        'headContent' => $this->pageConfigRenderer->
            renderHeadContent(),
        'htmlAttributes' => $this->pageConfigRenderer->
            renderElementAttributes($config::
            ELEMENT_TYPE_HTML),
        'headAttributes' => $this->pageConfigRenderer->
            renderElementAttributes($config::
            ELEMENT_TYPE_HEAD),
        'bodyAttributes' => $this->pageConfigRenderer->
            renderElementAttributes($config::
            ELEMENT_TYPE_BODY),
```

The `pageConfigRenderer` property is a `Magento\Framework\View\Page\Config\Renderer` object, set somewhat indirectly in the constructor by a factory that's injected via automatic constructor dependency injection.

We're **not** going to track down the code behind the setting of each of these variables. That is, again, a job for a more intrepid developer. We brought you down this deep into Magento's internals to

1. Show you where the root templates comes from
2. Make a point about "second system" syndrome.

While there's no doubt that the individual developer who made these changes to Magento's layout system did so with the best of intentions, we've ended up with a new system that has only **increased** the complexity of the layout system.

While we *could* keep going in our efforts to produce a true "blank" Magento 2 page, those efforts would only lead to additional instabilities. We *could* remove the `head.additional` and `requirejs` blocks from the layout, and figure out how to manipulate the `Magento\Framework\View\Page\Config\Renderer` object so it produces empty attributes. Also, if you're familiar with Magento's automatic constructor dependency injection system we could even **replace** the `root.phtml` template file with one of our own choosing.

However, doing so would likely **break** Magento's default themes in fundamental ways. Removing the RequireJS `<script/>` tag would break most of Magento's javascript. Removing default classes would break not only the styling, but javascript code that uses those classes in jQuery selectors, etc.

Much like the homeowner who turns off everything electric in their house and puzzles over the meter that's still recording usage, the above HTML skeleton is as close as most Magento developers will be able to get to a blank page **without** reimplementing all the functionality of a base Magento store.

That leaves us with one last questions to answer: What the heck happened to our *Hello Blank Page* content?

Page Layouts

In our layout handle XML file, we have the following code

```
1  #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
       layout/pulsestorm_nofrillslayout_blank_index.xml
2
3  <referenceContainer name="content">
4      <block
5          template="Pulsestorm_Nofrillslayout::blank/hello.
               phtml"
6          class="Magento\Framework\View\Element\Template"
7          name="pulsestorm_nofrillslayout_blank_hello"/>
8  </referenceContainer>
```

This code added our `pulsestorm_nofrillslayout_blank_hello` block to the already created container named `content`. When we changed our layout to `layout ="more-empty"` we **lost** the container named content. To fix this, we'll need to make our "more-empty" page layout a **real** page layout.

Page layouts are configured via page layout XML files. A page layout configuration file looks similar to a layout handle XML file. Magento loads page layout

XML files using many of the same classes that load layout handle XML, but Magento loads **page layout** files **separately** from **layout** handle XML files.

A quick example should help set that in your mind. Add the following file to the `Pulsestorm_Nofrillslayout` module

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    page_layout/more-empty.xml
<layout xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
        xsi:noNamespaceSchemaLocation="urn:magento:
            framework:View/Layout/etc/page_layout.xsd">

    <container name="our_first_container"/>
</layout>
```

This is a page layout XML file. While it looks similar to a layout handle XML file – notice the root node is named `<layout/>`, and not `<page/>`. By putting this file in a `page_layout` folder, we've told Magento it's a page layout XML file, and not a layout handle XML file. By naming this file `more-empty`, we've told Magento that the rules in this file make up the page layout named *more-empty*.

Inside this file, we have a single container node

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    page_layout/more-empty.xml

<container name="our_first_container"/>
```

This means our layout has a single container named `our_first_container`. Let's update our layout handle XML file to insert our block into **this** container instead of the container named `content`. In other words, replace `<referenceContainer name="content">` with `<referenceContainer name="our_first_container">`.

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    layout/pulsestorm_nofrillslayout_blank_index.xml

<referenceContainer name="our_first_container">
    <!-- ... --->
</referenceContainer>
```

With the above in place, clear your cache and reload the page.

> Hello Blank Page

Eureka – our content is restored!

Default Layouts

These page layout XML files do **one** of the jobs that Magento 1's `1column.phtml`, `2columns-left.phtml`, `2columns-right.phtml`, `3columns.phtml`, `empty.phtml` files did. A page layout sets up the default container structure for a page. However, as previously mentioned, in Magento 2 there's a separate system for creating the skeleton HTML. This is a pattern you'll see repeated all over Magento 2 – systems that did multiple things in Magento 1 will be broken out and separated into two separate systems in Magento 2.

When we initially configured `more-empty` as a page layout (i.e. when we created the `more-empty.xml` file), we effectively removed **all** the containers from the page. This meant when our layout handle XML file tried to get a reference to the `content` container (`<referenceContainer name="content">`), there was no container named content, and nowhere to put the block. Even if you're in developer mode, the `<referenceContainer/>` and `<referenceBlock/>` methods will fail silently when they can't find a block or container.

Let's take a look at the default/stock page layout files that ship with Magento 2

```
$ find vendor/magento/ -wholename '*/page_layout/*'
module-checkout/view/frontend/page_layout/checkout.xml

module-theme/view/frontend/page_layout/1column.xml
module-theme/view/frontend/page_layout/2columns-left.xml
module-theme/view/frontend/page_layout/2columns-right.xml
module-theme/view/frontend/page_layout/3columns.xml
module-theme/view/base/page_layout/empty.xml

module-layered-navigation/view/frontend/page_layout/1column
    .xml
module-layered-navigation/view/frontend/page_layout/2
    columns-left.xml
module-layered-navigation/view/frontend/page_layout/2
    columns-right.xml
module-layered-navigation/view/frontend/page_layout/3
    columns.xml
module-layered-navigation/view/frontend/page_layout/empty.
    xml

module-security/view/adminhtml/page_layout/admin-popup.xml
module-theme/view/adminhtml/page_layout/admin-1column.xml
module-theme/view/adminhtml/page_layout/admin-2columns-left
    .xml
module-theme/view/adminhtml/page_layout/admin-empty.xml
module-theme/view/adminhtml/page_layout/admin-login.xml
```

Based on the above, we can see Magento 2 has the following page layouts available in the front end cart.

```
layout="checkout"
layout="1column"
layout="2columns-left"
layout="2columns-right"
layout="3columns"
layout="empty"
```

In the backend admin (the `adminhtml` area) we have the following page layouts available

```
layout="admin-1column"
layout="admin-2column"
layout="admin-empty"
layout="admin-login"
layout="admin-popup"
```

You may be wondering why the layered navigation module contains its own versions of these page layout files

```
vendor/magento/module-theme/view/frontend/page_layout/1
    column.xml
vendor/magento/module-layered-navigation/view/frontend/
    page_layout/1column.xml
```

Well – much like layout handle XML files, Magento will load page layout XML files from **all** available modules and combine them. This allows *any* module to create (as we did with `more-empty`) their own page layout, or **add to** (as the `module-layered-navigation` module does) an existing page layout.

Where's The Content Block?

You'll recall our earlier attempt to create an empty layout with the

```
layout="empty"
```

attribute. Let's take a look at the XML DSL code for the `empty` page layout.

```
#File: vendor/magento/module-theme/view/base/page_layout/
    empty.xml
<layout xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xsi:noNamespaceSchemaLocation="urn:magento:
    framework:View/Layout/etc/page_layout.xsd">
    <container name="root">
```

```
4           <container name="after.body.start" as="after.body.
                start" before="-" label="Page Top"/>
5           <container name="page.wrapper" as="page_wrapper"
                htmlTag="div" htmlClass="page-wrapper">
6               <container name="global.notices" as="
                    global_notices" before="-"/>
7               <container name="main.content" htmlTag="main"
                    htmlId="maincontent" htmlClass="page-main">
8                   <container name="columns.top" label="Before
                         Main Columns"/>
9                   <container name="columns" htmlTag="div"
                        htmlClass="columns">
10                      <container name="main" label="Main
                            Content Container" htmlTag="div"
                            htmlClass="column main"/>
11                  </container>
12              </container>
13              <container name="page.bottom.container" as="
                    page_bottom_container" label="Before Page
                    Footer Container" after="main.content"
                    htmlTag="div" htmlClass="page-bottom"/>
14              <container name="before.body.end" as="
                    before_body_end" after="-" label="Page
                    Bottom"/>
15          </container>
16      </container>
17  </layout>
```

As we can see, an "empty" layout actually sets up a number of different containers. The above sets up a default hierarchy that looks like this

```
1   root
2       after.body.start
3       page.wrapper
4           global.notices
5           main.content
6               columns.top
7               columns
8                   main
9           page.bottom.container
10          before.body.end
```

As you can see, an empty layout already contains 10 individual containers for us to insert our blocks into. However – where's the content block?

Maybe it's in the layered navigation page layout XML file? While that would be a little weird, weirder things have happened in the Magento source. However, if we examine this second `empty.xml` file, there's no sign of a content block.

```
#File: vendor/magento/module-layered-navigation/view/
    frontend/page_layout/empty.xml
<layout xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xsi:noNamespaceSchemaLocation="urn:magento:
    framework:View/Layout/etc/page_layout.xsd">
    <move element="catalog.leftnav" destination="category.
        product.list.additional" before="-"/>
</layout>
```

This brings us to another example of second system syndrome in Magento 2. Although the page layout system was invented to set the default container structure for a page, the `content` block is actually added by a layout handle XML file.

```
#File: vendor/magento/module-theme/view/frontend/layout/
    default.xml
<referenceContainer name="main">
    <container name="content.top" label="Main Content Top"
        />
    <container name="content" label="Main Content Area"/>
    <container name="content.aside" label="Main Content
        Aside"/>
    <container name="content.bottom" label="Main Content
        Bottom"/>
</referenceContainer>
```

This `default` handle file (the `default` handle run on every page) gets a reference to the `main` container (this `main` container was created in the `empty.xml` page layout XML file), and adds four new containers `content.top`, `content`, `content.aside`, and `content.bottom`.

Why was `content`, a block that's available on all page of Magento 2, not added as part of the page layouts? Unfortunately, only the developers involved know, and while we can sure each individual developer has very good reasons for doing what they did here, the end result is a system where the implicit goal

> Pull page structure into individual page layout files

seems undermined by the actual implementation.

Container Features

Before we wrap-up our journey into the page layout system, let's take another look at the HTML source of the `our_first_container` container we rendered.

```
<em>Hello Blank Page.</em>
```

Container tags are, by default, just that: Containers to drop our blocks into. However, there's a few special attributes we can use with our container blocks that will impact the HTML created by our containers. For example, if we add a `htmlTag` attribute to our container, we can wrap the contents of the container in another tag. Try editing our `more-empty.xml` file so the `<container/>` looks like this (i.e. we've added a `htmlTag="div"`)

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    page_layout/more-empty.xml
<container
        name="our_first_container"
        htmlTag="div"  />
```

Clear the cache, reload the page, and the HTML rendered should now look like this

```
<div><em>Hello Blank Page.</em></div>
```

You'll need to be careful with this feature – Magento's XSD validation files only allow the following wrapper tags: `dd`, `div`, `dl`, `fieldset`, `main`, `header`, `footer`, `ol`, `p`, `section`, `table`, `tfoot`, `ul`, `nav`. Trying to wrap something in say, a span, will result in a fatal schema violation exception.

Once you have a wrapper tag in place, the `htmlId` and `htmlClass` attributes allow you to add an `id` and `class` to your wrapper tags. In other words, this

```
<container
        name="our_first_container"
        htmlTag="div"
        htmlId="our_id"
        htmlClass="our_id"
        />
```

results in HTML output that looks like this

```
<div id="our_id" class="our_class">
    <em>Hello Blank Page.</em>
</div>
```

With this new knowledge in mind, if we take another look at the main `empty.xml` page layout file

```
#File: vendor/magento/module-theme/view/base/page_layout/
    empty.xml
<layout xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xsi:noNamespaceSchemaLocation="urn:magento:
    framework:View/Layout/etc/page_layout.xsd">
    <container name="root">
```

```
4           <container name="after.body.start" as="after.body.
                start" before="-" label="Page Top"/>
5           <container name="page.wrapper" as="page_wrapper"
                htmlTag="div" htmlClass="page-wrapper">
6               <container name="global.notices" as="
                    global_notices" before="-"/>
7               <container name="main.content" htmlTag="main"
                    htmlId="maincontent" htmlClass="page-main">
8                   <container name="columns.top" label="Before
                         Main Columns"/>
9                   <container name="columns" htmlTag="div"
                        htmlClass="columns">
10                      <container name="main" label="Main
                            Content Container" htmlTag="div"
                            htmlClass="column main"/>
11                  </container>
12              </container>
13              <container name="page.bottom.container" as="
                    page_bottom_container" label="Before Page
                    Footer Container" after="main.content"
                    htmlTag="div" htmlClass="page-bottom"/>
14              <container name="before.body.end" as="
                    before_body_end" after="-" label="Page
                    Bottom"/>
15          </container>
16      </container>
17  </layout>
```

we'll see that `htmlTag` and `htmlClass` are used liberally throughout the layout. This explains where all those wrapper tags came from when we switched our layout to `empty.xml`.

Certified Backend Layout Developer

Now that we're four chapters in, you should have the basic architectural principles for understanding Magento's page layouts.

In our next chapter, we'll take a look at Magento's **theme** system. The intent of themes is to allow traditional HTML/CSS/Javascript developers to modify the look, feel, and behavior of a Magento web page. As we'll learn though, without a firm grasp of layout fundamentals, what you'll be able to do with themes is extremely limited.

Chapter 6

Magento 2 Theming

Theming systems occupy a strange space in web application frameworks. From a user facing perspective, it may seem obvious what themes are – they determine how a site looks and feels. However, on a technical level, a theming system is something that allows developers *limited* access to the system that outputs HTML and CSS for web pages. The question becomes **how** limited is that access.

Themes in Magento 2 are, technically speaking, pretty straight forward. A Magento theme allows theme developers to **replace** or **add to** assets added by Magento modules, or by their parent themes. These assets include

- Layout Handle XML files
- CSS/LessCSS files,
- Javascript source files
- Knockout.js html templates
- Email templates
- Additional requirejs-config.js configuration
- Additional translation strings

In this chapter we're going to create a new child theme, and then use that theme to replace/extend a few of Magento's stock layout handle xml files, phtml files, and LessCSS files.

Creating a Child Theme

Most developers will rarely create a new theme in Magento 2. Instead, you'll have your theme **inherit** from a theme that already exists. When you create a

child theme in Magento 2, you're telling Magento

> I want my theme to behave exactly the same as this other theme.

After you've created your child theme, you can customize it and make it behave *slightly* differently than the parent theme. This may mean some additional LessCSS rules, a different phtml template for one section, etc.

We're going to dive right in and create a new theme. The first thing we need to do is come up with a name and identifier for our theme. A theme's unique identifier is made up of three different parts.

```
1 [area]/[package-name]/[theme-name]
```

A theme's `area` is the Magento area this theme will apply to (See the appendix on areas if you're unfamiliar with the concept). The package name is a string that identifies the person or organization that created the theme. The theme's name is, well, the theme's name! Here's an example of an actual theme identifier for Magento's Luma theme

```
1 frontend/Magento/luma
```

The area, `frontend`, indicates this is the theme for the front end cart application. The package name, `Magento`, indicates this is a theme that comes from Magento Inc. The theme's name, `luma`, is (again), the theme's name.

A theme is a Magento component, which means a theme is meant to be distributed by Composer. This is why you can find the luma theme in the `vendor/magento/theme-frontend-luma` folder. In addition to distributing themes via Composer, Magento will also scan any folder that matches the following `glob` pattern for theme registration files

```
1 #File: app/etc/NonComposerComponentRegistration.php
2
3 /* ... */
4 $pathList[] = dirname(__DIR__) . '/design/*/*/*/
      registration.php';
5 /* ... */
```

If you're unfamiliar with components, checkout the component appendix for more information.

We're going to use the theme identifier `frontend/Pulsestorm/dram`. We're also going to create our theme in the `app/design` folder. We'll also have our theme inherit from the Luma theme. To create your new theme, first create a `registration.php` file in the following location

```
1 #File: app/design/frontend/Pulsestorm/dram/registration.php
2 <?php
```

```
    \Magento\Framework\Component\ComponentRegistrar::register(
        \Magento\Framework\Component\ComponentRegistrar::THEME,
        'frontend/Pulsestorm/dram',
        __DIR__
    );
```

Then, create a `theme.xml` file

```
#File: app/design/frontend/Pulsestorm/dram/theme.xml
<?xml version="1.0"?>
<theme xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="urn:magento:framework:Config/etc/theme.xsd">
    <title>Pulsestorm Dram</title>
    <parent>Magento/luma</parent>
    <media>
        <preview_image/>
    </media>
</theme>
```

The `registration.php` file is required by all Magento components. You'll notice we're using the `ComponentRegistrar::THEME` theme constant to tell Magento we're registering a theme, and the `frontend/Pulsestorm/dram` identifier we previously discussed.

Every theme file needs a `theme.xml` file. This file identifies the theme's parent, gives the theme a plain english title, and allows you to configure an (optional) preview image. With the above files in place, clear your cache, and navigate to the `Content -> Design -> Themes` section. You should see your theme listed in the theme grid.

```
Theme Title          Parent Theme         Theme Path
------------------------------------------------------------

Magento Luma         Magento Blank        Magento/luma
Pulsestorm Dram      Magento Luma         Pulsestorm/dram
Magento Blank                             Magento/blank
```

One important thing to note about this section: The first time Magento sees a new theme, it will parse the theme information and save it to the `theme` database table

```
mysql> select * from theme\G

/* ... */

```

```
*************************** 4. row
    ***************************
      theme_id: 4
     parent_id: 2
    theme_path: Pulsestorm/dram
   theme_title: Pulsestorm Dram
preview_image: NULL
  is_featured: 0
         area: frontend
         type: 0
         code: Pulsestorm/dram
```

Although your theme will always need its `theme.xml` file, from now on when Magento needs theme information it will look inside this table.

If you navigate to `Content -> Design -> Configuration`, you can select which store views should use this theme by clicking the `edit` button next to your store view, choosing *Pulsestorm Dram* from the Applied Theme drop down, and then clicking the `Save Configuration` button. Do this before moving on to the next section.

Adding Layout Handle XML Files

Now that we have a child theme, we can use that child theme to *add* layout handle XML files to our system. A theme's layout handle XML files work *slightly* differently than the module based layout handle XML files we've dealt with so far. While the files themselves contain the same XML, and the file name still refers to a specific Magento handle, in a theme

> Developers create layout handle XML files that are loaded immediately after **a specific module's** layout handle XML file

This means if a layout handle XML file doesn't exist in a module, a theme won't be able to use that handle. The upside is this new system allows you an unprecedented level of control over when Magento processes your layout handle XML files.

Let's get to the examples! If you load a category listing page in Magento 2

```
http://magento.example.com/gear/fitness-equipment.html
```

You'll see a list of products in a particular category. This page's full action handle is `catalog_category_view`, and the following XML file is responsible for adding the product listing blocks to a page.

```
vendor/magento/module-catalog/view/frontend/layout/
    catalog_category_view.xml
```

Using our `frontend/Pulsestorm/dram` theme, we can have Magento parse an **additional** layout handle XML file whenever Magento loads the catalog module's layout handle XML file. Just create the following layout handle XML file

```
#File: app/design/frontend/Pulsestorm/dram/Magento_Catalog/
    layout/catalog_category_view.xml
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="urn:magento:framework:
    View/Layout/etc/page_configuration.xsd">
    <body>
        <referenceContainer name="content">
            <block  name="
                pulsestorm_magento_catalog_category_view_example
                "
                    class="Magento\Framework\View\Element\
                        Template"
                    template="hello.phtml"/>
        </referenceContainer>
    </body>
</page>
```

along with the following template file

```
#File: app/design/frontend/Pulsestorm/dram/templates/hello.
    phtml
<h2>Hello Category Listing Page.</h2>
```

If you clear your cache, reload the category listing page, and have `frontend/Pulsestorm/dram` selected as the current front end theme, you should see the text *Hello Category Listing Page* below the product listings.

Why This Works

Before we get to the new things here, let's do a quick review of what we already know. Our layout handle XML file gets a reference to the already created container named `content`

```
#File: app/design/frontend/Pulsestorm/dram/Magento_Catalog/
    layout/catalog_category_view.xml
<referenceContainer name="content">
    <!-- ... -->
</referenceContainer>
```

and then adds a block named `pulsestorm_magento_catalog_category_view_example` to that container.

```
#File: app/design/frontend/Pulsestorm/dram/Magento_Catalog/
    layout/catalog_category_view.xml
<block  name="
    pulsestorm_magento_catalog_category_view_example"
        class="Magento\Framework\View\Element\Template"
        template="hello.phtml"/>
```

We can name this block anything we like, so long as the name is globally unique in the Magento layout.

This block's PHP class is `Magento\Framework\View\Element\Template`. Block objects created with a `Magento\Framework\View\Element\Template` class are simple template blocks. This block's template URN is `hello.phtml`.

What's new, and might be confusing, is

1. The layout handle XML file's path
2. The "module-less" template URN.

As previously mentioned, Magento loads a theme's layout handle XML *in addition to* a module's layout handle XML file. Since we're pairing our layout file with the following core file

```
vendor/magento/module-catalog/view/frontend/layout/
    catalog_category_view.xml
```

We placed our file in the following theme folder

```
path/to/theme/Magento_Catalog/layout
```

The `Magento_Catalog` portion of this path comes from the catalog module's component identifier. You can find this identifier by looking in the module's registration.php file.

```
#File: vendor/magento/module-catalog/registration.php

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

The `layout` portion of the path is just a hard-coded folder named layout. As we'll learn later, there are other asset types we can use in themes, so Magento forces us to put our layout handle XML files in the `layout` folder to keep things tidy.

Our file's name, `catalog_category_view.xml`, should match the name of the module layout handle XML file we're pairing our file with.

The other new thing is our template's URN/file-name.

```
#File: app/design/frontend/Pulsestorm/dram/Magento_Catalog/
    layout/catalog_category_view.xml

template="hello.phtml"
```

So far we're used to seeing templates with values like `Package_Module::path/to/template`. While you can use these sort of template URNs in your theme's layout handle XML file, if there's no module name in a template URN Magento will look *in the current theme's templates* folder for the file. For our example above, that means this file

```
#File: app/design/frontend/Pulsestorm/dram/templates/hello.
    phtml
<h2>Hello Category Listing Page.</h2>
```

This allows you to create completely new templates in your themes without needing an extra Magento module.

Replacing a Layout Handle XML File

Magento's theme system also allows you to **completely replace** a layout handle XML file. If, for some reason, we wanted to get rid of the catalog module's `catalog_category_view` rules, we could just add the following Layout Handle XML file.

```
#File: app/design/frontend/Pulsestorm/dram/Magento_Catalog/
    layout/override/base/catalog_category_view.xml
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="urn:magento:framework:
    View/Layout/etc/page_configuration.xsd">
    <body>
    </body>
</page>
```

Clear any category listing page with the above in place, and Magento will discard all the rules in the

```
vendor/magento/module-catalog/view/frontend/layout/
    catalog_category_view.xml
```

file and use the rules in `Magento_Catalog/layout/override/base/catalog_category_view.xml` instead. In other words, the page will no longer have a category listing.

Again, the `Magento_Catalog` portion of this path comes from the original module's registered component name. The `layout/override/base` path is hard coded.

The `override` folder tells Magento these files should **replace** the existing files. The `base` path component appears to be an area name, but you're required to use `base` when overriding a file. In other words, you can't use `frontend` or `area` here. Finally, the file's name (`catalog_category_view.xml`) is the file we want to replace.

Generally speaking, you'll want to avoid using an override unless it's the only way to achieve your goal. Many of Magento's core layout files create blocks that **other** layout files expect to find. While Magento will swallow these sort of layout errors in log files (i.e. Magento won't crash), your system's front end code will probably behave in strange ways with so much expected rendering HTML code missing.

Replacing a Template

While it's usually a better practice to leave Magento's rendered HTML as is, sometimes the only way to achieve the front end result we want is to replace a template file with our own. Magento's theming system allows you to replace **any** template you'd like with your own.

For example, consider the product attribute template

```
1 #File: vendor/magento/module-catalog/view/frontend/
      templates/product/view/attribute.phtml
2 /* ... */
3 <?php if ($_attributeValue): ?>
4 <div class="product attribute <?php /* @escapeNotVerified
      */ echo $_className?>">
5     <?php if ($_attributeLabel != 'none'): ?><strong class=
          "type"><?php /* @escapeNotVerified */ echo
          $_attributeLabel?></strong><?php endif; ?>
6     <div class="value" <?php /* @escapeNotVerified */ echo
          $_attributeAddAttribute;?>><?php /*
          @escapeNotVerified */ echo $_attributeValue; ?></
          div>
7 </div>
8 <?php endif; ?>
```

Magento uses this template to output product attribute values on the product details page. Using blocks, there's no way for us to insert a value between the `class="product` div and the `<div class="value"` div. It's cases like these where template replacement usually makes sense.

If we create the following file in our theme

```
1 #File: app/design/frontend/Pulsestorm/dram/Magento_Catalog/
      templates/product/view/attribute.phtml
```

and replace its contents with a slightly modified version of the original

```
#File: app/design/frontend/Pulsestorm/dram/Magento_Catalog/
    templates/product/view/attribute.phtml
<?php
/**
 * Copyright © 2016 Magento. All rights reserved.
 * See COPYING.txt for license details.
 */

// @codingStandardsIgnoreFile

/**
 * Product view template
 *
 * @see \Magento\Catalog\Block\Product\View\Description
 */
?>
<?php
$_helper = $this->helper('Magento\Catalog\Helper\Output');
$_product = $block->getProduct();
$_call = $block->getAtCall();
$_code = $block->getAtCode();
$_className = $block->getCssClass();
$_attributeLabel = $block->getAtLabel();
$_attributeType = $block->getAtType();
$_attributeAddAttribute = $block->getAddAttribute();

if ($_attributeLabel && $_attributeLabel == 'default') {
    $_attributeLabel = $_product->getResource()->
        getAttribute($_code)->getFrontendLabel();
}
if ($_attributeType && $_attributeType == 'text') {
    $_attributeValue = ($_helper->productAttribute(
        $_product, $_product->$_call(), $_code)) ?
        $_product->getAttributeText($_code) : '';
} else {
    $_attributeValue = $_helper->productAttribute($_product
        , $_product->$_call(), $_code);
}
?>

<?php if ($_attributeValue): ?>
<div class="product attribute <?php /* @escapeNotVerified
    */ echo $_className?>">
    <h2>This is an attribute: </h2>
    <?php if ($_attributeLabel != 'none'): ?><strong class=
        "type"><?php /* @escapeNotVerified */ echo
        $_attributeLabel?></strong><?php endif; ?>
    <div class="value" <?php /* @escapeNotVerified */ echo
```

```
            $_attributeAddAttribute;?>><?php /*
            @escapeNotVerified */ echo $_attributeValue; ?></
            div>
41  </div>
42  <?php endif; ?>
```

we'll see the *This is an attribute* text added every time Magento uses the `attriute.phtml` template to output a product's attribute.

If we examine our file's path, we'll see a pattern similar to our layout handle XML files. First, of course, is our base theme folder

```
1  app/design/frontend/Pulsestorm/dram
```

This is followed by the `Magento_Catalog` folder

```
1  Magento_Catalog
```

`Magento_Catalog` comes from the catalog module's registered name in its `registration.php` file. We're using the catalog module because that's where the core `attribute.phtml` file lives. Next, we have a

```
1  templates
```

folder. Like the `layout` folder, this folder exists to separate templates from other file types. Finally, we have the path to the file we want to replace

```
1  product/view/attribute.phtml
```

Again, like layout files, this path matches the path to the file in the original module, minus the area.

While our *This is an attribute* text is a somewhat silly example, replacing a template is often the only way to apply subtle changes to Magento's HTML output. For example, in this HTML chunk

```
1  #File: vendor/magento/module-catalog/view/frontend/
       templates/product/view/attribute.phtml
2  <div class="value" <?php /* @escapeNotVerified */ echo
       $_attributeAddAttribute;?>>
3       <?php /* @escapeNotVerified */ echo $_attributeValue;
            ?>
4  </div>
```

If a developer (you!) wanted to add an additional CSS class along-side `value`, the only way to do it would be to replace the `attribute.phtml` with your own version.

The downside of template replacement is, if Magento changes a template significantly during a upgrade release, your new template may not work right with the new system. Prior to an upgrade you'll need to test your theme with the new version of Magento to make sure your changes and Magento's changes play nice together.

Replacing CSS and LessCSS Files

The last file replacement technique we'll discuss is replacing a LessCSS file. However, to do this, we'll need to discuss how CSS files are handled in a stock version of Magento.

If you look at a page rendered with Magento's stock Luma theme, you'll see the following `<link/>` tag

```
<link   rel="stylesheet"
        type="text/css"
        media="screen and (min-width: 768px)"
        href="http://magento.example.com/.../   \;
              frontend/Pulsestorm/dram/en_US/css/styles-l.
                  css" />
```

That is, Magento loads in a CSS file named `styles-l.css`. Magento does this thanks to the following layout handle XML file.

```
#File: vendor/magento/theme-frontend-blank/Magento_Theme/
    layout/default_head_blocks.xml
<head>
    <!-- ... --->
    <css src="css/styles-l.css" media="screen and (min-
        width: 768px)"/>
    <!-- ... --->
</head>
```

However, if you search the Magento core for a file named `styles-l.css`, you won't find one. However, you **will** find a file named `styles-l.less`.

```
#File: vendor/magento//theme-frontend-blank/web/css/styles-
    l.less

/* ... a very long list of less rules ... */
```

This is the first thing we'll need to cover. While you can use the `<head/>` section of a layout update XML file to add a traditional CSS file, you can **also** use it to add a LessCSS file. However, to do so, you don't say

```
<!-- this is wrong -->
<css src="css/styles-l.less" media="screen and (min-width:
    768px)"/>
```

You say

```
<css src="css/styles-l.css" media="screen and (min-width:
    768px)"/>
```

Although you've provided a path to a CSS file, Magento knows to check for a `.less` file. Regardless of whether it's a `.less` file, or a `.css` file, you can replace either file in your theme. If you wanted to replace the entire `styles-l.less` file, you'd just create a file with one of the following names

```
#File: app/design/frontend/Pulsestorm/dram/web/css/styles-l
    .css
# or
#File: app/design/frontend/Pulsestorm/dram/web/css/styles-l
    .less
body{
    background-color:#f00;
}
```

If you clear your Magento cache, remove the LESS preprocessed cache folder (`var/view_preprocessed`), remove any generated `style-l.css` file from your `pub` directory

```
$ find pub/ -name styles-l.css
pub/static/frontend/Pulsestorm/dram/en_US/css/styles-l.css

$ rm pub/static/frontend/Pulsestorm/dram/en_US/css/styles-l
    .css
```

and then reload the page (or reload the CSS file URL), we'll see our simple file has replaced the stock `styles-l.css`.

You'll notice that, unlike our layout handle XML files and phtml template files, we **do not** need to put this file in a `Packagename_Modulename` folder

```
web/css/styles-l.css
```

This is because the layout update XML file rules use a file URN without a module name (`css/styles-l.css`).

```
<css src="css/styles-l.css" media="screen and (min-width:
    768px)"/>
```

If this URN had looked like `Magento_Theme::css/styles-l.css`, then we would have put our file in the following location

```
app/design/frontend/Pulsestorm/dram/Magento_Theme/web/css/
    styles-l.css
```

Before we move on, be sure to remove either of the files you created previously

```
app/design/frontend/Pulsestorm/dram/web/css/styles-l.css
app/design/frontend/Pulsestorm/dram/web/css/styles-l.less
```

Practical LessCSS Replacement

While the above technique works, it is a bit of a cudgel. Replacing the **entire** `styles-l.less` file with a custom CSS file means replacing **all** the rules for the entire base Magento theme. While that may be a necessary step if you're creating a new theme that implements a **different** CSS preprocessor (such as Sass), it would be a tremendous amount of work for more simple theme customizations.

While you can always add a new CSS/Less file via a layout update XML handle, Magento also has a solution for less invasive LessCSS customizations. To understand this system, we'll need to take a look at the stock `styles-l.less` file.

Throughout `styles-l.less`, you'll see sections that look like the following

```
#File: vendor/magento/theme-frontend-blank/web/css/styles-l
    .less

//@magento_import 'source/_widgets.less'; // Theme widgets
```

These lines may *look* like comments, but they're not. The `//@magento_import` command is a custom LessCSS rule that Magento's parser picks up on. In plain english,

> The `//@magento_import` rules tell LessCSS to search through every active Magento library, module, and theme and then run `import` on the Less rules it finds for the passed in file name

So, the source `source/_widgets.less` line above actually loads in the following core LessCSS rules.

```
./vendor/magento/magento2-base/lib/web/css/source/_widgets.
    less
./vendor/magento/theme-frontend-blank/
    Magento_AdvancedCheckout/web/css/source/_widgets.less
```

```
./vendor/magento/theme-frontend-blank/Magento_Banner/web/
    css/source/_widgets.less
./vendor/magento/theme-frontend-blank/Magento_Catalog/web/
    css/source/_widgets.less
./vendor/magento/theme-frontend-blank/Magento_CatalogEvent/
    web/css/source/_widgets.less
./vendor/magento/theme-frontend-blank/Magento_Cms/web/css/
    source/_widgets.less
./vendor/magento/theme-frontend-blank/Magento_GiftRegistry/
    web/css/source/_widgets.less
./vendor/magento/theme-frontend-blank/
    Magento_MultipleWishlist/web/css/source/_widgets.less
./vendor/magento/theme-frontend-blank/Magento_Reports/web/
    css/source/_widgets.less
./vendor/magento/theme-frontend-blank/Magento_Sales/web/css
    /source/_widgets.less
./vendor/magento/theme-frontend-blank/Magento_VersionsCms/
    web/css/source/_widgets.less
./vendor/magento/theme-frontend-luma/
    Magento_AdvancedCheckout/web/css/source/_widgets.less
```

We can add our own files to this list via our theme. For example, try adding the following file to our theme.

```
#File: app/design/frontend/Pulsestorm/dram/web/css/source/
    _widget.less
body.example-less-rule-base-css-folder {
  background-color: #f00;
}
```

Again, we're creating this file in our theme's `web/css` folder, and then matching the path provided in the `magento_import` rule (`_widget.less`). If we clear our cache, clear our preprocessed view files, delete the existing `styles-l.css` files from `pub`, and reload the CSS file's URL (which will look something like the following (view your document's source to find your own link).

```
http://magento.example.com/static/version1492893430/ \;
       frontend/Pulsestorm/dram/en_US/css/styles-l.css
```

We'll see Magento's added our example rule to the generated file.

```
/* Generated styles-l.css file */
/* ... */
@media only screen and (min-device-width: 320px) and (max-
    device-width: 780px) and (orientation: landscape) {
  .product-video {
    height: 100%;
    width: 81%;
  }
```

```
8  }
9  body.example-less-rule-base-css-folder {
10   background-color: #f00;
11 }
12 @media all and (min-width: 768px), print {
13   .abs-product-options-list-desktop dt,
14
15 /* ... */
```

In addition to adding a file to our theme's `web/css` folder, we can also add a rule to **any** valid module folder in our theme. For example, create the following file at the following location.

```
1  #File: app/design/frontend/Pulsestorm/dram/Magento_Catalog/
       web/css/source/_widgets.less
2  body.example-less-rule-in-mage-catalog{
3      background-color:#f00;
4  }
```

Clear the various caches and remove the already generated files, and then reload the styles-l.css URL. You should see this new CSS rule added to the file

```
1  body.example-less-rule-base-css-folder {
2    background-color: #f00;
3  }
4  body.example-less-rule-in-mage-catalog {
5    background-color: #f00;
6  }
```

Replacing Less Files

Magento's themes do not offer a way to replace/remote files imported via the `// @magento_import` directive. However, for files imported using the built-in `@import` command

```
1  #File: vendor/magento/theme-frontend-blank/web/css/styles-l
       .less
2  @import '_styles.less';
3  //...
4  @import 'source/_theme.less';
```

A theme developer *can* replace these files completely by adding a file to their theme's `web/css` folder.

```
1  app/design/frontend/Pulsestorm/dram/web/css/_styles.less
```

```
2 app/design/frontend/Pulsestorm/dram/web/css/source/_theme.
      less
```

Be careful when replacing a less file completely. If you remove an important variable declaration you may prevent the system from successfully compiling your your less files to CSS. Also, as with any file replacement technique, future Magento upgrades may change the structure of the original file in such a way that the system is no longer compatible with your changes. When upgrades come down the pipe, be sure to test your changes.

Wrap Up

That's all we have time for with Magento's themes. While we haven't covered every possible file type you can replace/extend with a theme, the patterns we've described above will hold true when you're trying to replace other files with a Magento theme.

Themes are the inflection point where we can transition from backend PHP systems to front end, browser based, code. Before we jump completely to the front end systems, we want to take a **very** deep dive in layout file loading in Magento 2. Our next chapter is the most optional one in this book – feel free to skip ahead to Chapter 7 if you're not quite ready for the deep dive it offers.

Chapter 7

Advanced XML Loading

Compared to Magento 1, the code behind Magento 2's layout rendering is woefully complicated. There's no clear story to be told about *how* a Magento 2 page renders. There is, perhaps, a story to be told about what happens when a team of engineers sets out to refactor and enhancing a domain specific language they neither use nor fully understand. That is, however, a story for another day.

While Magento 2's layout system uses many of the same objects as Magento 1's layout system and still uses XML as its medium for a domain specific language, *how* Magento 2 uses these objects has changed dramatically. There are also new, similarly named objects that perform new tasks, heretofore unseen in the world of Magento layouts.

This preamble is to warn you that this section will be both difficult and rarely necessary for day-to-day work. Skip ahead to the front end chapters if you like.

Layout Merge/Update Object

In Magento 1, the most important layout object was the `Mage::getSingleton(core/layout)` object. This object lives on in Magento 2 as the `Magento\Framework\View\Layout` object, but it's no longer the most important layout object.

Instead, if you want to understand **how** a layout gets built in Magento, you need to understand a slew of different classes.

Layout files are loaded by the `Magento\Framework\View\Model\Layout\Merge` objects. Code from this class has **two** responsibilities. First, it loads **all** possible layout handle XML files into a single XML tree that looks something like this

```
1 <layouts xmlns:xsi="http://www.w3.org/2001/XMLSchema-
      instance">
2     <handle id="...">
```

```
        <!-- ... -->
    </handle>
    <handle id="...">
        <!-- ... -->
    </handle>

    <!-- etc... -->
</layouts>
```

This is all the layout handle XML files in modules, and all the layout handle XML files in the current theme. This also includes the `page_layout` files we learned about in Chapter 4.

Each individual file is represented by a single `<handle id="...">` node. The bulk of this work is done here.

```
#File: vendor/magento/framework/View/Model/Layout/Merge.php
protected function _loadFileLayoutUpdatesXml()
{
    //...
}
```

We'll call this gigantic tree of nodes the *Global XML Handle Tree*.

There's one other job for `Magento\Framework\View\Model\Layout\Merge` objects. These objects can return portions of the the Global XML Handle Tree based on handle ID. Programmers tell this object

> Please give me the XML nodes for the following handles: `default`, `customer_account_login`, etc.

and the object returns *just* those XML nodes that are inside `<handle/>` nodes with matching IDs. Each returned node is known individually as an "update". This work is kicked off via the `load` method.

```
#File: vendor/magento/framework/View/Model/Layout/Merge.php
public function load($handles = [])
{
    if (is_string($handles)) {
        $handles = [$handles];
    } elseif (!is_array($handles)) {
        throw new \Magento\Framework\Exception\
            LocalizedException(
            new \Magento\Framework\Phrase('Invalid layout
                update handle')
        );
    }

    $this->addHandle($handles);

```

```
14      /* ... */
15
16      foreach ($this->getHandles() as $handle) {
17          $this->_merge($handle);
18      }
19
20      $layout = $this->asString();
21      /* ... */
22  }
```

and then continues in the `_merge` method.

```
1   #File: vendor/magento/framework/View/Model/Layout/Merge.php
2   protected function _merge($handle)
3   {
4       if (!isset($this->allHandles[$handle])) {
5           $this->allHandles[$handle] = $this->
                handleProcessing;
6           $this->_fetchPackageLayoutUpdates($handle);
7           $this->_fetchDbLayoutUpdates($handle);
8           $this->allHandles[$handle] = $this->handleProcessed
                ;
9       } elseif ($this->allHandles[$handle] == $this->
            handleProcessing
10          && $this->appState->getMode() === \Magento\
                Framework\App\State::MODE_DEVELOPER
11      ) {
12          $this->logger->info('Cyclic dependency in merged
                layout for handle: ' . $handle);
13      }
14      return $this;
15  }
```

After calling `load`, the `Magento\Framework\View\Model\Layout\Merge` object will have a populated `->updates` property, which is an array of XML strings. Programmers can fetch these results using the `Magento\Framework\View\Model\Layout\Merge` object's `asArray`, `asString`, or `asSimplexml` methods. The resulting set of nodes is very similar to the Page Layout XML tree in Magento 1, so we'll continue to call it the Page Layout XML tree.

Reading XML, Generating Blocks

So far, all this is very similar to what happens in Magento 1. There's small differences – instead of multiple handles in a single layout file, Magento 2 uses a file's name as a handle identifier for the nodes inside. However, the Global XML Handle Tree is still very similar to Magento 1's Package Layout XML tree.

At this point, if this were Magento 1, the `Mage::getSingleton(core/layout);` object would start parsing through the Page Layout XML tree and recursively generate blocks, and call action method nodes.

```
#File: app/code/core/Mage/Core/Model/Layout.php
public function generateBlocks($parent=null)
{
    if (empty($parent)) {
        $parent = $this->getNode();
    }
    foreach ($parent as $node) {
        $attributes = $node->attributes();
        if ((bool)$attributes->ignore) {
            continue;
        }
        switch ($node->getName()) {
            case 'block':
                $this->_generateBlock($node, $parent);
                $this->generateBlocks($node);
                break;

            case 'reference':
                $this->generateBlocks($node);
                break;

            case 'action':
                $this->_generateAction($node, $parent);
                break;
        }
    }
}
```

Magento 2 has the same goal – a set of blocks, organized in a tree structure, with a single root block at the top. However, it takes the long way to get there. Magento 2 takes the Page Layout XML tree from the `Magento\Framework\View\Model\Layout\Merge` object and

1. Hands it off to a series of reader objects, organized under a ReaderPool.
2. Each reader object will, based on what they find inside the Page Layout XML tree, call methods on the `Magento\Framework\View\Layout\ScheduledStructure` object. This schedules certain actions for later.
3. Once Magento has read the XML files and scheduled actions, control is passed to a series of Generator objects, organized under a GeneratorPool.
4. Generators use the information in the `Magento\Framework\View\Layout\ScheduledStructure` block to do three things.
5. First, generators create block objects.

6. Second, generators keep track of the parent block relationships between blocks and containers via the `Magento\Framework\View\Layout\Data\Structure` object.
7. Third, for non-block/container features (front end assets, page title, etc), the generators make note of this information in the `Magento\Framework\View\Page\Config\Structure` object.

Phew! Our apologies. That's a lot of information to dump on you at once. Don't worry if you didn't follow it all at first. We'll try to get to everything. Before we do that though – once Magento has done all of the above, Magento will use the structure and layout objects we've mentioned to populate some view variables

```
1  #File: vendor/magento/framework/View/Result/Page.php
2  protected function render(ResponseInterface $response)
3  {
4      $this->pageConfig->publicBuild();
5      if ($this->getPageLayout()) {
6          $config = $this->getConfig();
7          $this->addDefaultBodyClasses();
8          $addBlock = $this->getLayout()->getBlock('head.
               additional'); // todo
9          $requireJs = $this->getLayout()->getBlock('require.
               js');
10         $this->assign([
11             'requireJs' => $requireJs ? $requireJs->toHtml
                   () : null,
12             'headContent' => $this->pageConfigRenderer->
                   renderHeadContent(),
13             'headAdditional' => $addBlock ? $addBlock->
                   toHtml() : null,
14             'htmlAttributes' => $this->pageConfigRenderer->
                   renderElementAttributes($config::
                   ELEMENT_TYPE_HTML),
15             'headAttributes' => $this->pageConfigRenderer->
                   renderElementAttributes($config::
                   ELEMENT_TYPE_HEAD),
16             'bodyAttributes' => $this->pageConfigRenderer->
                   renderElementAttributes($config::
                   ELEMENT_TYPE_BODY),
17             'loaderIcon' => $this->getViewFileUrl('images/
                   loader-2.gif'),
18         ]);
19
20         $output = $this->getLayout()->getOutput();
21         $this->assign('layoutContent', $output);
22         $output = $this->renderPage();
23         $this->translateInline->processResponseBody($output
               );
```

```
24          $response->appendBody($output);
25      } else {
26          parent::render($response);
27      }
28      return $this;
29  }
```

And then, in `renderPage`, use those variables to render a `phtml` template via a PHP `include`.

```
1   #File: vendor/magento/framework/View/Result/Page.php
2   protected function renderPage()
3   {
4       $fileName = $this->viewFileSystem->getTemplateFileName(
            $this->template);
5       if (!$fileName) {
6           throw new \InvalidArgumentException('Template "' .
                $this->template . '" is not found');
7       }
8
9       ob_start();
10      try {
11          extract($this->viewVars, EXTR_SKIP);
12          include $fileName;
13      } catch (\Exception $exception) {
14          ob_end_clean();
15          throw $exception;
16      }
17      $output = ob_get_clean();
18      return $output;
19  }
```

These are the same variables and templates we covered briefly in Chapter 4.

Reader Pools

After `Magento\Framework\View\Model\Layout\Merge` does its work, most of the action for rendering a Magento layout kicks off here

```
1   #File: vendor/magento/framework/View/Layout.php
2   public function generateElements()
3   {
4       /* ... */
5       if ($result) {
6           /* ... */
7       } else {
8           /* ... */
```

```
9           $this->readerPool->interpret($this->
                getReaderContext(), $this->getNode());
10          /* ... */
11      }
12
13      /* ... */
14
15      $this->generatorPool->process($this->getReaderContext()
            , $generatorContext);
16
17      /* ... */
18
19      $this->addToOutputRootContainers();
20      \Magento\Framework\Profiler::stop(__CLASS__ . '::' .
            __METHOD__);
21  }
```

We've commented out a lot of the extraneous code to draw attention to the reader pool

```
1  #File: vendor/magento/framework/View/Layout.php
2
3  $this->readerPool->interpret($this->getReaderContext(),
       $this->getNode());
```

the generator pool

```
1  #File: vendor/magento/framework/View/Layout.php
2
3  $this->generatorPool->process($this->getReaderContext(),
       $generatorContext);
```

and a final command that tells the layout object which structure blocks it should start rendering (which kicks off the rendering of children)

```
1  #File: vendor/magento/framework/View/Layout.php
2
3  $this->addToOutputRootContainers();
```

Examining the call to the readerPool's interpret method

```
1  #File: vendor/magento/framework/View/Layout.php
2
3  $this->readerPool->interpret($this->getReaderContext(),
       $this->getNode());
```

The `$this->getNode()` method returns the tree we've been calling the Page Layout XML tree. As a reminder, the Page Layout XML tree is the XML that

comes from loading **all** the layout related XML files in the system, and then plucking out nodes associated with a specific handle name. This tree will look something like the following

```
<?xml version="1.0"?>
<layout xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance">
    <body>
        <referenceContainer name="header.panel">
            <block class="Magento\Directory\Block\Currency"
                name="currency" before="store_language"
                template="currency.phtml"/>
        </referenceContainer>
    </body>

    <head>
        <meta name="viewport" content="width=device-width,
            initial-scale=1"/>
        <css src="mage/calendar.css"/>
        <script src="requirejs/require.js"/>
    </head>

    <body>
        <referenceContainer name="after.body.start">
            <block class="Magento\Framework\View\Element\Js\
                Components" name="head.components" as="
                components" template="Magento_Theme::js/
                components.phtml" before="-"/>
        </referenceContainer>
    </body>

    <!-- ... more nodes -->
</layout>
```

The `readerPool` property is a `Magento\Framework\View\Layout\ReaderPool` object. Its `interpret` method looks like the following

```
#File: vendor/magento/framework/View/Layout/ReaderPool.php

public function interpret(Reader\Context $readerContext,
    Layout\Element $element)
{
    $this->prepareReader($this->readers);
    /** @var $node Layout\Element */
    foreach ($element as $node) {
        $nodeName = $node->getName();
        if (!isset($this->nodeReaders[$nodeName])) {
            continue;
        }
```

```
12          /** @var $reader Layout\ReaderInterface */
13          $reader = $this->nodeReaders[$nodeName];
14          $reader->interpret($readerContext, $node, $element)
                ;
15      }
16      return $this;
17  }
```

Here's where things get a little weird. On the surface, this code is simple enough. The loop `foreach`es over the top level nodes of the layout update XML (`<body/>`, `<head/>`, and `<body/>` in the above example).

If there's no reader object in the `nodeReaders` array for the named node

```
1  #File: vendor/magento/framework/View/Layout/ReaderPool.php
2
3  // $nodeName = 'body';
4  if (!isset($this->nodeReaders[$nodeName])) {
5      continue;
6  }
```

then Magento skips on to the next one. Otherwise, Magento calls the reader's `interpret` method, passing along the context object, the specific child node (`$node`), and its parent (`$element`).

The `nodeReaders` object are configured via automatic constructor dependency injection, via a virtualType. The readerPool is injected into the `Magento\Framework\View\Layout` object here

```
1  #File: vendor/magento/magento2-base/app/etc/di.xml
2  <type name="Magento\Framework\View\Layout">
3      <arguments>
4          <argument name="readerPool" xsi:type="object"
               shared="false">commonRenderPool</argument>
5          <argument name="cache" xsi:type="object">Magento\
               Framework\App\Cache\Type\Layout</argument>
6      </arguments>
7  </type>
```

and the `commonRenderPool` virtual type is configured here.

```
1  #File: vendor/magento/magento2-base/app/etc/di.xml
2  <virtualType name="commonRenderPool" type="Magento\
       Framework\View\Layout\ReaderPool">
3      <arguments>
4          <argument name="readers" xsi:type="array">
5              <item name="html" xsi:type="string">Magento\
                   Framework\View\Page\Config\Reader\Html</
                   item>
```

```
6               <item name="head" xsi:type="string">Magento\
                    Framework\View\Page\Config\Reader\Head</
                    item>
7               <item name="body" xsi:type="string">Magento\
                    Framework\View\Page\Config\Reader\Body</
                    item>
8               <item name="container" xsi:type="string">
                    Magento\Framework\View\Layout\Reader\
                    Container</item>
9               <item name="block" xsi:type="string">Magento\
                    Framework\View\Layout\Reader\Block</item>
10              <item name="move" xsi:type="string">Magento\
                    Framework\View\Layout\Reader\Move</item>
11              <item name="uiComponent" xsi:type="string">
                    Magento\Framework\View\Layout\Reader\
                    UiComponent</item>
12          </argument>
13      </arguments>
14  </virtualType>
```

The readers are strings that the `prepareReader` method will use to to instantiate actual reader objects.

```
1  #File: vendor/magento/framework/View/Layout/ReaderPool.php
2
3  protected function prepareReader($readers)
4  {
5      if (empty($this->nodeReaders)) {
6          /** @var $reader Layout\ReaderInterface */
7          foreach ($readers as $readerClass) {
8              $reader = $this->readerFactory->create(
                   $readerClass);
9              $this->addReader($reader);
10         }
11     }
12 }
```

Also, each reader object can support multiple tag/node types (`container` and `containerReference` are just two examples). The `addReader` method makes sure that each reader object is indexed in the `nodeReaders` array via the node names it supports.

```
1  #File: vendor/magento/framework/View/Layout/ReaderPool.php
2  public function addReader(Layout\ReaderInterface $reader)
3  {
4      foreach ($reader->getSupportedNodes() as $nodeName) {
5          $this->nodeReaders[$nodeName] = $reader;
6      }
7      return $this;
```

```
8 }
```

This is pretty heavy on Object Manager concepts (which you can read about in the appendix), but not **too** bad, right? Let's take a look at one of those reader objects

```
1  public function interpret(
2      Layout\Reader\Context $readerContext,
3      Layout\Element $bodyElement
4  ) {
5      /** @var \Magento\Framework\View\Layout\Element
           $element */
6      foreach ($bodyElement as $element) {
7          if ($element->getName() === self::BODY_ATTRIBUTE) {
8              $this->setBodyAttributeToStructure(
                   $readerContext, $element);
9          }
10     }
11     $this->readerPool->interpret($readerContext,
           $bodyElement);
12     return $this;
13 }
```

This is the code Magento uses to parse the `<body/>` nodes of the layout updates. When it's done parsing, you'll notice a call here

```
1  $this->readerPool->interpret($readerContext, $bodyElement);
```

The individual `Magento\Framework\View\Page\Config\Reader\Body` object has **it's own** `ReaderPool` object. Your instincts might be that this is the same `ReaderPool` object as before. Those instincts are normal, but in this case, incorrect. While the injected object uses the same `Magento\Framework\View\Layout\ReaderPool` class

```
1  #File: vendor/magento/framework/View/Page/Config/Reader/
       Body.php
2  public function __construct(
3      \Magento\Framework\View\Layout\ReaderPool $readerPool)
4  {
5      $this->readerPool = $readerPool;
6  }
```

Thanks to some more virtual type shenanigans, this reader pool uses a different virtual type (`bodyRenderPool`).

```
1  #File: vendor/magento/magento2-base/app/etc/di.xml
2  <type name="Magento\Framework\View\Page\Config\Reader\Body"
       >
```

```
    <arguments>
        <argument name="readerPool" xsi:type="object">
            bodyRenderPool</argument>
    </arguments>
</type>
```

That virtual type's definition configures a **different** set of reader pools

```
#File: vendor/magento/magento2-base/app/etc/di.xml
<virtualType name="bodyRenderPool" type="Magento\Framework\
    View\Layout\ReaderPool">
    <arguments>
        <argument name="readers" xsi:type="array">
            <item name="container" xsi:type="string">
                Magento\Framework\View\Layout\Reader\
                Container</item>
            <item name="block" xsi:type="string">Magento\
                Framework\View\Layout\Reader\Block</item>
            <item name="move" xsi:type="string">Magento\
                Framework\View\Layout\Reader\Move</item>
            <item name="uiComponent" xsi:type="string">
                Magento\Framework\View\Layout\Reader\
                UiComponent</item>
        </argument>
    </arguments>
</virtualType>
```

This means nodes inside the `<body/>` node will only be scanned by the container, block, move, and uiComponent readers.

Many of the other readers also have their own reader pools. A reader's reader pool can contain a reference to itself, which is how Magento reads nested `container` and nested `block` configurations.

Whether using a reader pool and individual readers to parse through the XML files is *the right* thing to do is a judgment call. One thing that's clear though – this system makes full use of Magento's object manager and automatic constructor dependency injection systems, and can be particularly difficult to debug with so much object instantiation happening magically for us.

What does a Reader Do?

Putting aside the complicated pool mechanics, what do work do reader objects do? Two things. First, depending on the data they find in the XML file, they call methods on a "Scheduled Structure" object.

```
#File: vendor/magento/framework/View/Layout/Reader/Block.
    php
protected function scheduleReference(
    Layout\ScheduledStructure $scheduledStructure,
    Layout\Element $currentElement
) {
    $elementName = $currentElement->getAttribute('name');
    $elementRemove = filter_var($currentElement->
        getAttribute('remove'), FILTER_VALIDATE_BOOLEAN);
    if ($elementRemove) {
        $scheduledStructure->setElementToRemoveList(
            $elementName);
    } else {
        $data = $scheduledStructure->
            getStructureElementData($elementName, []);
        $data['attributes'] = $this->mergeBlockAttributes(
            $data, $currentElement);
        $this->updateScheduledData($currentElement, $data);
        $this->evaluateArguments($currentElement, $data);
        $scheduledStructure->setStructureElementData(
            $elementName, $data);
    }
}
```

This `Magento\Framework\View\Layout\ScheduledStructure` object is a way for programmers to tell Magento what sort of block objects Magento should create when it creates the layout, or what sort of additional actions it should take w/r/t to those object. Regarding the later, the move reader has a good example of this

```
#File: vendor/magento/framework/View/Layout/Reader/Move.php
$scheduledStructure->setElementToMove(
    $elementName,
    [$destination, $siblingName, $isAfter, $alias]
);
```

When Magento parses through `<move/>` elements, it calls the `setElementToMove` method on the `Magento\Framework\View\Layout\ScheduledStructure`. This is like Magento writing a reminder down on a list

> And then, I'll move the element named foo under the block named bar

However, this is **only** a list of instructions to be completed later. The scheduled structure object is Magento's attempt to separate out out the **instructions** configured in the layout update XML files from the **actions** (creating a block, manipulating a block's children, etc.) to take.

The other thing a reader will do is manipulate the `Magento\Framework\View\Page\Config\Structure` object. This object keeps track of things like the page's title,

and its front end asset files

```
#File: vendor/magento/framework/View/Page/Config/Reader/
    Head.php
public function interpret(
    Layout\Reader\Context $readerContext,
    Layout\Element $headElement
) {
    $pageConfigStructure = $readerContext->
        getPageConfigStructure();
    /* ... */
    $pageConfigStructure->addAssets($node->getAttribute('
        src'),
        $this->getAttributes($node));
    /* ... */
}
```

When it comes time to render these elements later, Magento will reference the `Magento\Framework\View\Page\Config\Structure` object.

Generator Objects

If we return to our `GenerateElements` method

```
#File: vendor/magento/framework/View/Layout.php
public function generateElements()
{
    /* ... */
    if ($result) {
        /* ... */
    } else {
        /* ... */
        $this->readerPool->interpret($this->
            getReaderContext(), $this->getNode());
        /* ... */
    }

    /* ... */

    $this->generatorPool->process($this->getReaderContext()
        , $generatorContext);

    /* ... */

    $this->addToOutputRootContainers();
    \Magento\Framework\Profiler::stop(__CLASS__ . '::' .
        __METHOD__);
```

```
21 }
```

Once Magento's finished running through the reader pools, it turns over control to the generator pool

```
1 #File: vendor/magento/framework/View/Layout.php
2
3 $this->generatorPool->process($this->getReaderContext(),
      $generatorContext);
```

The generator pool is a `Magento\Framework\View\Layout\GeneratorPool` object. If we take a look at its `process` method,

```
1 #File: vendor/magento/framework/View/Layout/GeneratorPool.
      php
2 public function process(Reader\Context $readerContext,
      Generator\Context $generatorContext)
3 {
4     $this->buildStructure($readerContext->
          getScheduledStructure(), $generatorContext->
          getStructure());
5     foreach ($this->generators as $generator) {
6         $generator->process($readerContext,
              $generatorContext);
7     }
8     return $this;
9 }
```

we'll see a `foreach` loop that runs through an array of generator objects and calls `process` on each one of them. However, before we investigate the generators in the pool, we should look at the call to `buildStructure`

```
1 #File: vendor/magento/framework/View/Layout/GeneratorPool.
      php
2 protected function buildStructure(ScheduledStructure
      $scheduledStructure, Data\Structure $structure)
3 {
4     //Schedule all element into nested structure
5     while (false === $scheduledStructure->isStructureEmpty
          ()) {
6         $this->helper->scheduleElement($scheduledStructure,
               $structure, key($scheduledStructure->
              getStructure()));
7     }
8     $scheduledStructure->flushPaths();
9     while (false === $scheduledStructure->isListToSortEmpty
          ()) {
10        $this->reorderElements($scheduledStructure,
              $structure, key($scheduledStructure->
```

```
                getListToSort()));
11      }
12      foreach ($scheduledStructure->getListToMove() as
            $elementToMove) {
13          $this->moveElementInStructure($scheduledStructure,
                $structure, $elementToMove);
14      }
15      foreach ($scheduledStructure->getListToRemove() as
            $elementToRemove) {
16          $this->removeElement($scheduledStructure,
                $structure, $elementToRemove);
17      }
18      foreach ($scheduledStructure->getIfconfigList() as
            $elementToCheckConfig) {
19          list($configPath, $scopeType) = $scheduledStructure
                ->getIfconfigElement($elementToCheckConfig);
20          if (!empty($configPath)
21              && !$this->scopeConfig->isSetFlag($configPath,
                    $scopeType, $this->scopeResolver->getScope
                    ())
22          ) {
23              $this->removeIfConfigElement(
                    $scheduledStructure, $structure,
                    $elementToCheckConfig);
24          }
25      }
26      return $this;
27  }
```

The `$scheduledStructure` object is the object populated during the runs through the readerPool. The `$structure` object is something we haven't see yet, a `Magento\Framework\View\Layout\Data\Structure` object. This structure object is where Magento 2 keeps track of a layout's **structure**. In Magento 1, the parent/child relationships between blocks were kept track of in the block objects themselves – with each block having a reference to its child blocks. Magento 2 has moved that data that tracks these relationships out of the blocks and into the `Magento\Framework\View\Layout\Data\Structure` object.

In the `buildStructure` method mentioned above, Magento reads through the list of instructions built up during the reader pool run (stored in the `Magento\Framework\View\Layout\ScheduledStructure` object), and runs those instructions against the `Magento\Framework\View\Layout\Data\Structure` object.

The first while loops runs through the scheduled structure and creates each element in the actual structure object (via `scheduleElement`).

```
1   #File: vendor/magento/framework/View/Layout/GeneratorPool.
        php
2
```

```
3  while (false === $scheduledStructure->isStructureEmpty()) {
4      $this->helper->scheduleElement($scheduledStructure,
           $structure, key($scheduledStructure->getStructure()
           ));
5  }
```

Then Magento runs through any requests to reorder elements

```
1  #File: vendor/magento/framework/View/Layout/GeneratorPool.
       php
2
3  while (false === $scheduledStructure->isListToSortEmpty())
       {
4      $this->reorderElements($scheduledStructure, $structure,
           key($scheduledStructure->getListToSort()));
5  }
```

Then Magento runs through any requests to move an element from one place in the layout to another

```
1  #File: vendor/magento/framework/View/Layout/GeneratorPool.
       php
2
3  foreach ($scheduledStructure->getListToMove() as
       $elementToMove) {
4      $this->moveElementInStructure($scheduledStructure,
           $structure, $elementToMove);
5  }
```

Then Magento runs through any requests to remove any elements

```
1  #File: vendor/magento/framework/View/Layout/GeneratorPool.
       php
2
3  foreach ($scheduledStructure->getListToRemove() as
       $elementToRemove) {
4      $this->removeElement($scheduledStructure, $structure,
           $elementToRemove);
5  }
```

Finally, Magento runs through any requests to remove elements that fail an `ifconfig` check.

```
1  #File: vendor/magento/framework/View/Layout/GeneratorPool.
       php
2
3  foreach ($scheduledStructure->getIfconfigList() as
       $elementToCheckConfig) {
```

```
4       list($configPath, $scopeType) = $scheduledStructure->
            getIfconfigElement($elementToCheckConfig);
5       if (!empty($configPath)
6           && !$this->scopeConfig->isSetFlag($configPath,
                $scopeType, $this->scopeResolver->getScope())
7       ) {
8           $this->removeIfConfigElement($scheduledStructure,
                $structure, $elementToCheckConfig);
9       }
10  }
```

In each of those helper methods, Magento manipulates the `Magento\Framework\View\Layout\Data\Structure` object. This `Magento\Framework\View\Layout\Data\Structure` object becomes the "source of truth" for what blocks are and aren't a part of Magento's layout.

The Generator Pools

One the `buildStructure` method finishes, the `process` method runs through each generator object

```
1   #File: vendor/magento/framework/View/Layout/GeneratorPool.
        php
2   public function process(Reader\Context $readerContext,
        Generator\Context $generatorContext)
3   {
4       $this->buildStructure($readerContext->
            getScheduledStructure(), $generatorContext->
            getStructure());
5       foreach ($this->generators as $generator) {
6           $generator->process($readerContext,
                $generatorContext);
7       }
8       return $this;
9   }
```

Generators are configured via the dependency injection system – take note that these are `xsi:type="object"` items, meaning Magento will automatically instantiate these objects for us

```
1   #File: vendor/magento/magento2-base/app/etc/di.xml
2   <type name="Magento\Framework\View\Layout\GeneratorPool">
3       <arguments>
4           <argument name="generators" xsi:type="array">
5               <item name="head" xsi:type="object">Magento\
                    Framework\View\Page\Config\Generator\Head</
                    item>
```

```
6               <item name="body" xsi:type="object">Magento\
                    Framework\View\Page\Config\Generator\Body</
                    item>
7               <item name="block" xsi:type="object">Magento\
                    Framework\View\Layout\Generator\Block</item
                    >
8               <item name="container" xsi:type="object">
                    Magento\Framework\View\Layout\Generator\
                    Container</item>
9               <item name="uiComponent" xsi:type="object">
                    Magento\Framework\View\Layout\Generator\
                    UiComponent</item>
10          </argument>
11      </arguments>
12  </type>
```

By and large, these generators make additional adjustments to the `Magento\Framework\View\Layout\Data\Structure` object. The one big exception is the `Magento\Framework\View\Layout\Generator\Block` generator. This generator is the one that actually instantiates the block objects. This starts in the `generateBlock` method

```
1   #File: vendor/magento/framework/View/Layout/Generator/Block
        .php
2   protected function generateBlock(
3       Layout\ScheduledStructure $scheduledStructure,
4       Layout\Data\Structure $structure,
5       $elementName
6   ) {
7       list(, $data) = $scheduledStructure->getElement(
            $elementName);
8       $attributes = $data['attributes'];
9
10      if (!empty($attributes['group'])) {
11          $structure->addToParentGroup($elementName,
                $attributes['group']);
12      }
13      if (!empty($attributes['display'])) {
14          $structure->setAttribute($elementName, 'display',
                $attributes['display']);
15      }
16
17      // create block
18      $className = $attributes['class'];
19      $block = $this->createBlock($className, $elementName, [
20          'data' => $this->evaluateArguments($data['arguments
                '])
21      ]);
22      if (!empty($attributes['template'])) {
```

```
23          $block->setTemplate($attributes['template']);
24      }
25      if (!empty($attributes['ttl'])) {
26          $ttl = (int)$attributes['ttl'];
27          $block->setTtl($ttl);
28      }
29      return $block;
30  }
```

It's worth tracing this method down through `createBlock`, and then `getBlockInstance`, but we'll leave that as an exercise for the reader. Also, notice how the generator uses information from the scheduled structure to manipulate the actual structure object – this is typical of the work done by the generators.

The Structure Object

If you've only interacted with the layout via XML files and blocks, you may be surprised to discover it's the structure object that's keeping track of everything. However, if you look at a method you may have used from your block code – `getChildBlock`

```
1  #File: vendor/magento/framework/View/Layout.php
2  public function getChildBlock($parentName, $alias)
3  {
4      $this->build();
5      $name = $this->structure->getChildId($parentName,
           $alias);
6      if ($this->isBlock($name)) {
7          return $this->getBlock($name);
8      }
9      return false;
10 }
```

you'll see the `$this->structure` object is the same `Magento\Framework\View\Layout\Data\Structure` object we've been talking about above. Magento uses the structure object to fetch the child block's real name, and then returns the block.

Returning one last time to our `generateElements` method

```
1  #File: vendor/magento/framework/View/Layout.php
2
3  public function generateElements()
4  {
5      /* ... */
6      if ($result) {
7          /* ... */
```

```
8       } else {
9           /* ... */
10          $this->readerPool->interpret($this->
                getReaderContext(), $this->getNode());
11          /* ... */
12      }
13
14      /* ... */
15
16      $this->generatorPool->process($this->getReaderContext()
            , $generatorContext);
17
18      /* ... */
19
20      $this->addToOutputRootContainers();
21      \Magento\Framework\Profiler::stop(__CLASS__ . '::' .
            __METHOD__);
22  }
```

We have the final `$this->addToOutputRootContainers();` to consider.

```
1   #File: vendor/magento/framework/View/Layout.php
2
3   protected function addToOutputRootContainers()
4   {
5       foreach ($this->structure->exportElements() as $name =>
             $element) {
6           if ($element['type'] === Element::TYPE_CONTAINER &&
                 empty($element['parent'])) {
7               $this->addOutputElement($name);
8           }
9       }
10      return $this;
11  }
12  /* ... */
13  public function addOutputElement($name)
14  {
15      $this->_output[$name] = $name;
16      return $this;
17  }
```

Once the generators have finished creating the structure object, the `addToOutputRootContainers` will fetch every top-level *container* element and add them to the `_output` array. This is the array of elements Magento will output when it's time to render the page

```
1   #File: vendor/magento/framework/View/Layout.php
2   public function getOutput()
3   {
```

```
4       $this->build();
5       $out = '';
6       foreach ($this->_output as $name) {
7           $out .= $this->renderElement($name);
8       }
9       return $out;
10  }
```

Remember, rendering a container means rendering every element inside of it. Rendering a block also means rendering its child blocks, so these root containers kick-off rendering of *the entire* layout tree.

More Gotchas

That was a bit of a long and winding road, wasn't it? Well, we're not quite done yet. There's one other major change to how Magento makes use of the layout system. In Magento 1, loading layout XML was a one time, singular event for each HTTP page request. However, in Magento 2, most pages load **two** sets of layout handle XML files. Some load even more.

Take a break. Drink some tea. Pet your cat. When you're ready and recharged, jump back in.

Page Layout and Regular Layout

In Magento 1, every page was separated out into sections like `content`, `before-body-end`, etc. Behind the scenes, these sections were a special sort of block – a `text/list` block (`Mage_Core_Text_List`). These Magento 1 blocks automatically render all their children.

As we've already learned, Magento replaced these `text/list` blocks with `<container/>` nodes. We've also learned that Magento took the additional step of separating out the containers that make up individual pages into their own page layout XML files.

```
1  $ ls -1 vendor/magento/module-theme/view/base/page_layout \
2          vendor/magento/module-theme/view/frontend/
               page_layout
3
4  vendor/magento/module-theme/view/base/page_layout:
5  empty.xml
6
7  vendor/magento/module-theme/view/frontend/page_layout:
8  1column.xml
9  2columns-left.xml
```

```
10 2columns-right.xml
11 3columns.xml
```

While these page layout files use the same syntax as a regular layout handle XML files, their updates **are not** loaded at the same time as the regular layout handle XML files.

Additionally, consider the following code

```
1 #File: vendor/magento/framework/Pricing/Render.php
2 protected function _prepareLayout()
3 {
4     $this->priceLayout->addHandle($this->
          getPriceRenderHandle());
5     $this->priceLayout->loadLayout();
6     return parent::_prepareLayout();
7 }
```

The price renderer adds a layout handle to a `priceLayout` property. This also triggers a loading/parsing of the layout handle XML files. Making things even more confusing, Magento calls the `_prepareLayout` method when blocks are instantiated and assigned to the layout object. This means the above layout handle parsing happens well after Magento's passed through the Merge/ReaderPool/GeneratorPool process for the normal layout rules.

We're going to explain conceptually how this works, and then walk you through the execution chains that lead up to each of the above use cases.

Multiple Merges

The key to understanding how these multiple merges work is the multiple instance `Magento\Framework\View\Model\Layout\Merge` objects, the multiple instance `Magento\Framework\View\Layout\ScheduledStructure` objects, and the multiple instance structure object (`Magento\Framework\View\Layout\Data\Structure`) we mentioned earlier.

First, unlike most automatic constructor dependency injection objects, the `Magento\Framework\View\Model\Layout\Merge` objects are **not** "single instance/singleton" objects. This is important, as it means calls to the `load` method on individual `Magento\Framework\View\Model\Layout\Merge` objects **will not** share state with each other. This means loading and merging the `page_layout` XML files is completely separate from loading and merging the `layout` files, which is completely separate from the `priceLayout` invocations.

The way Magento goes about doing this is worth calling out. Magento **does not** use the `shared=false` feature of the object manager system. Instead, each individual Merge object is created via a factory object.

Here's where it happens for the main layout object

```
#File: vendor/magento/framework/View/Layout.php

public function getUpdate()
{
    if (!$this->_update) {
        $theme = $this->themeResolver->get();
        $this->_update = $this->_processorFactory->create([
            'theme' => $theme]);
    }
    return $this->_update;
}
```

Here's where it happens for the page layout object

```
#File: vendor/magento/framework/View/Page/Layout/Reader.php
protected function getPageLayoutMerge()
{
    if ($this->pageLayoutMerge) {
        return $this->pageLayoutMerge;
    }
    $this->pageLayoutMerge = $this->processorFactory->
        create([
        'theme'       => $this->themeResolver->get(),
        'fileSource'  => $this->pageLayoutFileSource,
        'cacheSuffix' => self::MERGE_CACHE_SUFFIX,
    ]);
    return $this->pageLayoutMerge;
}
```

Finally, here's where it happens for the price rendering layout object.

```
#File: vendor/magento/framework/Pricing/Render/Layout.php
public function __construct(
    LayoutFactory $layoutFactory,
    \Magento\Framework\View\LayoutInterface $generalLayout
) {
    $this->layout = $layoutFactory->create(['cacheable' =>
        $generalLayout->isCacheable()]);
}
```

We'll talk more about this when we trace out the method calls below.

Another interesting thing about the `Magento\Framework\View\Model\Layout\Merge` objects – each one still loads **every** layout XML file in the system. Or, more accurately, the first object loads every layout XML file in the system. Any `Magento\Framework\View\Model\Layout\Merge` objects instantiated later read that same information from cache. Even though the XML files are separated

out into `page_layout` and `layout` folders, they're all loaded into one giant tree. It's the layout handles that ensure each individual merge object returns only the updates we're after.

We also need to be aware that the `Magento\Framework\View\Layout\ScheduledStructure` object is also an unshared, multiple instance object. This time, however, Magento uses their `di.xml` system to make sure of this.

```
1 #File: vendor/magento/magento2-base/app/etc/di.xml
2 <type name="Magento\Framework\View\Layout\
      ScheduledStructure" shared="false" />
```

This is (again) important because since this is an instance object, it means there's a **different** set of rules and state at play for each run through the reader and generator pools.

This is also true of the structure object itself

```
1 #File: vendor/magento/magento2-base/app/etc/di.xml
2 <type name="Magento\Framework\View\Layout\Data\Structure"
      shared="false" />
```

This may give you pause. The whole point of the `Magento\Framework\View\Layout\Data\Structure` is it keeps track of the entire page's structure. From a certain point of view, it seems illogical that this wouldn't be a single instance/singleton object.

This is why it's important to understand how Magento 2 composes these objects and invokes the Merge/Reader/Generator process.

Invoking the Merges

In the simplest case, a Magento controller's `execute` method returns a

```
1 Magento\Framework\View\Result\Page
```

object, and the system code calls this object's `render` method. While Magento may instantiate multiple `Magento\Framework\View\Result\Page` objects, it only ever **renders** a single one.

```
1 #File: vendor/magento/framework/View/Result/Page.php
2 protected function render(ResponseInterface $response)
3 {
4     $this->pageConfig->publicBuild();
5     /* ... */
6     return $this;
7 }
```

The first line of `render` is important – the `pageConfig` object is a `Magento\Framework\View\Page\Config` object. This is **different** from the `Magento\Framework\View\Page\Config\Structure` object we saw earlier. This config object's `build` method looks like this

```
#File: vendor/magento/framework/View/Page/Config.php
/**
 * TODO Will be eliminated in MAGETWO-28359
 * @return void
 */
public function publicBuild()
{
    $this->build();
}

protected function build()
{
    if (!empty($this->builder)) {
        $this->builder->build();
    }
}
```

The `builder` property, (a `Magento\Framework\View\Page\Builder` object), is our next stop. The build method is defined on this object's parent, `Magento\Framework\View\Layout\Builder` class.

```
#File: vendor/magento/framework/View/Layout/Builder.php
public function build()
{
    if (!$this->isBuilt) {
        $this->isBuilt = true;
        $this->loadLayoutUpdates();
        $this->generateLayoutXml();
        $this->generateLayoutBlocks();
    }
    return $this->layout;
}
```

Each of these methods – `loadLayoutUpdates`, `generateLayoutXml`, and `generateLayoutBlocks` – eventually call methods on a `layout` property.

```
#File: vendor/magento/framework/View/Layout/Builder.php

protected function loadLayoutUpdates()
{
    /* ... */
    $this->layout->getUpdate()->load();
    /* ... */
}
```

```
9
10  protected function generateLayoutXml()
11  {
12      /* ... */
13      $this->layout->generateXml();
14      /* ... */
15  }
16
17  protected function generateLayoutBlocks()
18  {
19      /* ... */
20      $this->layout->generateElements();
21      /* ... */
22  }
```

This layout property comes from the automatic constructor dependency injection system injecting a `Magento\Framework\View\LayoutInterface` interface, which resolves to a `Magento\Framework\View\Layout` object. We'll want to remember this for later.

The next two methods we're interested in are `generateLayoutBlocks` (in the parent class) and `readPageLayout`.

```
1  #File: vendor/magento/framework/View/Page/Builder.php
2  protected function generateLayoutBlocks()
3  {
4      $this->readPageLayout();
5      return parent::generateLayoutBlocks();
6  }
7
8  protected function readPageLayout()
9  {
10     $pageLayout = $this->getPageLayout();
11     if ($pageLayout) {
12         $readerContext = $this->layout->getReaderContext();
13         $this->pageLayoutReader->read($readerContext,
               $pageLayout);
14     }
15 }
```

The `pageLayoutReader` is a `Magento\Framework\View\Page\Layout\Reader` object. Its `read` method looks like this.

```
1  #File: vendor/magento/framework/View/Page/Layout/Reader.php
2  public function read(Layout\Reader\Context $readerContext,
       $pageLayout)
3  {
4      $this->getPageLayoutMerge()->load($pageLayout);
5      $xml = $this->getPageLayoutMerge()->asSimplexml();
```

```
6       $this->reader->interpret($readerContext, $xml);
7   }
```

There's two really important, but subtle, things going on here. The first is the `getPageLayoutMerge` method.

```
1   #File: vendor/magento/framework/View/Page/Layout/Reader.php
2   protected function getPageLayoutMerge()
3   {
4       if ($this->pageLayoutMerge) {
5           return $this->pageLayoutMerge;
6       }
7       $this->pageLayoutMerge = $this->processorFactory->
            create([
8           'theme'       => $this->themeResolver->get(),
9           'fileSource'  => $this->pageLayoutFileSource,
10          'cacheSuffix' => self::MERGE_CACHE_SUFFIX,
11      ]);
12      return $this->pageLayoutMerge;
13  }
```

This method uses a factory to instantiate a **fresh** `Magento\Framework\View\Model\Layout\Merge` object, and calls its load method here

```
1   #File: vendor/magento/framework/View/Page/Layout/Reader.php
2
3   $this->getPageLayoutMerge()->load($pageLayout);
```

This triggers a loading of the layout handle XML files into a Global XML Handle Tree, and then reduces it using the single handle in `$pageLayout` (which will be something like `1column`, `empty`, `2columns-left`, etc.

Then, the `Magento\Framework\View\Page\Layout\Reader` object triggers a run through the reader pools with

```
1   #File: vendor/magento/framework/View/Page/Layout/Reader.php
2
3   $xml = $this->getPageLayoutMerge()->asSimplexml();
4   $this->reader->interpret($readerContext, $xml);
```

This fetches the XML from the `Magento\Framework\View\Model\Layout\Merge` object with the `asSimpleXml` method (which will be XML from the `page_layout` folders **or** any other layout handle XML folder if a user has dropped a `1column.xml`, `empty.xml`, etc. in those locations) Then, Magento starts the reader pool interpret process with

```
1   #File: vendor/magento/framework/View/Page/Layout/Reader.php
2
```

```
3  $this->reader->interpret($readerContext, $xml);
```

The `reader` property is a `Magento\Framework\View\Layout\ReaderPool` created with the `pageLayoutRenderPool` virtual type configuration.

```
1  #File: vendor/magento/magento2-base/app/etc/di.xml
2  <type name="Magento\Framework\View\Page\Layout\Reader">
3      <arguments>
4          <argument name="pageLayoutFileSource" xsi:type="
               object">pageLayoutFileCollectorAggregated</
               argument>
5          <argument name="reader" xsi:type="object">
               pageLayoutRenderPool</argument>
6      </arguments>
7  </type>
```

This gives the reader pool the following reader objects

```
1  #File: vendor/magento/magento2-base/app/etc/di.xml
2  <virtualType name="pageLayoutRenderPool" type="Magento\
       Framework\View\Layout\ReaderPool">
3      <arguments>
4          <argument name="readers" xsi:type="array">
5              <item name="container" xsi:type="string">
                   Magento\Framework\View\Layout\Reader\
                   Container</item>
6              <item name="move" xsi:type="string">Magento\
                   Framework\View\Layout\Reader\Move</item>
7          </argument>
8      </arguments>
9  </virtualType>
```

All this is important because it means that, before Magento runs through its main layout reading process, some reader pools have already run on a separate set of XML files. Also, notice the `$readerContext` object. Magento grabbed this with `$this->layout->getReaderContext();` in the builder object. Remember, the builder's `layout` property is an automatic constructor dependency injection object, injected with the `Magento\Framework\View\LayoutInterface` interface, and resolves to a `Magento\Framework\View\Layout` object.

The reader context object is a `Magento\Framework\View\Layout\Reader\Context` object, and contains the `Magento\Framework\View\Layout\ScheduledStructure` and `Magento\Framework\View\Page\Config\Structure` objects that individual readers will use. Keep this in mind, it will be important later.

Before we move on – some of you may have noticed the reader object appears to configure a different file collector via the `$pageLayoutFileSource` constructor parameter

```
#File: vendor/magento/magento2-base/app/etc/di.xml
<type name="Magento\Framework\View\Page\Layout\Reader">
    <arguments>
        <argument name="pageLayoutFileSource" xsi:type="
            object">pageLayoutFileCollectorAggregated</
            argument>
        <argument name="reader" xsi:type="object">
            pageLayoutRenderPool</argument>
    </arguments>
</type>
```

which is, in turn, used in the merge/update object

```
#File: vendor/magento/framework/View/Page/Layout/Reader.php
protected function getPageLayoutMerge()
{
    if ($this->pageLayoutMerge) {
        return $this->pageLayoutMerge;
    }
    $this->pageLayoutMerge = $this->processorFactory->
        create([
        'theme'       => $this->themeResolver->get(),
        'fileSource'  => $this->pageLayoutFileSource,
        'cacheSuffix' => self::MERGE_CACHE_SUFFIX,
    ]);
    return $this->pageLayoutMerge;
}
```

This would *seem* to run counter to what we're saying about the merge objects loading **every** XML file. However, if we look at the code that loads the XML files from the file system

```
#File: vendor/magento/framework/View/Model/Layout/Merge.php
protected function _loadFileLayoutUpdatesXml()
{
    /* ... */
    $updateFiles = $this->fileSource->getFiles($theme, '*.
        xml');
    $updateFiles = array_merge($updateFiles, $this->
        pageLayoutFileSource->getFiles($theme, '*.xml'));
    //...
}
```

we see that this `pageLayoutFileSource` is **an additional** source of files, and not a replacement. So, the reader instantiating a merge object with a different set of files for the page layout is a red herring.

This is potentially extra important since the results of `_loadFileLayoutUpdatesXml`

are cached

```
#File: vendor/magento/framework/View/Model/Layout/Merge.php
public function getFileLayoutUpdatesXml()
{
    if ($this->layoutUpdatesCache) {
        return $this->layoutUpdatesCache;
    }
    $cacheId = $this->generateCacheId($this->cacheSuffix);
    $result = $this->_loadCache($cacheId);
    if ($result) {
        $result = $this->_loadXmlString($result);
    } else {
        $result = $this->_loadFileLayoutUpdatesXml();
        $this->_saveCache($result->asXml(), $cacheId);
    }
    $this->layoutUpdatesCache = $result;
    return $result;
}
```

This means if, the first time a merge object gets instantiated, the page layout files aren't properly included, the newer `Magento\Framework\View/Model/Layout/Merge` object may be missing them as well.

Loading the Rest of the Updates

Alight, coming back up to `generateLayoutBlocks`

```
#File: vendor/magento/framework/View/Page/Builder.php
protected function generateLayoutBlocks()
{
    $this->readPageLayout();
    return parent::generateLayoutBlocks();
}
```

We now know that, after `readPageLayout` finished, Magento will have run through the page layout handles and scheduled them into the `Magento\Framework\View\Layout\ScheduledStructure` object. Let's jump to the **parent** `generateLayoutBlocks` method.

```
#File: vendor/magento/framework/View/Layout/Builder.php
protected function generateLayoutBlocks()
{
    /* ... */

    /* generate blocks from xml layout */
    $this->layout->generateElements();
```

```
8
9       /* ... */
10
11      return $this;
12  }
```

As previously mentioned, the builder's `layout` property is an injected `Magento\Framework\View\LayoutInterface` interface, and resolves to a `Magento\Framework\View\Layout` object. If we take a look at the `generateElements` method there

```
1   #File: vendor/magento/framework/View/Layout.php
2   public function generateElements()
3   {
4       /* ... */
5
6       if ($result) {
7           $this->readerContext = unserialize($result);
8       } else {
9           \Magento\Framework\Profiler::start('build_structure
                ');
10          $this->readerPool->interpret(
11              $this->getReaderContext(),
12              $this->getNode()
13          );
14          /* ... */
15      }
16
17      $generatorContext = $this->generatorContextFactory->
            create(
18          [
19              'structure' => $this->structure,
20              'layout' => $this,
21          ]
22      );
23
24      /* ... */
25      $this->generatorPool->process($this->getReaderContext()
            , $generatorContext);
26      /* ... */
27  }
```

You'll recall from earlier that this is the main method that kicks off the main reader pools and generator pools. This time through, we're most interested in the call to `getReaderContext`

```
1   #File: vendor/magento/framework/View/Layout.php
2
3   public function getReaderContext()
4   {
```

```
5       if (!$this->readerContext) {
6           $this->readerContext = $this->readerContextFactory
                ->create();
7       }
8       return $this->readerContext;
9   }
```

You'll recall the reader context object holds references to the `Magento\Framework\View\Layout\ScheduledStructure` and `Magento\Framework\View\Page\Config\Structure` objects. By itself, this method makes it look like we're creating a **new** reader context object, which would mean new instance objects for each of the above. **However** – remember our earlier call to `readPageLayout`?

```
1   #File: vendor/magento/framework/View/Page/Builder.php
2   protected function readPageLayout()
3   {
4       $pageLayout = $this->getPageLayout();
5       if ($pageLayout) {
6           $readerContext = $this->layout->getReaderContext();
7           $this->pageLayoutReader->read($readerContext,
                $pageLayout);
8       }
9   }
```

The builder's `layout` property is the same `Magento\Framework\View\Layout` object. In other words, when we called `getReaderContext` in `readPageLayout`, we ensured the `Magento\Framework\View\Layout` object would already have a reader context object instantiated. This means when we call `getReaderContext` from `generateLayoutBlocks`, the context object will contain the same `Magento\Framework\View\Layout\ScheduledStructure` and `Magento\Framework\View\Page\Config\Structure` objects that Magento manipulated during the page layout reader pool run.

Finally, we have the generator context object.

```
1   #File: vendor/magento/framework/View/Layout.php
2
3   $generatorContext = $this->generatorContextFactory->create(
4       [
5           'structure' => $this->structure,
6           'layout' => $this,
7       ]
8   );
9
10  /* ... */
11  $this->generatorPool->process($this->getReaderContext(),
        $generatorContext);
```

This object contains the `Magento\Framework\View\Layout\Data\Structure` object and a reference to the current layout object. These objects are not re-instantiated in the context object returned by the `generatorContextFactory`. This factory's create method

```
#File: vendor/magento/framework/View/Layout/Generator/
    ContextFactory.php
public function create(array $data = [])
{
    return $this->objectManager->create('Magento\Framework\
        View\Layout\Generator\Context', $data);
}
```

simply passes these arguments as constructor arguments via the (little used) second argument to the object manager's `create` method. This ensures the context object receives the already instantiated objects

```
#File: vendor/magento/framework/View/Layout/Generator/
    Context.php
public function __construct(
    Layout\Data\Structure $structure,
    LayoutInterface $layout
) {
    $this->structure = $structure;
    $this->layout = $layout;
}
```

So, what have we learned? We've learned that the page layout handles and the regular layout handles, which Magento merges via different instances of the `Magento\Framework\View\Model\Layout\Merge` object will both operate on the same instances of the `Magento\Framework\View\Page\Config\Structure`, `Magento\Framework\View\Layout\ScheduledStructure` and `Magento\Framework\View\Layout\Data\Structure` objects.

We've also learned that their ability to do so is fragile, and relies on a confusing hierarchy of objects.

Preparing the Layout

This leaves us with this curious bit of code

```
#File: vendor/magento/framework/Pricing/Render.php
protected function _prepareLayout()
{
    $this->priceLayout->addHandle($this->
        getPriceRenderHandle());
    $this->priceLayout->loadLayout();
```

```
6       return parent::_prepareLayout();
7   }
```

In Magento 1, the `_prepareLayout` method was available for block developers to make additional adjustments to the `Mage::getSingleton(core/layout)` object and/or its own children. This is still the purpose of `_prepareLayout` in Magento 2. Magento calls a block's `_prepareLayout` method when some other code assigns a layout to a block

```
1   #File: vendor/magento/framework/View/Element/AbstractBlock.
        php
2   public function setLayout(\Magento\Framework\View\
        LayoutInterface $layout)
3   {
4       $this->_layout = $layout;
5       $this->_prepareLayout();
6       return $this;
7   }
```

For blocks, "some other code" is either the `createBlock` method in the layout object

```
1   #File: vendor/magento/framework/View/Layout.php
2   public function createBlock($type, $name = '', array
        $arguments = [])
3   {
4       $this->build();
5       $name = $this->structure->createStructuralElement($name
            , Element::TYPE_BLOCK, $type);
6       $block = $this->_createBlock($type, $name, $arguments);
7       $block->setLayout($this);
8       return $block;
9   }
```

or (of more interest to us) this loop in the block generator.

```
1   #File: vendor/magento/framework/View/Layout/Generator/Block
        .php
2   // Set layout instance to all generated block (trigger
        _prepareLayout method)
3   foreach ($blocks as $elementName => $block) {
4       try {
5           $block->setLayout($layout);
6           $this->eventManager->dispatch('
                core_layout_block_create_after', ['block' =>
                $block]);
7       } catch (\Exception $e) {
8           $this->handleRenderException($e);
9           $layout->setBlock(
```

```
10              $elementName,
11              $this->exceptionHandlerBlockFactory->create(['
                    blockName' => $elementName])
12          );
13          unset($blockActions[$elementName]);
14      }
15      $scheduledStructure->unsetElement($elementName);
16  }
```

This makes the price render block all the more curious

```
1  #File: vendor/magento/framework/Pricing/Render.php
2  protected function _prepareLayout()
3  {
4      $this->priceLayout->addHandle($this->
           getPriceRenderHandle());
5      $this->priceLayout->loadLayout();
6      return parent::_prepareLayout();
7  }
```

This method appears to add a layout handle to a layout object, and then loads that layout. Semantically this seems to be saying "hey, load this extra layout handle and add the blocks. But how can that happen when Magento's already at the generator stage? The handles have already done their thing by now.

To understand this, we need to look at the `priceLayout` object

```
1  #File: vendor/magento/framework/Pricing/Render.php
2  use Magento\Framework\Pricing\Render\Layout;
3  /* ... */
4  public function __construct(
5      Template\Context $context,
6      Layout $priceLayout,
7      array $data = []
8  ) {
9      $this->priceLayout = $priceLayout;
10     parent::__construct($context, $data);
11 }
```

This is a new object we haven't seen before – injected with the class `Magento\Framework\Pricing\Render\Layout`. If we look at that class's source

```
1  #File: vendor/magento/framework/Pricing/Render/Layout.php
2  namespace Magento\Framework\Pricing\Render;
3
4  use Magento\Framework\View\LayoutFactory;
5  use Magento\Framework\View\LayoutInterface;
6
7  /* ... */
```

```
8  class Layout
9  {
10     /* ... */
11     public function __construct(
12         LayoutFactory $layoutFactory,
13         \Magento\Framework\View\LayoutInterface
               $generalLayout
14     ) {
15         $this->layout = $layoutFactory->create(['cacheable'
               => $generalLayout->isCacheable()]);
16     }
17     /* ... */
18 }
```

We see this is a stand alone class, with **no** parent. We also can see this `Magento\Framework\Pricing\Render\Layout` object has a `layout` property thats built with a `Magento\Framework\View\LayoutFactory`. This will be a **new** instance of the `Magento\Framework\View\Layout` object. If we look at the `addHandle` and `loadLayout` methods defined on the `Magento\Framework\Pricing\Render\Layout` object

```
1  #File: vendor/magento/framework/Pricing/Render/Layout.php
2
3  public function addHandle($handle)
4  {
5      $this->layout->getUpdate()->addHandle($handle);
6  }
7
8  /**
9   * Load layout
10  *
11  * @return void
12  */
13 public function loadLayout()
14 {
15     $this->layout->getUpdate()->load();
16     $this->layout->generateXml();
17     $this->layout->generateElements();
18 }
```

we see they're calling through to the new layout object.

This presents a problem. If this is a new instance of the `Magento\Framework\View\Layout` object, that means a new instance of the `Magento\Framework\View\Model\Layout\Merge` object, as well as new instances of the `Magento\Framework\View\Layout\ScheduledStructure`, `Magento\Framework\View\Page\Config\Structure`, and `Magento\Framework\View\Layout\Data\Structure` objects. In other words, this whole things *seems to* trigger the rendering of a second layout tree, but one that Magento will never have a chance to actually render.

This one had me stumped until I looked at how the system uses the `Magento\Framework\Pricing\Render` block.

```
#File: vendor/magento/module-catalog/Block/Product/ListProduct.php

$priceRender = $this->getPriceRender();

$price = '';
if ($priceRender) {
    $price = $priceRender->render(
        \Magento\Catalog\Pricing\Price\FinalPrice::PRICE_CODE,
        $product,
        [
            'include_container' => true,
            'display_minimal_price' => true,
            'zone' => \Magento\Framework\Pricing\Render::ZONE_ITEM_LIST,
            'list_category_page' => true
        ]
    );
}
```

The `$priceRender` variable above fetches the `Magento\Framework\Pricing\Render` block from the layout, and then calls its render method. In that `render` method

```
#File: vendor/magento/framework/Pricing/Render.php
public function render($priceCode, SaleableInterface $saleableItem, array $arguments = [])
{
    $useArguments = array_replace($this->_data, $arguments);

    /** @var \Magento\Framework\Pricing\Render\RendererPool $rendererPool */
    $rendererPool = $this->priceLayout->getBlock('render.product.prices');
    if (!$rendererPool) {
        throw new \RuntimeException('Wrong Price Rendering layout configuration. Factory block is missed');
    }

    // obtain concrete Price Render
    $priceRender = $rendererPool->createPriceRender($priceCode, $saleableItem, $useArguments);
    return $priceRender->toHtml();
}
```

Magento *reaches into* this separate, stand-alone `priceLayout` object and pulls out blocks to render.

```
$rendererPool = $this->priceLayout->getBlock('render.
    product.prices');
```

This price layout **is** a layout object loaded completely separately from the rest of the layout system. This means its XML files

```
$ find vendor/magento/ -name catalog_product_prices.xml
vendor/magento//module-bundle/view/base/layout/
    catalog_product_prices.xml
vendor/magento//module-catalog/view/base/layout/
    catalog_product_prices.xml
vendor/magento//module-configurable-product/view/base/
    layout/catalog_product_prices.xml
vendor/magento//module-grouped-product/view/base/layout/
    catalog_product_prices.xml
vendor/magento//module-msrp/view/base/layout/
    catalog_product_prices.xml
vendor/magento//module-tax/view/base/layout/
    catalog_product_prices.xml
vendor/magento//module-weee/view/base/layout/
    catalog_product_prices.xml
vendor/magento//module-wishlist/view/base/layout/
    catalog_product_prices.xml
```

do not have access to the other Magento layout blocks. It also means the other layout blocks won't have access to this price renderer information.

While an interesting example of building an isolated layout, this has caused a lot of confusion for early adopters, and will continue to cause confusion as developers transition to Magento 2.

Wrap Up

As you can see, what was once a simple two class (plus all the blocks) affair in Magento 1 has become something much more complex in Magento 2. The most confusing bits here are

- Two different "layout" objects, each with different responsibilities
- The update/merge object's reliance on internal state leading to the need for instance objects
- The heavy use of factories to *sometimes* create multiple-instance objects

This all leads to a situation where the system *may* behave differently, depending on how its invoked. Adding to the confusion? The above examples assume you're

using the new page layout objects. If you try (as some Magento core objects do) to load an interact with the layout with these page layout objects

```
#File: vendor/magento/module-cms/Helper/Page.php
if ($this->_page->getCustomLayoutUpdateXml() && $inRange) {
    $layoutUpdate = $this->_page->getCustomLayoutUpdateXml
        ();
} else {
    $layoutUpdate = $this->_page->getLayoutUpdateXml();
}
if (!empty($layoutUpdate)) {
    $resultPage->getLayout()->getUpdate()->addUpdate(
        $layoutUpdate);
}

$contentHeadingBlock = $resultPage->getLayout()->getBlock('
    page_content_heading');
```

you run the risk of invoking these objects in an unintended way and creating behavior that's closer to a side effect than the intended system behavior.

Unlike Magento 1, where working directly with the `Mage::getSingleton(core/layout);` object was a practical shortcut that could simplify your project, you're better off steering clear of Magento's layout related PHP objects and sticking the XML files. Even if you are capable of following all the way along in this chapter, all this layout code leaves me with the distinct impression of a house that's has its sheetrock put up, but no fixtures installed.

Chapter 8

Front End Starter Kit – CSS

We've just spent 6 chapters winding our way through the complex world of generating HTML using Magento's domain specific layout XML language. While HTML generation is the layout system's primary purpose, HTML is only one third of a browser based page-or-application's appearance and behavior. The remainder of this book will be a crash course on Magento's systems for working with Cascading Style Sheets (CSS) and the javascript programming language.

Magento, CSS and LessCSS

Magento, like all other browser based applications, uses CSS files to provide style and layout rules for its various pages and systems. You'll see this if you view the source of any Magento page

```
1 <link  rel="stylesheet" type="text/css"  media="all" href="
     http://magento.example.com/static/version1484770320/
     frontend/Magento/luma/en_US/css/styles-m.css" />
2 <link  rel="stylesheet" type="text/css"  media="screen and
     (min-width: 768px)" href="http://magento.example.com/
     static/version1484770320/frontend/Magento/luma/en_US/
     css/styles-l.css" />
3 <link  rel="stylesheet" type="text/css"  media="print" href
     ="http://magento.example.com/static/version1484770320/
     frontend/Magento/luma/en_US/css/print.css" />
```

However, where Magento 2 diverges from Magento 1 is in **how** these files are created, and how the URLs for these assets are generated.

Magento 2, (like Symfony and other PHP frameworks), has a "static content generator" that gathers up all the CSS, Javascript, and "other" front end files bundled with the code modules. Once gathered, the system creates a final set of front end files in the `pub/static` folder.

Magento 2, **unlike** Symfony and other PHP frameworks, ships with two possible default web root/index.php files –

```
index.php
pub/index.php
pub/static
```

This means that Magento 2 generates static asset URLs differently depending on which webroot you're serving Magento from.

```
http://magento.example.com/pub/static/...

vs.

http://magento.example.com/static/...
```

In Magento 1 it was possible to ship your own top level CSS folder and generate your own CSS `<link/>` tags. This is not practical in Magento 2. With Magento 2, you really need to stick to Magento's front end abstractions for adding and linking CSS/javascript.

Adding a CSS File

Adding a CSS file to Magento is as simple as adding a layout handle XML file. Let's add a default handle

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    layout/default.xml
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="urn:magento:framework:
    View/Layout/etc/page_configuration.xsd">
    <head>
        <css src="Pulsestorm_Nofrillslayout::chapter-7/
            example.css"/>
    </head>
</page>
```

Per previous chapters, we know Magento will run the code in this `default.xml` file on every page for the `frontend` area. Be sure you use the `page_configuration.xsd` schema for your file – it's the schema that allows you to use the `<head/>` tag.

As for the XML itself

```
<head>
    <css src="Pulsestorm_Nofrillslayout::chapter-7/example.
        css"/>
</head>
```

The <head/> section tells Magento the sub-nodes are Magento's special, head modifying instructions. In other words, these nodes don't relate directly to building block objects. The <css/> node is the layout handle XML command that tells Magento we want to add an HTML <link/> tag to the document's head.

If you clear your cache and reload the page with the above in place, you should see the following HTML (or something very similar to it) in your source

```
<link  rel="stylesheet" type="text/css"  media="all" href="
    http://magento-2-2-x.dev/static/version1514092162/
    frontend/Pulsestorm/dram/en_US/
    Pulsestorm_Nofrillslayout/chapter-7/example.css" />
```

Magento has taken our src="Pulsestorm_Nofrillslayout::chapter-7/example.css" URN, and converted it into a full asset URL. The above URL is from a system that's using the /pub/index.php folder/filter as its webroot. If we were using the root folder as the webroot, we'd see the following instead.

```
<link  rel="stylesheet" type="text/css"  media="all" href="
    http://magento-2-2-x.dev/pub/static/version1514092162/
    frontend/Pulsestorm/dram/en_US/
    Pulsestorm_Nofrillslayout/chapter-7/example.css" />
```

Regardless of which URL your system loads, if we look at the CSS URL directly in our browser (or, via curl – see the appendix if you're not familiar with this command), we'll get a 404 error

```
$ curl -I 'http://magento-2-2-x.dev/pub/static/
    version1514092162/frontend/Pulsestorm/dram/en_US/
    Pulsestorm_Nofrillslayout/chapter-7/example.css'
HTTP/1.1 404 Not Found
Date: Tue, 13 Feb 2018 02:56:24 GMT
Server: Apache/2.4.28 (Unix) PHP/7.0.18
X-Powered-By: PHP/7.0.18
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
X-UA-Compatible: IE=edge
Content-Type: text/plain;charset=UTF-8
```

That's because we need to add the actual CSS file. Let's do that now! The `Pulsestorm_Nofrillslayout::chapter-7/example.css` identifier works similarly to the `phtml` template identifiers we've already seen. However, when used in the `<css/>` tag, the identifier tells Magento to look in the following folder for CSS files.

```
app/code/[Namespace]/[Module]/view/[area]/web
```

This means we'll want to create our CSS file here

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/web
    /chapter-7/example.css
body {
    background-color:#f00;
}
```

With the above in place, reload/re-download your CSS URL and you should see our file. Reload the main page, and these CSS rules will be included, just like any other.

Magento and LessCSS

Adding a traditional CSS file to a page is just the beginning of Magento 2's CSS improvements. The next thing we'll want to talk about is LessCSS. LessCSS is one of the first CSS preprocessors to gain traction in the front end web development world. The default themes in Magento's CSS build systems are deeply ingrained with LessCSS workflows and concepts.

While there's lots of efforts out there to create alternatives using the Sass preprocessor, or even "headless" systems using the REST API – the fact that Magento's base themes ship using LessCSS styles that target specific DOM nodes/`class=`/`id=`/etc. means a good chunk of the Magento theme and extension ecosystem will follow along with LessCSS rules. i.e. Even if you like Sass or headless workflows, chances are you'll need to understand LessCSS if you want to get your Magento job done.

If you've already made your way through our theming chapter you know how to add a LessCSS stylesheet, but it never hurts to repeat things. The syntax for adding a LessCSS stylesheet to the system is surprisingly similar to adding a CSS file. Open up our `default.xml` file and add another `<css/>` node.

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    layout/default.xml
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="urn:magento:framework:
    View/Layout/etc/page_configuration.xsd">
```

```
3       <head>
4           <css src="Pulsestorm_Nofrillslayout::chapter-7/
                example.css"/>
5
6           <!-- START: our new node -->
7           <css src="Pulsestorm_Nofrillslayout::chapter-7/
                example-2.css"/>
8           <!-- END:   our new node -->
9       </head>
10  </page>
```

Adding a LessCSS file uses the exact same syntax as adding a CSS file. All you need to do is add a new `<css/>` node to your layout handle XML file, and add a new `src` URN. One weird, but important thing? You'll want your source URL to have a `.css` extension, **even though** we're adding a `.less` file.

With the above layout handle XML in place, clear your cache and reload the page. You should see the following `<link/>` tag added to the page.

```
1   <link  rel="stylesheet" type="text/css"  media="all" href="
        http://magento.example.com/static/frontend/Magento/luma
        /en_US/Pulsestorm_Nofrillslayout/chapter-7/example-2.
        css" />
```

Just like before, if we load our CSS URL directly in a web browser, we'll get a 404 error. Let's add our file.

```
1   #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/web
        /chapter-7/example-2.less
2   @myBlue: #00f;
3   body {
4       background-color:@myBlue;
5   }
```

Now, load the `example-2.css` URL, and you should see the following

```
1   body {
2     background-color: #0000ff;
3   }
```

Also, all the pages in your Magento front end should now have a blue background.

At this high level, Magento's LessCSS implementation is relatively simple – whenever you add a CSS file to Magento via a layout handle XML file

```
1   <css src="Pulsestorm_Nofrillslayout::chapter-7/example-2.
        css"/>
```

Magento will first look for a `.less` file with the same name. If Magento finds a `.less` file, the LessCSS code embedded into Magento's system will transform the LessCSS file immediately to a CSS file. That's what happened above – when we loaded the CSS URL, Magento processed our LessCSS file and transformed it into CSS for us.

Magento LessCSS Performance

Once you have the basic mechanics of Magento's CSS down, the first question that probably comes to mind is

> Isn't it CPU/perf expensive to do this much LessCSS preprocessing on the fly?

The answer here is both yes **and** no. It's true that, when creating or editing your LessCSS files, Magento will need to recreate the CSS on every request. **However**, when Magento's running in production mode, **no** automatic LessCSS generation happens. Instead, part of deploying a Magento system involves running the `bin/magento setup:static-content:deploy` command. (If you're not familiar with running Magento commands via the command line, checkout the CLI appendix).

```
1 $ php bin/magento setup:static-content:deploy
```

One of the things this command does is scan through every layout handle XML file in the system, look for `<css/>` directives with associated LessCSS files, and automatically generate the needed CSS. The `setup:static-content:deploy` command will generate these files once. Alternately, when Magento's in development mode it will render them on the fly.

Magento LessCSS Caching

When you're working with LessCSS files in development mode and want to see changes, there's a number of things you'll need to do to make sure your changes show up.

First, obviously, you'll want to edit your file. Below we're editing our `example-2.less` file to use a less garish blue.

```
1 #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/web
      /chapter-7/example-2.less
2 @myBlue: #00a;
3 body {
4     background-color:@myBlue;
5 }
```

Next, you'll need to remove a cached version of the `.less` file. This file doesn't live in Magento's normal cache – instead there's a `var/view_preprocessed` folder

```
1 var/view_preprocessed
```

that contains **all** the preprocessed view files. You can remove the entire `view_preprocessed` folder **or** find your specific `.less` file and remove it. (if you're not familiar with the unix `find` command we're using below, please see the appendix for a quick tutorial)

```
1 $ find var/view_preprocessed -name example-2.less
2 var/view_preprocessed/css/frontend/Magento/luma/en_US/
      Pulsestorm_Nofrillslayout/chapter-7/example-2.less
3
4 $ find var/view_preprocessed -name example-2.less -exec rm
      '{}' \;
5 $ rm var/view_preprocessed/css/frontend/Magento/luma/en_US/
      Pulsestorm_Nofrillslayout/chapter-7/example-2.less
```

Once you've removed the `.less` file, you'll **also** need to remove the generated `.css` file from `pub/static`.

```
1 $ find pub/static -name example-2.css
2 pub/static/frontend/Magento/luma/en_US/
      Pulsestorm_Nofrillslayout/chapter-7/example-2.css
3
4 $ find pub/static -name example-2.css -
5 $ rm pub/static/frontend/Magento/luma/en_US/
      Pulsestorm_Nofrillslayout/chapter-7/example-2.css
```

Once both these files are deleted, you should be able to reload the CSS file, which will regenerate your LessCSS file.

The Magento CLI tool **does not** ship with commands to automatically clear out the LessCSS cache. Instead, you'll need to use the `grunt` task runner to manage Magento's LessCSS pipeline. We have a quick tutorial on this tool in the appendix. While the `grunt` task runner has commands that let you do the above automatically, we've found it's a good idea to know where these cache files live in case you need to run down sticky style rules yourself.

Magento's LessCSS Style Sheets

So, we now know how to add our own LessCSS style sheets to the system, edit them efficiently, and have accepted the production-mode/development-mode differences as the territory we're dealing with. The last thing we'll need to talk about is how Magento's two default themes use LessCSS.

When Magento 2.0 shipped, there were only **six** top level `.less` files that Magento included via `<css/>` tags.

```
1  css/print.css
2  vendor/magento//theme-frontend-blank/web/css/print.less
3
4  css/styles-l.css
5  vendor/magento//theme-frontend-blank/web/css/styles-l.less
6
7  css/styles-m.css
8  vendor/magento//theme-frontend-blank/web/css/styles-m.less
9
10 css/styles-old.css
11 vendor/magento//theme-adminhtml-backend/web/css/styles-old.
       less
12
13 css/styles.css
14 vendor/magento//theme-adminhtml-backend/web/css/styles.less
15
16 mage/gallery/gallery.css
17 vendor/magento//magento2-base/lib/web/mage/gallery/gallery.
       less
```

However, this may be a little confusing if you've searched Magento's code base for files with a `.less` extension. Despite there being only six distinct `<css/>` nodes, there's almost 500 `.less` files.

```
1  $ find vendor/magento/ -name '*.less' | wc -l
2   485
3
4  $ find vendor/magento/ -name '*.less'
5  vendor/magento//framework/Css/Test/Unit/PreProcessor/_files
       /invalid.less
6  vendor/magento//framework/Css/Test/Unit/PreProcessor/_files
       /valid.less
7  ...
8  vendor/magento//theme-frontend-luma/web/css/source/_theme.
       less
9  vendor/magento//theme-frontend-luma/web/css/source/
       _variables.less
10 vendor/magento//theme-frontend-luma/web/css/source/
       components/_modals_extend.less
```

What gives?

Well, if we take a look at `styles-l.less`, the main `.less` file for desktop styles.

```
1  #File: vendor/magento/theme-frontend-blank/web/css/styles-l
       .less
2
```

```
3  //...
4
5  @import '_styles.less';
6  @import (reference) 'source/_extends.less';
7
8  //
9  //  Magento Import instructions
10 //  ---------------------------------------------
11
12 //@magento_import 'source/_module.less'; // Theme modules
13 //@magento_import 'source/_widgets.less'; // Theme widgets
14
15 //...
```

Unlike our simple background color setting example, the LessCSS styles for a full Magento site are significantly more complicated. Rather than jam all the style rules in a single, megalithic file, Magento takes advantage of LessCSS's `@import` command, as well as the special Magento `//@magento_import` command to separate styles into different files.

The LessCSS `@import` directive is relatively simple to understand. It's a stock LessCSS command, and simply loads in all the style rules from a separate file.

The `//@magento_import` directive is a little less straight forward. Your first thought may be to ignore this line – after all, it's prepended with a comment, isn't it? LessCSS should ignore it, right?

Under normal circumstances, you'd be correct. However, Magento's custom LessCSS preprocessor code will search any Magento `.less` file for `//@magento_import` (with the comments), and replace it with the contents of a number of different files. Specifically, Magento will search *all modules and themes* for the provided `.less` files, and if found, add them to the final generated CSS. For example, in the code above, when Magento sees

```
1  //@magento_import 'source/_module.less';
```

It will search **every** module for file that matches

```
1  path/to/component/[Namespace]/[Module]/view/[area]/web/css/
       source/_module.less
```

Additionally, since these are considered view files by Magento, the same rules that apply to merging layout handle XML in themes also applies to LessCSS files pulled in via `magento_import`. i.e. **all** the following `_module.less` files are merged into the final generated CSS files when you're using the Luma theme.

```
1  $ find vendor/magento/ -wholename '*theme-frontend-luma*web
       /css/source/_module.less*'
```

```
2 vendor/magento//theme-frontend-luma/
      Magento_AdvancedCheckout/web/css/source/_module.less
3 vendor/magento//theme-frontend-luma/Magento_AdvancedSearch/
      web/css/source/_module.less
4 //...
5 vendor/magento//theme-frontend-luma/Magento_Vault/web/css/
      source/_module.less
6 vendor/magento//theme-frontend-luma/Magento_Wishlist/web/
      css/source/_module.less
```

You can see a working example of `magento_import` in the `Pulsestorm_Nofrills` module. We've included the following LessCSS files with the module

```
1 #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/web
      /css/source/_module.less
2 #someWeirdRuleInModule{
3     color:#fff;
4 }
5
6 #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/web
      /css/source/widgets.less
7 #someWeirdRuleInWidget{
8     color:#fff;
9 }
```

Because these paths match the above mentioned rules, you should find the `#someWeird...` rules in your generated `styles-l.css` file.

PHP Code for Magento Imports

If you're curious how the `magento_import` preprocessor loads its files, you'll find the code in the following class/file

```
1 vendor/magento/framework/Css/PreProcessor/Instruction/
      MagentoImport.php
```

If you're curious how Magento's LessCSS processor itself works, you'll find the code that kicks off preprocessing here

```
1 vendor/magento/framework/Css/PreProcessor/Adapter/Less/
      Processor.php
```

The Problem With Sassy Alternatives

For folks working for a certain sort of interactive agency, Magento 2's choice of LessCSS was somewhat disappointing. Other preprocessors like Sass and Stylus seem more popular with this set of developers. Because of this, we've seen some efforts to create alternate CSS workflows using these systems.

While these are interesting projects, and do create a workable alternative for folks heavily invested in these stacks, these projects (along with things like "headless" API only implementations) end up sacrificing a huge chunk of Magento's value as a software ecosystem.

There's no such thing as a preprocessor standard – each of these toolsets is capable of different things. So far, each alternative preprocessor project ends up outputting CSS that's different from the LessCSS generated CSS. They also often need to (or want to) implement changes to the layout handle files or generated HTML.

While all of this is OK on an individual system, the tradeoff is the wealth of third party code (open source and commercial) that relies on the existing LessCSS rules, the existing layout structure, or the existing HTML structure may end up not working. It's hard to overstate the value that consistency in the available layout blocks, style rules, and HTML code brings for folks making redistributable Magento code.

Even if these alternate preprocessors somehow manage to maintain compatibility with the LessCSS versions of Magento's themes – extension developers are placed in an odd position: Do they adopt LessCSS rules and provide Sass/Stylus alternatives? Do they do this for each individual Sass or Stylus project?

Unfortunately, there aren't easy answers here. While it's possible to use pure CSS (and therefore your own CSS workflows) by adding new files to Magento, most working Magento developers won't be able to avoid working with LessCSS, and it seems unlikely Magento 2 will rip out the guts of a system that's already taken countless developer hours to build.

Chapter 9

Front End Starter Kit – Javascript

It's hard to escape the fact that the modern web runs on javascript. This puts Magento in a tricky position. As a traditional PHP MVC framework, it's hard for Magento to take advantage of everything the modern javascript world has to offer. There's also parts of Magento 2 that are still running on Magento 1 code, and still require the grandparent of all javascript frameworks – PrototypeJS!

In this chapter we'll explain the basics of what you'll need to know to get started with Javascript development in Magento 2.

RequireJS

Javascript files face a set of challenges similar to those faced by CSS files. Magento considers javascript files static content, so source files and deployed files are two different things. The `/pub/static` vs. `/static` document root problem also exists. Fortunately, just like Magento 2 provides a `<css/>` tag for adding CSS files, it also provides a `<script/>` tag for adding javascript files. Let's add the following node to our `default.xml` file we created in Chapter 7

```
1  #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
       layout/default.xml
2
3  <page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="urn:magento:framework:
       View/Layout/etc/page_configuration.xsd">
4      <head>
5          <!-- ... -->
```

```
6           <script src="Pulsestorm_Nofrillslayout::chapter-8/
                example-script.js"/>
7
8           <!-- alternate <link/> syntax for the same thing
                -->
9           <!-- <link src="Pulsestorm_Nofrillslayout::chapter
                -8/example-script.js"/>              -->
10          <!-- ... -->
11      </head>
12  </page>
```

Here we've added a `<script/>` tag to the `<head/>` section of our layout handle XML file, with a `src` URN of `Pulsestorm_Nofrillslayout::chapter-8/example-script.js`. If we clear our cache and reload with the above in place, you should see a `<script></script>` tag added to the source of all the pages in your Magento system.

```
1  <script  type="text/javascript"  src="http://magento.
       example.com/static/version1514092162/frontend/
       Pulsestorm/dram/en_US/Pulsestorm_Nofrillslayout/chapter
       -8/example-script.js"></script>
```

Similar to our `<css/>` layout commands, Magento's used the URN in our `<script/>` tag to automatically generate a full URL. Also similar to our `<css/>` tag, we'll need to actually create a `chapter-8/example-script.js` file.

```
1  #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/web
       /chapter-8/example-script.js
2  alert("Hello World");
```

Reload the page with the above in place, and you should see a "Hello World" alert box. So we're all set right? Not quite. Let's give the following script a try.

```
1  #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/web
       /chapter-8/example-script.js
2
3  jQuery(function(){
4      alert("Hello World");
5  });
```

The above javascript program uses jQuery's document ready functionality to hold off on calling out alert until the document DOM has loaded. However, if you try loading the page with the above in place, you probably **won't** see an alert. Instead you'll see the following in your javascript error console.

> Uncaught ReferenceError: jQuery is not defined

To the uninitiated, this may look like Magento reneged on its promise to include jQuery with Magento 2. However, if you paste the above program into your

javascript console, javascript will find the global jQuery object without issue.

How is it possible that an inline bit of javascript in a page can't access jQuery, but a fully loaded page can?

While it's possible to use plain "raw" javascript in your Magento 2 projects, Magento's default front end systems are **heavily** biased towards an open-source, javascript AMD system named RequireJS. If you're hewing closely to Magento's idea of how javascript should be written, you'll need to become intimately familiar with this system.

Understanding RequireJS Programs

You run a RequireJS program by passing a list of *module dependencies* to the `require` function (or its alias, `requirejs`), along with a single javascript function that defines your program.

```
require([...list of dependencies...], function(){
    //your program code
});
```

So, the simplest hello world program might look something like this

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/web
    /chapter-8/example-script.js
require([], function(){
    alert("Hello RequireJS");
});
```

Reload the page with the above code in place, and you should see your "Hello RequireJS" alert displayed. The above program has **zero** module dependencies – i.e. an empty array (`[]`).

A RequireJS module is a javascript object, function, or string, returned by a unique namespace path. One of the goals of RequireJS is to abstract away the loading of individual javascript files, and let the javascript programmer think about their code in terms of functionality. For example, instead of referencing a global `jQuery` object, RequireJS provides a `jquery` module. Consider the following program

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/web
    /chapter-8/example-script.js

require(['jquery'], function(jqueryModule){
    //the jquery module returns the object we normally
    //think about as the global jQuery/$ obejct
    var body = jqueryModule('body')[0];
```

```
7       jqueryModule(body).html('<p>Hello jQuery</p>')
8   });
```

If you reload your page with the above in place, you'll find that the contents of the <body/> tag have been replaced with the Hello jQuery HTML. The key to understanding this program is the following

```
1   #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/web
        /chapter-8/example-script.js
2   require(['jquery'], function(jqueryModule){
3       //...
4   });
```

Here, we've told RequireJS that our program needs to use the jQuery module ([jquery). Because we listed the module named jquery as the first item in in our dependency list, RequireJS passed the jQuery module object to our program function as the first parameter (jqueryModule above). RequireJS will pass any module specified in the module list as parameters to your program.

For example, in an imaginary system that had modules named one, two, and three, you'd be able to include them in a program like this.

```
1   require(['one','two','three'], function(oneModule,
        twoModule, threeModule){
2   });
```

Creating your Own RequireJS Modules

Magento 2's javascript systems are strongly biased towards third party developers creating new RequireJS modules for their code. Unless you already know what you're doing, you will not want to fight this.

Magento provides a RequireJS configuration that allows each Magento module to have a RequireJS namespace. For example, give the following program a try.

```
1   #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/web
        /chapter-8/example-script.js
2
3   require(
4       ['jquery', 'Pulsestorm_Nofrillslayout/messages'],
5       function(jQuery, messages){
6           var body = jQuery('body')[0];
7           jQuery(body).html(messages.getMessage())
8       }
9   );
```

The program above has two module dependencies – `jquery` and `Pulsestorm_Nofrillslayout/messages`. The former is a Magento core provided module for accessing jQuery – the second is a module we've provided as part of the `Pulsestorm_Nofrillslayout` Magento module.

Magento's RequireJS bootstrap allows us to use a RequireJS name like this

```
[Magento Module Name]/some-string
Pulsestorm_Nofrillslaoyut/messages
```

and have that RequireJS module correspond to a **Magento module** (`Pulsestorm_Nofrillslaoyut` above). You can see the RequireJS module definition here

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/web
    /messages.js
define([], function(){
    var moduleObject = {};
    moduleObject.getMessage = function(){
        return "Hello RequireJS Custom Module";
    };
    return moduleObject;
});
```

Put another way, Magento knows to look for this RequireJS module in `path/to/Pulsestorm/Nofrillslayout` because the first portion of the module name is `Pulsestorm_Nofrillslayout`.

Magento takes the second portion of the module name (`messages` above), and turns that into a `view/[area]/web/...` file path (`view/frontend/web/messages.js` above). If you're not familiar with the specifics of defining a RequireJS module, the official RequireJS documentation is a good place to start.

http://requirejs.org/docs/api.html#define

RequireJS Bootstrap

Covering RequireJS in full is beyond the scope of this book. However, what **is** in the scope of this book is covering how Magento bootstraps its RequireJS instance. Once you understand this, you should be able to track down anything you need by using the official RequireJS docs. http://requirejs.org/docs.

There are **three** `<script/>` tags you'll want to be aware of w/r/t the RequireJS bootstrap

```
<script  type="text/javascript"  src="http://magento.
    example.com/static/version1514092162/frontend/Magento/
    luma/en_US/requirejs/require.js"></script>
```

```
2 <script type="text/javascript" src="http://magento.
    example.com/static/version1514092162/frontend/Magento/
    luma/en_US/mage/requirejs/mixins.js"></script>
3 <script type="text/javascript" src="http://magento.
    example.com/static/version1514092162/_requirejs/
    frontend/Magento/luma/en_US/requirejs-config.js"></
    script>
```

The `requirejs/require.js` file is the main RequireJS source file. This implements the functions that make RequireJS work.

The `mage/requirejs/mixins.js` file is a special Magento file. This file **redefines** several RequireJS source functions. Magento does this to implement a "mixins" system. This system allows RequireJS developers to intercept the creation of any RequireJS module and redefine it – sort of like Magento 1 class rewrites, but for RequireJS javascript code. The specifics of this system are beyond the scope of this book, but you can read more online.

https://alanstorm.com/the-curious-case-of-magento-2-mixins/

The final file, `requirejs-config.js`, is the most important one to understand. If you load this in your web browser directly, you'll see a number of calls to `require.config`

```
1  (function(require){
2  (function() {
3  /**
4   * Copyright © 2016 Magento. All rights reserved.
5   * See COPYING.txt for license details.
6   */
7
8  var config = {
9      /* ... */
10 };
11
12 require.config(config);
13 })();
```

This file is **not** a static javascript file. It's dynamically generated from the contents of each module's `requirejs-config.js` file(s).

```
1 $ find vendor/magento/ -name requirejs-config.js
2 vendor/magento//module-admin-notification/view/adminhtml/
    requirejs-config.js
3 vendor/magento//module-authorizenet/view/frontend/requirejs
    -config.js
4 /*...*/
5 vendor/magento//module-weee/view/frontend/requirejs-config.
    js
```

```
6 vendor/magento//module-wishlist/view/frontend/requirejs-
      config.js
```

These files are Magento's way of letting each individual module author call RequireJS's built-in configuration method

http://requirejs.org/docs/api.html#config

in order to specify the behavior of RequireJS. Magento's core code uses these files for a variety of reasons – including setting up aliases for modules, configuring dependencies for legacy modules, setting jQuery's no conflict mode, and more.

The specifics of how all this is implemented are beyond the scope of this book, but if you're curious, this Magento 2 and RequireJS article is a good place to start.

http://alanstorm.com/magento_2_and_requirejs/

Module Dependencies

One important bit of Magento's Require.js configuration we **will** cover is here.

```
1  #File: vendor/magento/module-theme/view/frontend/requirejs-
       config.js
2  var config = {
3      deps: [
4          "jquery/jquery.mobile.custom",
5          "js/responsive",
6          "mage/common",
7          "mage/dataPost",
8          "js/theme",
9          "mage/bootstrap"
10     ]
11 }
12
13 #File: vendor/magento/module-theme/view/adminhtml/requirejs
       -config.js
14 var config = {
15     "deps": [
16         "js/theme",
17         "mage/backend/bootstrap",
18         "mage/adminhtml/globals"
19     ],
20 };
```

When you see `deps` as a top level key in a RequireJS configuration object, it means Magento will load the listed modules once it's finished loading RequireJS – i.e. on every HTML page load. The `mage/bootstrap` and `mage/backend/bootstrap`

are particularly interesting, as they enable some important functionality we'll mention below. However, **all** these modules are worth investigating, as they often set global state or configuration that your javascript application may depended on. i.e. if things are acting weird, it may be because of something in one of these modules.

Running RequireJS Programs in Magento

Earlier, we used the global `require` function to run a RequireJS based program. While this is a perfectly adequate way to run a javascript program in Magento 2, Magento provides us with **two** other ways to kick-off a RequireJS based program.

The first of these two methods are "`x-magento-init`" script tags. You can see these tags on most Magento pages. Here's one example

```
<script type="text/x-magento-init">
    {
        "*": {
            "mage/cookies": {
                "expires": null,
                "path": "/",
                "domain": ".magento.example.com",
                "secure": false,
                "lifetime": "3600"
            }
        }
    }
</script>
```

This may look like a standard javascript tag, but take a closer look. The `type` attribute is `text/x-magento-init`.

```
<script type="text/x-magento-init">
```

This means the browser **will not** execute code in the `script` tag on its own. Also, if you take a look at this `<script/>` tag's content

```
{
    "*": {
        "mage/cookies": {
            "expires": null,
            "path": "/",
            "domain": ".magento.example.com",
            "secure": false,
            "lifetime": "3600"
        }
```

```
10      }
11  }
```

you'll see there's nothing to execute – there's just a JSON object. What gives?

The key to understanding this are the `mage/backend/bootstrap` and `mage/bootstrap` modules we saw earlier. The javascript in these modules parses every Magento page for these `x-magento-init` tags, loads each RequireJS module listed in the nested object (`mage/cookies` above), and then executes the function returned by that module, passing in the configuration. Here's what that looks like in pseudo code

```
var moduleFunction = requirejs('mage/cookies');
moduleFunction({"expires": null,
                "path": "/",
                "domain": ".magento.example.com",
                "secure": false,
                "lifetime": "3600"});
```

The intention of these `x-magento-init` blocks is to allow PHP developers to generate a JSON string via PHP, and securely pass it to a RequireJS program for execution.

The Other Initialization

There's a second way to initialize a RequireJS program in Magento 2, and that's via a `data-mage-init` attribute in an HTML node. Again, here's an example from some Magento generated HTML

```
<ul id="main-dropdown" class="dropdown switcher-dropdown"
    data-mage-init='{"dropdownDialog":{
        "appendTo":"#switcher-currency > .options",
        "triggerTarget":"#switcher-currency-trigger",
        "closeOnMouseLeave": false,
        "triggerClass":"active",
        "parentClass":"active",
        "buttons":null}}'>
```

The `data-mage-init` attribute accepts a JSON object. The object's key (`dropdownDialog` above) is the name of a RequireJS module. When Magento's bootstrapping code encounters a `data-mage-init` attribute, it will load this RequireJS module, and call the function returned from this module, passing in the configuration object provided in the `data-mage-init` script. Again, in pseudo-code, here's what Magento will do when it sees a `data-mage-init` attribute.

```
var moduleFunction = requirejs('dropdownDialog');
moduleFunction('{
        "appendTo":"#switcher-currency > .options",
        "triggerTarget":"#switcher-currency-trigger",
        "closeOnMouseLeave": false,
        "triggerClass":"active",
        "parentClass":"active",
        "buttons":null}', jQuery('$main-dropdown'));
```

Finally, there's also a form of `x-magento-init` which behaves just like `data-mage-init` – just replace the `*` from the earlier `data-mage-init` with a `jQuery`/CSS selector.

```
<script type="x-magento-init">
    {
        ".switcher-dropdown":{
            "dropdownDialog":{
                "appendTo":"#switcher-currency > .options",
                "triggerTarget":"#switcher-currency-trigger
                    ",
                "closeOnMouseLeave": false,
                "triggerClass":"active",
                "parentClass":"active",
                "buttons":null
            }
        }
    }
</script>
```

The only difference here is `data-magento-init` allows us to target a **specific** DOM element, while `x-magento-init` allows us to target any number of DOM elements with generic selectors, **or** to target no DOM elements with the special `"*"` selector.

Next Steps

We've only touched the surface of what you can do with javascript in Magento 2. Below these initialization scripts, Magento 2 contains a completely custom javascript framework that includes its own javascript object system. Here's some online resources that will help you get to the core of what's going on in Javascript in Magento 2

Magento 2: Advanced Javascript

https://alanstorm.com/category/magento-2/#magento2-advanced-javascript

Magento 2 UI Components

https://alanstorm.com/category/magento-2/#magento-2-ui

Magento 2: uiElement Internals

https://alanstorm.com/category/magento-2/#magento-2-uielement-internals

Chapter 10

Advanced Front End Topics

Now that you're grounded in the basics of Magento's CSS and Javascript systems, there's a few advanced topics to cover. First, we'll cover everything you can do in a layout handle XML file's <head/> section. Second, we'll show you how to add arbitrary code to your HTML page's head sections. Third, we'll explain how those Magento CSS and Javascript URLs are served during development mode, which should help you debug missing file problems.

All <head/> directives

In earlier chapters we showed you how to add javascript and CSS/LessCSS files via a layout handle XML file's <head/> node.

```
1  <head>
2      <script src="Packagename_Modulename::path/to/file.js"/>
3      <css src="Packagename_Modulename::path/to/file.css"/>
4  </head>
```

There are a few other tags you can use inside <head/> that will influence your HTML page's output.

Head Tag Attributes

The <attribute/> tag will let you set an attribute on **your page's** <head/> tag. i.e. the following layout handle XML

```
1  <head>
2      <attribute name="foo" value="bar" />
3  </head>
```

will give you an HTML page that looks something like this

```
<html>
    <head foo="bar">
    </head>
    <body>
    </body>
</html>
```

Page title

The <title/> tag will let you give your page a title – i.e., you can set the text value of the HTML title node. Layout handle XML that looks like this

```
<head>
    <title>Ten Reasons Magento 2 isn't Right for your
        Business</title>
</head>
```

will give you an HTML page that looks something like this

```
<html>
    <head>
        <title>Ten Reasons Magento 2 isn't Right for your
            Business</title>
    </head>
    <body>
    </body>
</html>
```

Meta Tags

Similar to the <title/> tag, you'll use a layout update XML's <meta/> to create <meta/> tags in your HTML page. Layout update XML that looks like this

```
<head>
    <meta name="viewport" content="width=device-width,
        initial-scale=1"/>
</head>
```

will produce an HTML page that looks something like this

```
<html>
    <head>
        <meta name="viewport" content="width=device-width,
            initial-scale=1"/>
```

```
    </head>
    <body>
    </body>
</html>
```

The `<meta/>` tag also has a second form that uses a `property` attribute instead of a `name` attribute.

```
<head>
    <meta property="viewport" content="width=device-width,
        initial-scale=1"/>
</head>
```

In this second form, `property` is simply an alias for `name`. The above layout update XML will produce the same HTML output.

CSS, Link, and Script Tags

We've already discussed the `<css/>` tag in earlier chapters. You use it to add a CSS file, or a LessCSS file compiled to CSS, to your page

```
<head>
    <css src="path/to/file.css"/>
    <css src="path/to/file.less"/>
</head>
```

Layout handle XML files also support a `<link/>` tag.

```
<head>
    <link src="path/to/file.xxx" />
</head>
```

You can use the `<link/>` to create HTML `<link/>` elements to files that are not CSS files (an RSS file, for example).

And of course, layout handle XML files also support a `<script/>` tag for adding javascript files to your page.

```
<head>
    <script src="path/to/file.js" />
</head>
```

Together, these three tags are known as *asset* tags in Magento 2. Asset tags all share a few extra abilities we haven't yet discussed. For example, by default, the `src` attribute is a path to a file, or a Magento URN

```
<script src="path/to/file.js" />
```

```
2 <script src="Package_Module::path/to/file.js" />
```

In both these cases, Magento will generate a final URL that reflects the location of Magento's static asset folder relative to the root and the current area/theme. If you want to use a full URL instead of these local file paths, you can specify this via the `src_type` attribute.

```
1 <script src="https://example.com/path/to/file.js" src_type=
      "url" />
2 <css src="https://example.com/path/to/file.js"    src_type=
      "url" />
3 <link src="https://example.com/path/to/file.js"   src_type=
      "url"  />
```

A `src_type` of URL will allow you to set a hard-coded full URL path. The default is `src_type="resource"`, which treats the `src="..."` value as a Magento asset resource (and expands the path accordingly). The final (if little used) valid value for `src_type` is `src_type="controller"`. This allows you to use a Magento MVC controller path as the `src="..."` value. If you don't understand what an MVC controller is, don't worry. It won't be on the test.

If you use an attribute name other than `src_type` in these three asset tags Magento will pass that attribute along to the final rendered HTML. For example, the following layout update XML

```
1 <css src="css/styles-l.css" media="screen and (min-width:
      768px)"/>
```

will pass along the `media` attribute to the final rendered HTML.

```
1 <link  rel="stylesheet"
2        type="text/css"
3        media="screen and (min-width: 768px)"
4        href="http://magento.example...css/styles-l.css" />
```

The only exceptions to this are the `content_type="..."` and `ie_condition="..."` attributes. Magento used the `content_type` tag in a previous version to set whether an asset was a CSS or Javascript file. Similarly, Magento used the `ie_condition` attribute to surround the linked asset in Internet Explorer conditional comments. Magento 2 uses neither of these tags in its current set of layout handle XML files, and their futures are uncertain. Because of that uncertain future, we recommend you avoid using them.

Removing an Asset

The final `<head/>` tag to cover is the `<remove/>` tag

```
<head>
    <remove src="path/to/file.css" />
</head>
```

The `<remove/>` tag allows you to remove a file added by a different layout handle XML file. All you need to do is pass in the `src` of the file (matching the `src` from the other layout handle XML file) and Magento will skip adding that particular asset to the page.

The head.additional Block

In addition to using Magento 2's layout XML system to automatically add front end asset URLs to your project, you can also create these URLs via PHP using a `Magento\Framework\View\Asset\Repository` object.

We'll show you how to do this below, as well as how to add arbitrary HTML to the `<head/>` of a Magento HTML page.

Starting with the later, add the following node to our previously created `default.xml` layout handle XML file

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/layout/default.xml
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/page_configuration.xsd">
    <body>
        <referenceBlock name="head.additional">
            <block  template="Pulsestorm_Nofrillslayout::chapter-9/head.phtml"
                class="Pulsestorm\Nofrillslayout\Block\Head"
                name="pulsestorm_javascriptcssexample_block_head" />
        </referenceBlock>
    </body>
    <!-- ... -->
</page>
```

The above code

1. Gets a reference to the `head.additional` block
2. Creates a new `Pulsestorm\Nofrillslayout\Block\Head` block named `pulsestorm_javascriptcssexample_block_head` that uses the `Pulsestorm_Nofrillslayout::chapter-9/head.phtml` template.

3. Adds that new block to the `head.additional` block using the reference from #1

The `head.additional` block is a special block. Any block added to `head.additional` will automatically be output into the `<head/>` area of a Magento page. It's a little weird that we're operating on the `<head/>` of a the document in the `<body/>` section of the layout update xml file, but sometimes Magento's a little weird.

Regardless, once we've got the layout XML in place, we'll want to create our new `Head` block class

```
#File: app/code/Pulsestorm/Nofrillslayout/Block/Head.php
<?php
namespace Pulsestorm\Nofrillslayout\Block;
class Head extends \Magento\Framework\View\Element\Template
{
}
```

As well as a template

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    templates/chapter-9/head.phtml
<!-- Hello There -->
```

With the above in place, clear your Magento cache and reload your page. You should see the `<!-- Hello There -->` comment in your page's `<head/>` node.

```
<!-- Hello There -->
</head>
```

With a new template rendered in `<head/>`, we're ready to render an asset URL using the asset repository.

The Asset Repository

The `Magento\Framework\View\Asset\Repository` object allows us to create asset objects. Asset objects can convert a file identifier like `foo/test.js` or `Pulsestorm_Nofrillslayout::test.js` into a full URL.

Magento blocks come with an asset repository ready for us to use, but we need to do something a little weird to use it. Change your `Head.php` file so it matches the following

```
#File: app/code/Pulsestorm/Nofrillslayout/Block/Head.php
<?php
namespace Pulsestorm\Nofrillslayout\Block;
```

```
class Head extends \Magento\Framework\View\Element\Template
{
    public $assetRepository;
    public function __construct(
        \Magento\Framework\View\Element\Template\Context
            $context,
        array $data = []
    )
    {
        $this->assetRepository = $context->
            getAssetRepository();
        return parent::__construct($context, $data);
    }
}
```

Magento block objects have so many dependencies that the Magento core team created a single `Magento\Framework\View\Element\Template\Context` object to manage them all. This includes the asset repository object.

We've grabbed the asset repository from the context object, and assigned it to the `assetRepository` property of our block object. The other parameters in `__construct` and the call to `parent::__construct` are there for compatibility with the base template block class. Also, notice we made `assetRepository` a **public** property. This means we'll be able to access it in our `phtml` template.

Edit your `head.phtml` file so it matches the following.

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    templates/chapter-9/head.phtml
<?php
    $asset_repository = $this->assetRepository;
    $asset  = $asset_repository->createAsset('
        Pulsestorm_Nofrillslayout::test.js');
    $url    = $asset->getUrl();
?>
<!-- Hello There -->
<script src="<?php echo $url; ?>"></script>
```

With the above in place, clear your cache, delete the files in `var/generate/*` (because you changed an automatic constructor dependency injection constructor), and reload the page. If you view the raw HTML source, you should see a new `<script></script>` tag rendered with a full asset URL.

```
<script src="http://magento.example.com/static/
    version1514092162/frontend/Pulsestorm/luma/en_US/
    Pulsestorm_Nofrillslayout/test.js"></script>
```

What we've done above is use the `createAsset` method of the asset repository object to create an asset object. Then, we use the `getUrl` method of the asset

object to fetch the url of the asset. All we need to know is the file identifier – Magento handles the grunt work of pulling together the correct URL path parameters for us.

Static Files Serving

To close out this chapter, we're going to take an in depth look at the `static` /`pub`/`static` issue. We've already mentioned that Magento 2 ships with **two** `index.php` files.

```
1 /path/to/magento2/index.php
2 /path/to/magento2/pub/index.php
```

One is at the top level of Magento 2's distribution folder. The second is inside the "`pub`" folder. There's also separate, but similar, `.htaccess` files in each folder.

The file you **want** to use for your Magento system's root folder is `pub`. This is a modern PHP framework convention, and is one layer in a layered approach to protecting your PHP and configuration files from being accidentally exposed to the world. However, Magento still ships with the root level `index.php` because many hosting companies make changing your web root difficult, or impossible.

Whatever folder you end up using will have consequences for the paths Magento generates to front end asset files. For the purposes of this chapter, unless we explicitly state otherwise, assume we've setup our web server to point to the `pub` folder. We're also assuming you're still running your system in `developer` mode.

Serving a Front End Asset File

Let's take a look at each segment of a Magento front end asset URL.

```
1 http://magento.example.com      #domain name
2 /pub/static                     #base public asset path
3 /version1514092162              #version slug
4 /frontend/Magento/luma          #magento area (frontend,
      adminhtml, etc.)
5 /en_US                          #theme identifier
6 /Package_Module                 #module identifier (
      optional)
7 /path/to/file.css               #asset path
```

Domain Name

The domain name is your website's configured domain name. It may seem obvious and redundant to mention this, but Magento does allow you to configure a domain name to use site wide at both

```
1 Stores -> Configuration -> Web -> Base URLs
2 Stores -> Configuration -> Web -> Base URLs (Secure)
```

Subtle differences between what you think your site's URL is and what it actually is (https vs. http, www vs. non-www) can sometimes cause problems with your assets, so be sure you know what's going on with your system here.

Base Public Path

Next we have the base static asset path. As mentioned throughout this book, this path will vary depending on whether you've configured your web server to use Magento's root `index.php` file, or the `pub/index.php` file.

Version Slug

The version slug may be a little confusing, and almost certainly features a different number in your Magento system. This is here to "cache bust" the URLs when you deploy a new set of assets. Web browsers can be aggressive about caching CSS, Javascript, and other front end assets. This caching is problematic if you're changing these files on a regular basis. By injecting this extra portion in an asset's URL, Magento tricks the browser into thinking it's a new file that needs to be downloaded and re-cached.

Magento Area

This portion of the URL is the Magento area. Different areas (the `frontend` cart, the back office `adminhtml`) may have a javascript/css file with the same name and path, but with different contents.

Theme Identifier

This portion of the URL is your theme's package name (`Magento` above) and your theme name (`luma`). This segment needs to be here since Magento's theming system allows you to replace specific front end asset files.

Locale

This portion of the URL is the locale, (usually thought of as language), identifier for a site. It may seem a little excessive to allow different locales to have different CSS and Javascript files, but if you consider things like CSS `:after` text, or the need to localize string constants in UI libraries, it makes sense.

Asset Path

Finally, we have the path to the asset file.

Module Name (optional)

If your file is a part of a module (as opposed to being a part of a theme), this portion of the URL is the `Packagename_Modulename` module identifier for the module where the asset is located.

Turning URLs into Paths

Magento will take the above URL and transform it into to a file path. For example, this URL for the stock Magento `style-m.css` file

```
1 http://magento.example.com
2 /pub/static
3 /version1514092162
4 /frontend
5 /Magento
6 /luma
7 /en_US
8 /css
9 /styles-m.css
```

corresponds to the following file path

```
1 $ ls -lh ./pub/static/frontend/Magento/luma/en_US/css/
      styles-m.css
2 -rw-rw-r--  1 _www  staff   334K Dec 23 21:09 ./pub/static/
      frontend/Magento/luma/en_US/css/styles-m.css
```

That is, Magento takes each segment (minus the version slug) and treats it as a path on your file system.

Some of you may be scratching your head – we've never created or mentioned a `styles-m.css` file in the `pub/static` folder. Where did this file come from?

Here's a quick experiment that will help clear that up. You'll need to be in developer mode for this to work.

First, lets use `curl` to make sure there's a `styles-m.css` file. We'll use `curl`'s `-I` modifier to look at the HTTP response headers, but you can also just load the URL directly in a browser. Remember, the `version...` segment of your URL will be different from the one below. You also may or may not need the `pub` segment on your URL.

```
1 curl -I 'http://magento.example.com/pub/static/
      version1514092162/frontend/Magento/luma/en_US/css/
      styles-m.css'
2 HTTP/1.1 200 OK
3 //...
```

So, that's a `200 OK` code – that means the file's there. Next, let's **remove** `styles-m.css`

```
1 $ rm ./pub/static/frontend/Magento/luma/en_US/css/styles-m.
      css
```

With the file removed, lets try downloading it again. We'd expect to get a `404 Not Found` error – except

```
1 curl -I 'http://magento.example.com/pub/static/
      version1514092162/frontend/Magento/luma/en_US/css/
      styles-m.css'
2 HTTP/1.1 200 OK
3 //...
```

The URL still serves the file! What's going on?!

Your first thought may be that we've told you an un-true thing, and that the `styles-m.css` file we pointed you at isn't the file that Magento's serving. At least, that's what I thought the first time I encountered this behavior. However, if you take a look at your file system again

```
1 $ ls -lh ./pub/static/frontend/Magento/luma/en_US/css/
      styles-m.css
2 -rw-rw-r--  1 _www  staff   334K [today] ./pub/static/
      frontend/Magento/luma/en_US/css/styles-m.css
```

You'll see that **something** has recreated the file for us!

Static Asset Serving

When you deploy a Magento system to production, you need to run the following command

```
$ php bin/magento setup:static-content:deploy
```

This command looks through all of Magento's module's and themes for front end assets, and then creates them in the `pub/static` folder file hierarchy we discussed above.

While this is an OK solution for a production system – **developing features** under this system would be incredibly tedious and time consuming. Change a file, deploy the site, wait, whoops that wasn't the right change, repeat. To combat this problem, Magento deploys front end assets **on demand** when you're running in developer mode.

If we take a look inside the `pub/static` folder, we'll find an `.htaccess` file

```
#File: pub/static/.htaccess
# ...
<IfModule mod_rewrite.c>
    RewriteEngine On

    # Remove signature of the static files that is used to
        overcome the browser cache
    RewriteRule ^version.+?/(.+)$ $1 [L]

    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-l

    RewriteRule .* ../static.php?resource=$0 [L]
</IfModule>
# ...
```

This rewrite rule takes every request for a file in a `pub/static/version*` folder and, **if the file doesn't already exist**, routes the request to the program in `static.php`. In other words, a request for this URL

```
http://magento.example.com/pub/static/version1514092162/
    frontend/Magento/luma/en_US/css/styles-m.css
```

is **actually** a request for

```
http://magento.example.com/pub/static.php?resource=frontend
    /Magento/luma/en_US/css/styles-m.css
```

The pub/static.php program is Magento's static asset server. Its job is to use the resource parameter to find a front end asset file in Magento's themes and modules, save that file to disk for future requests, and serve the file.

The .htaccess file is also the place where the version... slug is stripped out of the URL. If you're using the nginx web server, Magento has a sample configuration with similar rules

```
#...

location ~ ^/static/version {
    rewrite ^/static/(version\d*/)?(.*)$ /static/$2 last;
}

#...

if (!-f $request_filename) {
    rewrite ^/static/?(.*)$ /static.php?resource=$1 last;
}

#...
```

Be careful with your nginx configuration – it's not uncommon for devops engineers to remember one of these rules, but forget the other. If you're not seeing front end assets, this is the first place to look.

The Static Asset Server

If we take a look at static.php

```
#File: pub/static.php
require realpath(__DIR__) . '/../app/bootstrap.php';
$bootstrap = \Magento\Framework\App\Bootstrap::create(BP,
    $_SERVER);
/** @var \Magento\Framework\App\StaticResource $app */
$app = $bootstrap->createApplication(\Magento\Framework\App
    \StaticResource::class);
$bootstrap->run($app);
```

We see code that bootstraps and runs an independent application. This is similar to (but simpler than) what happens in Magento's main index.php file. If we take a look at that application's source file, we can see the main work of the application in the launch method (which is eventually called after the run method above)

```
#File: vendor/magento/framework/App/StaticResource.php
public function launch()
```

```
3  {
4      // disabling profiling when retrieving static resource
5      \Magento\Framework\Profiler::reset();
6      $appMode = $this->state->getMode();
7      if ($appMode == \Magento\Framework\App\State::
           MODE_PRODUCTION) {
8          $this->response->setHttpResponseCode(404);
9      } else {
10         $path = $this->request->get('resource');
11         $params = $this->parsePath($path);
12         $this->state->setAreaCode($params['area']);
13         $this->objectManager->configure($this->configLoader
               ->load($params['area']));
14         $file = $params['file'];
15         unset($params['file']);
16         $asset = $this->assetRepo->createAsset($file,
               $params);
17         $this->response->setFilePath($asset->getSourceFile
               ());
18         $this->publisher->publish($asset);
19     }
20     return $this->response;
21 }
```

We're not going to dive deep into this program, but here's a few lines worth paying attention to.

```
1  #File: vendor/magento/framework/App/StaticResource.php
2  if ($appMode == \Magento\Framework\App\State::
       MODE_PRODUCTION) {
3      $this->response->setHttpResponseCode(404);
4  }
```

If Magento detects you're in production mode, it automatically returns a 404. Magento still needs to use the `.htaccess` rules from above to strip the `version` ... string from URLs, but doesn't want to dynamically serve files in production mode for performance and security reasons.

These three lines

```
1  #File: vendor/magento/framework/App/StaticResource.php
2
3  $asset = $this->assetRepo->createAsset($file, $params);
4  $this->response->setFilePath($asset->getSourceFile());
5  $this->publisher->publish($asset);
```

are where the bulk of the work happens. The `createAsset` method uses the `resource` path to find an actual file. The call to `setFilePath` on the response object is what tells Magento how to send the front end file to the browser **for**

this request only. The call to `publish` on the `publisher` object is what saves the file to the `pub/static` folder.

In an ideal world, you wouldn't need to be aware of the static serving application. However – when there's bugs in this application you may need to go on a deep dive here to diagnose the problem. Also, if your file permissions aren't set correctly Magento may not be able to create the file in your `pub/static` folder. Beyond being annoying, the large number of Javascript files Magento needs means there are dozens of requests to this program when you first start using a Magento system.

Wrap Up

And there we have it. Everything you ever wanted to know about HTML rendering and front end coding in Magento 2 but were afraid to ask. You're now ready to start slinging HTML, CSS, and Javascript like it's was 2004!

In our final chapter, we're going to take a look at Magento 2's attempt to create a new, more modern javascript development experience. While far from comprehensive, knowing these newer systems (or knowing how to stay out of their way) is just as important as understanding Magento's Layout XML system.

Chapter 11

Registering Knockout.js Custom Scopes

To close out the book, we're going to take a deeper look at Magento's javascript based, front end view model system. Unfortunately, we don't have the time or space to cover every topic in the depth that we've covered Magento's layout XML. In other words, this chapter assumes some knowledge we haven't yet covered in this book. If you're stuck on a fundamental concept or specific problem remember that help is just a Stack Overflow question away.

Magento's front end template library is the venerable Knockout.js. While not the latest and greatest, Knockout is a well documented library with known patterns and behaviors. However, if you look at the source code of a Magento page looking for a Knockout.js view model, you may be a little confused. There's no obvious place where Magento uses Knockout's `ko.applyBindings` method to bind a view model to the page.

That's because Knockout.js's default mechanisms start to fall apart when you have many different developers trying to use the Knockout.js view model for their own template and business logic. A single view model is perfect for a three person team building a marketing site. It's less perfect for a team of thirty, supporting a community thousands of developers strong.

If you load the Magento homepage, you'll see HTML that looks something like this.

```
1 <li class="greet welcome" data-bind="scope: 'customer'">
2     <span data-bind="text: customer().fullname ? $t('
          Welcome, %1!').replace('%1', customer().fullname) :
           'Default welcome msg!'"></span>
3 </li>
```

This looks like standard Knockout.js template code, with one glaring exception

```
1 data-bind="scope: 'customer'"
```

The `scope` binding is unfamiliar, even to experienced Knockout.js developers. That's because this is a **custom** binding, written by the Magento core team. When you say `scope: customer`, you're telling Knockout.js that the inner nodes are bound to the "customer" view model.

In this way, different modules can apply different Knockout.js view models to different areas of the page, and everyone's code can coexist in harmony.

You're probably wondering **how** to create these scoped view models. That's what we'll cover in this chapter.

Getting Started

Load up this chapter's URL in your browser

```
1 http://magento.example.com/pulsestorm_nofrillslayout/
      chapter10
```

We've set you up with a page that renders the following HTML

```
1 <div data-bind="scope: '
      pulsestorm_nofrillslayout_chapter10_viewmodel'">
2     <p data-bind="text:helloMessage"></p>
3 </div>
```

This is an HTML `div` that uses Magento's Knockout.js scope binding. This scope binding binds the view model named `pulsestorm_nofrillslayout_chapter10_viewmodel` to the inner nodes. The inner nodes render the `helloMessage` property of the view model using Knockout.js's standard `data-bind` attribute.

There's one problem – we haven't registered a view model named `pulsestorm_nofrillslayout_chapter10_viewmodel`. To do that, we'll need to add a bit of `x-magento-init` javascript to the page.

First, let's change our module to use a custom template. Open the following layout handle XML file and add these nodes to the bottom

```
1 #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
      layout/pulsestorm_nofrillslayout_chapter10_index.xml
2 <?xml version="1.0"?>
3 <page xsi:noNamespaceSchemaLocation="urn:magento:framework:
      View/Layout/etc/page_configuration.xsd"
4       layout="1column">
5
```

```
    <!-- ... leave the existing nodes in place ... -->

    <!-- START: add the following new nodes -->
    <referenceBlock name="
        pulsestorm_nofrillslayout_chapter10_hello">
        <action method="setTemplate">
            <argument name="template" xsi:type="string">
                Pulsestorm_Nofrillslayout::chapter10/user/
                main.phtml</argument>
        </action>
    </referenceBlock>
    <!-- END: add the following new nodes -->
</page>
```

and then create a new template file for our block

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    templates/chapter10/user/main.phtml
<h1>Hello Custom Template</h1>
<div data-bind="scope: '
    pulsestorm_nofrillslayout_chapter10_viewmodel'">
    <p data-bind="text:helloMessage"></p>
</div>
```

Clear your cache and reload the page. If you see the added title text `Hello Custom Template` then your changes were successful.

Creating the x-magento-init Script

The new template we created uses the same Knockout.js template code as the default template. We're going to add an `x-magento-init` script that will register a view model named `pulsestorm_nofrillslayout_chapter10_viewmodel`.

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    templates/chapter10/user/main.phtml
<h1>Hello Custom Template</h1>
<div data-bind="scope: '
    pulsestorm_nofrillslayout_chapter10_viewmodel'">
    <p data-bind="text:helloMessage"></p>
</div>

<!-- START: add this code -->
<script type="text/x-magento-init">
{
    "*": {
        "Magento_Ui/js/core/app": {
            "components": {
```

```
13                  "
                        pulsestorm_nofrillslayout_chapter10_viewmodel
                        ": {
14                      "component": "Magento_Ui/js/lib/core/
                            element/element",
15                      "helloMessage":"Hello View Model"
16                  }
17              }
18          }
19      }
20  }
21  </script>
22  <!-- END:   add this code -->
```

There's a lot happening in the above code snippet. Before we get to it, try reloading your Magento page. You should see the *Hello View Model* text in your page. Congratulations! You just created your first scoped Magento Knockout.js view model.

The above code uses the `x-magento-init` mechanism to run the program in the `Magento_Ui/js/core/app` RequireJS module. The `Magento_Ui/js/core/app` RequireJS module is responsible for registering scoped Knockout.js view models based on the configuration passed in. Let's look at that configuration, piece by piece.

```
1  {
2      "components": {
3          /* ... */
4      }
5  }
```

The JSON object we pass to `Magento_Ui/js/core/app` has a single key, named `components`. Magento's internal name for *scoped Knockout.js view models* is "components". Magento also calls other things components, so we're going to stick to calling them *scoped Knockout.js view models*.

Inside this `components` key is another JSON object

```
1  "pulsestorm_nofrillslayout_chapter10_viewmodel": {
2      /* ... */
3  }
```

This is another object of key/value pairs. The key (`pulsestorm_nofrillslayout_chapter10_viewmodel`) is the name of the view model you want to register. The value

```
1  {
2      "component": "Magento_Ui/js/lib/core/element/element",
3      "helloMessage":"Hello View Model"
4  }
```

is the configuration for the specific RequireJS module that Magento will load and use as a constructor function for your new view model. The `component` key is the RequireJS module, (`Magento_Ui/js/lib/core/element/element` above). This RequireJS module is Magento's *base* constructor function for view models.

Magento will use the other keys at this level (`helloMessage` above) as data properties for your view model object. Because we said

```
"helloMessage":"Hello View Model"
```

in our JSON, Knockout.js has access to a `helloMessage` variable in its HTML template

```
<p data-bind="text:helloMessage"></p>
```

An Aside on uiElement Objects

The `Magento_Ui/js/lib/core/element/element` module is the full Magento path to a RequireJS module. However, this module is one that Magento has aliased using the RequireJS `map` feature. You'll usually see this module referenced as `uiElement`. In other words, the following configuration

```
{
    "component": "uiElement",
    "config":{
        "helloMessage":"Hello View Model"
    }
}
```

will behave the same as one with a `Magento_Ui/js/lib/core/element/element` identifier. The `uiElement` objects are actually part of a complex and powerful javascript system for giving constructor functions class-like inheritance. While we'll get into this system a little bit in this chapter, it's beyond our scope to describe it in full. What you need to know for now is when Magento instantiates your view model, it does so with pseudo-code that's equivalent to something like this

```
requirejs(['uiElement'], function(UiElement){
    var viewModel = new UiElement;
    viewModel.helloMessage = "Hello View Model";
    return viewModel;
});
```

Magento uses the function returned by the `uiElement`/`Magento_Ui/js/lib/core/element/element` module as a javascript constructor function. Magento then assigns each data key in `config` to the object as a value.

The actual inner workings of this `uiElement` object system are much more complex. Fortunately, there's articles covering this system online

```
Magento 2: Advanced Javascript
http://alanstorm.com/series/magento2-advanced-javascript/

UI Components
http://alanstorm.com/series/magento-2-ui/

UiElement Internals
http://alanstorm.com/series/magento-2-uielement-internals/
```

While you can get by without understanding all the nitty gritty details of this object system, if you're interested in becoming a true Magento 2 master, the above articles are an important step in that journey.

Magento and Knockout.js Templates

Knockout.js has a template data-binding. Try editing our `phtml` file so the HTML portion matches the following

```
<h1>Hello Custom Template</h1>
<div data-bind="scope: '
    pulsestorm_nofrillslayout_chapter10_viewmodel'">
    <!-- ko template: "ourKnockoutTemplateId"--><!-- /ko
        -->
</div>

<script type="text/html" id="ourKnockoutTemplateId">
    <h2>Rendered in a Knockout.js Template</h2>
    <p data-bind="text:helloMessage"></p>
</script>

<!-- leave x-magento-init in place -->
```

Reload the page, and you should see the slightly modified HTML output.

We've replaced the text inside our `scope` div with a call to the tag-less form of the Knockout.js template binding.

```
<!-- ko template: "ourKnockoutTemplateId"--><!-- /ko -->
```

This tells Knockout to render the template named `ourKnockoutTemplateId`. We define this template via a `script` tag

```
<script type="text/html" id="ourKnockoutTemplateId">
    <h2>Rendered in a Knockout.js Template</h2>
```

```
    <p data-bind="text:helloMessage"></p>
</script>
```

Notice the `id="ourKnockoutTemplateId"` – this identifies the template, and is the `id` we we pass to the template binding.

While neat, the template system suffers from the fact that you need to render the template you want to use as separate HTML DOM nodes (a `<script/>` with a `type` of `text/html`). With this default Knockout.js template handling, there's no easy way to create a library of client side templates. Magento's solved this problem by extending Knockout.js to work with templates loaded via URLs. This allows you to store your Knockout.js templates in plain `.html` files, and only load in the ones you need via the template binding.

If that didn't make sense, a sample should make it clear.

Change the HTML section of `main.phtml` so it matches the following

```
<div data-bind="scope: '
    pulsestorm_nofrillslayout_chapter10_viewmodel'">
    <p>
        We're going to try rendering our view model's
            template.
    </p>
    <div>
        <!-- ko template: 'Pulsestorm_Nofrillslayout/
            chapter10/remote-template' --><!-- /ko -->
    </div>
</div>

<!-- leave x-magento-init in place -->
```

We've replaced the template identifier in the template binding with `Pulsestorm_Nofrillslayout/chapter10/remote-template`. This identifier is actually a static Knockout.js template URN that corresponds with the file at

```
app/code/Pulsestorm/Nofrillslayout/view/frontend/web/
    template/chapter10/remote-template.html
```

The first portion of the URN (`Pulsestorm_Nofrillslayout`) indicates the module folder where Magento can find your template. The second portion (`chapter10/remote-template`) indicates the folder path and file, **from the base** `web/[area]` folder, and appended with the `.html` file extension. In other words, the following file (which we've included as part of the book's default code)

```
view/frontend/web/template/chapter10/remote-template.html
```

Because this folder/file is in our module's `web` folder, this template is also available via a URL something like the following (remember, the `version...` string

will be different on your system).

```
http://magento.example.com/static/version1514092162/
    frontend/Pulsestorm/dram/en_US/
    Pulsestorm_Nofrillslayout/template/chapter10/remote-
    template.html
```

If you reload our page with the edited `main.phtml`, you'll see the contents of `remote-template.html` rendered. Behind the scenes, Magento translates `Pulsestorm_Nofrillslayout/chapter10/remote-template` into a URL like the one above, loads the `.html` file via AJAX, and then uses the loaded HTML as a Knockout.js template.

This template has the same functionality of any standard Knockout.js template loaded via a `text/html` `<script/>` tag. For example, lets create our own template that references the `helloMessage` view variable.

```
#File: view/frontend/web/template/chapter10/user/remote-
    template.html
<h2>This is a user generated Magento 2 remote Knockout.js
    template.</h2>
<p data-bind="text:helloMessage"></p>
```

and then change our template binding to reference the new template URN

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    templates/chapter10/user/main.phtml
<div>
    <!-- ko template: 'Pulsestorm_Nofrillslayout/chapter10/
        user/remote-template' --><!-- /ko -->
</div>
```

If you reload the page with the above in place, you should see the `helloMessage` variable rendered inside the `<p></p>` tags.

The `remote-template.html` file uses our view model because we loaded it inside our `data-bind="scope: pulsestorm_nofrillslayout_chapter10_viewmodel"` node. If we'd loaded it inside another node, it would be bound to that view model.

While this approach comes with its own set of tradeoffs (less transparency, more individual network requests to load templates) its main advantage is a server agnostic way to organize templates into files, and the ability to reduce the size of your initial page load (i.e. it's not bloated with `<script type="text/html"/>` tags your code may or may not use).

Using uiRegistry to Debug View Models

One thing that can be frustrating with Knockout.js, particularly with Magento's scoped view models, is the inability to examine your view model object. While tools like Google Chrome's Knockout.js Context Debugger https://chrome.google.com/webstore/detail/knockoutjs-context-debugg/oddcpmchholgcjgjdnfjmildmlielhof?hl=en

can help, the lack of interactive debugging is a hindrance.

Fortunately, if you're comfortable with your browser's javascript debugger, there is a solution. When you create a scoped view model with the `Magento_Ui/js/core/app` application, Magento registers and stores each view model via the `uiRegistry` RequireJS module. You can use the following javascript to peek at any named view model.

```
1 //normally you wouldn't use requirejs directly like this
2 //but for debugging purposes it's safe.
3 > var uiRegistry  = requirejs('uiRegistry');
4 > var viewModel   = uiRegistry.get('
      pulsestorm_nofrillslayout_chapter10_viewmodel');
5 > console.log(viewModel);
```

The above code uses the `uiRegistry`'s `get` method to fetch our named view model. You can use this to fetch **any** named view model used in a `scope` binding, and then examine the fetched model with your javascript debugger.

Rendering Multiple Templates/View Models

There may be times where you want to render multiple templates/view models. For this, Magento created a special `uiCollection` view model constructor. Let's try a `main.phtml` that looks like the following

```
1  #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
       templates/chapter10/user/main.phtml
2
3  <h1>Hello Collection of  Template</h1>
4  <div data-bind="scope: '
       pulsestorm_nofrillslayout_chapter10_viewmodel'">
5      <div>
6          <!-- ko template: getTemplate() --><!-- /ko -->
7      </div>
8  </div>
9
10 <!-- START: add this code -->
11 <script type="text/x-magento-init">
```

```
{
    "*": {
        "Magento_Ui/js/core/app": {
            "components": {
                "
                    pulsestorm_nofrillslayout_chapter10_viewmodel
                    ": {
                    "component": "Magento_Ui/js/lib/core/
                        collection"
                }
            }
        }
    }
}
</script>
```

Based on what we've learned so far, the above code creates a scoped view model. The following DOM code lists this view model as the `scope` for its inner nodes.

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    templates/chapter10/user/main.phtml

<div data-bind="scope: '
    pulsestorm_nofrillslayout_chapter10_viewmodel'">
    <div>
        <!-- ko template: getTemplate() --><!-- /ko -->
    </div>
</div>
```

and calls that view model's `getTemplate` method to render a Knockout.js template.

If we take a look at the view model configuration

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    templates/chapter10/user/main.phtml

<!-- START: add this code -->
<script type="text/x-magento-init">
{
    "*": {
        "Magento_Ui/js/core/app": {
            "components": {
                "
                    pulsestorm_nofrillslayout_chapter10_viewmodel
                    ": {
                    "component": "Magento_Ui/js/lib/core/
                        collection"
                }
            }
```

```
        }
    }
}
</script>
```

We see the `pulsestorm_nofrillslayout_chapter10_viewmodel` view model is a `Magento_Ui/js/lib/core/collection`.

Note: This RequireJS module is another one that's aliased. The `Magento_Ui/js/lib/core/collection` module is also available with the `uiCollection` name. i.e., the above is equivalent to

```
<script type="text/x-magento-init">
{
    "*": {
        "Magento_Ui/js/core/app": {
            "components": {
                "
                    pulsestorm_nofrillslayout_chapter10_viewmodel
                    ": {
                    "component": "uiCollection"
                }
            }
        }
    }
}
</script>
```

The "collection" here has nothing to do with Magento 2's backend collection models. Instead, the `uiCollection` module *collects* together child view models.

If you reload the page – you'll only see the HTML that the server renders. There's nothing from the Knockout.js template. If we peek at the `pulsestorm_nofrillslayout_chapter10_viewmodel` view model and call its `getTemplate` method

```
//from your javascript console
uiRegistry = requirejs('uiRegistry');
viewModel  = uiRegistry.get('
    pulsestorm_nofrillslayout_chapter10_viewmodel');
console.log(viewModel.getTemplate());
```

The output should be

```
ui/collection
```

This means Magento's rendering the `ui/collection` template URN. This corresponds to the following stock template file.

```
#File: vendor/magento//module-ui/view/base/web/templates/
    collection.html
<each args="data: elems, as: 'element'">
    <render if="hasTemplate()"/>
</each>
```

So, there's a number of different things going on here. First – the reason we didn't see any output for our `uiCollection` model? We didn't configure any children. A `uiCollection` view model works because the `collection.html` template will `foreach` over all the child templates and render them. Since we didn't configure any children, there was nothing to render.

Second, even to an experienced Knockout.js developer, the above template looks super weird. Knockout.js doesn't have an `<each/>` tag or a `<render/>` tag. Those `if` and `args` attributes look weird as well – Knockout.js uses the `data-bind` attribute for that sort of thing. What gives?

If turns out that, while these remote `.html` template **are** Knockout.js templates, Magento has **enhanced** these template with an advanced syntax that replaces many of Knockout.js's tag-less bindings with tag versions. For example, the above template translates into the following "raw" Knockout.js code

```
<!-- ko foreach: {data: elems, as: 'element'} -->
    <!-- ko if: hasTemplate() --><!-- ko template:
        getTemplate() --><!-- /ko --><!-- /ko -->
<!-- /ko -->
```

Covering all these tags is beyond the scope of this chapter, but you can read about them online

https://alanstorm.com/design-problems-with-magentos-knockoutjs/

Adding Child View Models

Configuring children is relatively simple – just add a `children` node to your configuration

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    templates/chapter10/user/main.phtml

{
    "*": {
        "Magento_Ui/js/core/app": {
            "components": {
                "
                    pulsestorm_nofrillslayout_chapter10_viewmodel
                    ": {
```

```
                    "component": "uiCollection"
                    "children":{
                        /* ... */
                    }
                }
            }
        }
    }
}
```

And inside that node add a list of key/value pairs, each one configuring a child view model object.

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    templates/chapter10/user/main.phtml

"*": {
    "Magento_Ui/js/core/app": {
        "components": {
            "pulsestorm_nofrillslayout_chapter10_viewmodel"
                : {
                "component": "Magento_Ui/js/lib/core/
                    collection",
                "children":{
                    "child1":{
                        "component":"uiElement",
                        "template":"
                            Pulsestorm_Nofrillslayout/
                            chapter10/child-one"
                    },
                    "child2":{
                        "component":"uiElement",
                        "template":"
                            Pulsestorm_Nofrillslayout/
                            chapter10/child-two"
                    }
                }
            }
        }
    }
}
```

Each key (`child1`, `child2`) is the name of your child view model, and each object is another scoped view model configuration, the same as your top level configuration. You'll notice that in addition to setting the `component` property to `uiElement`, we've also set the `template` property

```
"child1":{
    /* ... */,
```

```
    "template":"Pulsestorm_Nofrillslayout/chapter10/child-
        one"
},
"child2":{
    /* ... */,
    "template":"Pulsestorm_Nofrillslayout/chapter10/child-
        two"
```

We've provided the `child-one.html` and `child-two.html` templates in the sample module

```
app/code/Pulsestorm/Nofrillslayout/view/frontend/web/
    template/chapter10/child-one.html
app/code/Pulsestorm/Nofrillslayout/view/frontend/web/
    template/chapter10/child-two.html
```

In a stock system, any **child** of a `uiCollection` view model will **automatically** have its template rendered. This is slightly different from the top level view model, which requires an explicit call to Knockout's template binding.

```
<!-- ko template: getTemplate() --><!-- /ko -->
```

A bit lost? Modify your `main.phtml` template so it matches the following.

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    templates/chapter10/user/main.phtml
<h1>Hello Collection of  Template</h1>
<div data-bind="scope: '
    pulsestorm_nofrillslayout_chapter10_viewmodel'">
    <div>
        <!-- ko template: getTemplate() --><!-- /ko -->
    </div>
</div>

<!-- START: add this code -->
<script type="text/x-magento-init">
{
    "*": {
        "Magento_Ui/js/core/app": {
            "components": {
                "
                    pulsestorm_nofrillslayout_chapter10_viewmodel
                    ": {
                    "component": "uiCollection",
                    "children":{
                        "child1":{
                            "component":"uiElement",
                            "template":"
                                Pulsestorm_Nofrillslayout/
```

```
                                    chapter10/child-one"
                            },
                            "child2":{
                                "component":"uiElement",
                                "template":"
                                    Pulsestorm_Nofrillslayout/
                                    chapter10/child-two"
                            }
                        }
                    }
                }
            }
        }
    }
}
</script>
```

and reload the page. You'll see each of the `child-one.html` and `child-two.html` templates have rendered.

Naming Child Elements

When we created our top level element, we gave it a unique-ish name.

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    templates/chapter10/user/main.phtml

"components": {
    "pulsestorm_nofrillslayout_chapter10_viewmodel": {
        "component": "Magento_Ui/js/lib/core/element/
            element",
        "helloMessage":"Hello View Model"
    }
}
```

While not required, by using our module namespace (`pulsestorm_nofrillslayout_`) name as part of the component name, we're (almost) guaranteed that no one else will attempt to create a component with our same name. This sort of "thrifty namespacing" is necessary when you're creating code that could be deployed into thousands of systems worldwide, or when you're using code from thousands of other developers without everyone being on the same page.

If you're already on board with that point of view, it may have surprised you to see us use the more generic names of `child1` and `child2`.

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    templates/chapter10/user/main.phtml

```

```
"children":{
    "child1":{
        "component":"uiElement",
        "template":"Pulsestorm_Nofrillslayout/chapter10/
            child-one"
    },
    "child2":{
        "component":"uiElement",
        "template":"Pulsestorm_Nofrillslayout/chapter10/
            child-two"
    }
}
```

Normally, you'd want to avoid un-namespaced names like this. However, a view model's name in Magento is its **full path** in the `x-magento-init/Magento_Ui/js/core/app` configuration.

In other words, the name of the view model with the template of `Pulsestorm_Nofrillslayout/chapter10/child-one` is **not** `child1`. It's actually `pulsestorm_nofrillslayout_chapter10_viewmodel.child1`, because `child1` is a child of `pulsestorm_nofrillslayout_chapter10_viewmodel`.

You can test this out for yourself if you're using the `uiRegistry` debugging technique we previously mentioned. Use the registry's `get` method to grab a view model by name.

```
//normally you wouldn't use requirejs directly like this
//but for debugging purposes it's safe.
var uiRegistry  = requirejs('uiRegistry');
var viewModel   = uiRegistry.get('
    pulsestorm_nofrillslayout_chapter10_viewmodel.child1');
console.log(viewModel);
```

Javascript Objects and Your Own View Models

We now have the the ability to instantiate and register a scoped Magento Knockout.js view model. We also know how to assign the view model data properties, and how to load a custom template from a `.html` file on the server. What we **don't** have is the ability to use our own custom objects as view models and create our own view model method with complex logic. In order to do that, we'll need to have a quick talk about object construction in javascript.

At this point in time, creating new objects in javascript is fraught with a lot of historical baggage. Creating a basic object in javascript is simple

```
var object = {};
```

However, javascript has an alternative syntax for creating objects. This syntax uses the `new` keyword.

```
1 var object = new SomeConstructorFunction();
```

Where `SomeConstructorFunction` is a javascript function. This function **definition** does not return a new object. Instead, when invoked with `new`, the function will assign properties to an object via javascript's magic `this` variable, and javascript will set those new properties on the returned object.

```
1 var SomeConstructorFunction = function(){
2     this.foo = "bar";
3 }
4
5 var object = new SomeConstructorFunction();
6 console.log(object.foo);
```

This function is sometimes called a "constructor function" – although javascript has no internal concept of a constructor function. Javascript only treats this function as special when you use it with the `new` keyword.

This form of object creation can be a sore point in certain javascript circles. Some developers recommend you not use it. While Knockout.js doesn't **require** you to use constructor functions, their default example

```
1 function AppViewModel() {
2     this.firstName = "Bert";
3     this.lastName = "Bertington";
4 }
5
6 // Activates knockout.js
7 ko.applyBindings(new AppViewModel());
```

does use one. Magento seems to have followed their lead, and built their entire front end object system around constructor functions.

Remember, when you tell Magento your *component* is a `Magento_Ui/js/lib/core/element/element` (or its alias, `uiElement`)

```
1 "pulsestorm_nofrillslayout_chapter10_viewmodel": {
2     "component": "Magento_Ui/js/lib/core/element/element",
3     "helloMessage":"Hello View Model"
4 }
```

What you're really saying to Magento is

> Hey Magento? Load the module at `Magento_Ui/js/lib/core/element/element`. This module should return a function. Then, create a new object with this function using the `new` keyword.

Or, in code

```
ConstructorFunction = requirejs('Magento_Ui/js/lib/core/
    element/element');
viewModel = new ConstructorFunction({
    "component": "Magento_Ui/js/lib/core/element/element",
    "helloMessage":"Hello View Model"
});
```

If you didn't follow everything we said about javascript constructors, don't worry. You don't need to understand these sort of things once you've learned the boilerplate. If you want a career in software engineering, these fundamentals are good to know. If you're just looking to get your job done, read on!

Creating Our Own View Models

From a high level, what we need to do is

1. Create a new RequireJS module that returns a `uiElement` constructor function
2. Add our view variables to the object created by that constructor function
3. Configure our `x-magento-init` script to use this new component/view-model.

We'll give our new RequireJS module the name of `Pulsestorm_Nofrillslayout/chapter10/user/view-model`. With this name, we'll need to create the following file in the following location.

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/web
    /chapter10/user/view-model.js
define(['uiElement'], function(UiElement){
    return UiElement;
});
```

Then, we need to configure our `x-magento-init` script to **use** this new RequireJS module as our `x-magento-init` script. Replace our `main.phtml` with the following

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    templates/chapter10/user/main.phtml
<h1>Hello View Model</h1>
<div data-bind="scope: '
    pulsestorm_nofrillslayout_chapter10_viewmodel'">
    <div>
        <!-- ko template: getTemplate() --><!-- /ko -->
    </div>
</div>
```

```
8
9  <!-- START: add this code -->
10 <script type="text/x-magento-init">
11 {
12     "*": {
13         "Magento_Ui/js/core/app": {
14             "components": {
15                 "
                       pulsestorm_nofrillslayout_chapter10_viewmodel
                       ": {
16                     "component": "Pulsestorm_Nofrillslayout
                           /chapter10/user/view-model",
17                     "template": "Pulsestorm_Nofrillslayout/
                           chapter10/hello-view-model"
18                 }
19             }
20         }
21     }
22 }
23 </script>
```

If you reload with the above in place, you should see the `Pulsestorm_Nofrillslayout/chapter10/user/view-model` view model rendered with the (provided by us) `Pulsestorm_Nofrillslayout/chapter10/hello-view-model` template.

```
1 Hello View Template
2
3 Value of helloWorld variable [].
```

What Just Happened

Let's start by throwing things into reverse. If we look at the `x-magento-init` portion of our code, our one big change comes here.

```
1 #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
      templates/chapter10/user/main.phtml
2 "components": {
3     "pulsestorm_nofrillslayout_chapter10_viewmodel": {
4         "component": "Pulsestorm_Nofrillslayout/chapter10/
              user/view-model",
5         "template": "Pulsestorm_Nofrillslayout/chapter10/
              hello-view-model"
6     }
7 }
```

Previously, this would have looked like the following

```
"components": {
    "pulsestorm_nofrillslayout_chapter10_viewmodel": {
        "component": "uiElement",
        "template": "Pulsestorm_Nofrillslayout/chapter10/
            hello-view-model"
    }
}
```

We've changed the `component` from a `uiElement` to a `Pulsestorm_Nofrillslayout/chapter10/user/view-model` module. This tells Magento to instantiate the view model named `pulsestorm_nofrillslayout_chapter10_viewmodel` from the constructor function returned by `Pulsestorm_Nofrillslayout/chapter10/user/view-model` **instead of** from the constructor function returned by `uiElement/Magento_Ui/js/lib/core/element/element`.

If we take a look at the constructor function that `Pulsestorm_Nofrillslayout/chapter10/user/view-model` returns

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/web
    /template/chapter10/user/view-model.js
define(['uiElement'], function(UiElement){
    return UiElement;
});
```

We see that all we've done is import the `uiElement` module via RequireJS, and then have our module return that module. i.e. – we've created a view model that behaves exactly the same as one with a `uiElement` configured component. If you want a view model with **new** behavior, you'll need to use the `UIElement`'s `extend` method (which we'll cover momentarily). Also, you probably noticed that this template line

```
Value of <code>helloWorld</code> variable [<span data-bind=
    "text:helloWorld"></span>].
```

rendered without any `helloWorld` value. Let's kill two birds with one stone by updating our RequireJS module to look like this

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/web
    /template/chapter10/user/view-model.js
define(['uiElement'], function(UiElement){
    return UiElement.extend({
        defaults:{
            helloWorld:"Hello World"
        }
    });
});
```

If you reload with the above `view-model.js` in place, you should see our view rendered with a value for the `helloWorld` variable.

```
Value of helloWorld variable [Hello World!].
```

Understanding the extend Method

The key to understanding why the above code worked is this block here.

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/web
    /template/chapter10/user/view-model.js

return UiElement.extend({
    defaults:{
        helloWorld:"Hello World"
    }
});
```

Previously, we were returning the `UiElement` variable. This variable contains a function. Specifically, a constructor function that the system will use to instantiate the `pulsestorm_nofrillslayout_chapter10_viewmodel` view model.

In javascript, functions are also objects. This means they can have methods. Above, we've called the `extend` method on the `UiElement` constructor function. This is a special, Magento provided method that allows you to create **new** constructor functions. The object you pass to `extend` allows you to add default properties and methods to your new objects.

If that didn't make sense, lets take a look by using our browser's javascript console. First, let's fetch the `uiElement` constructor function, and then look at its source code. This

```
> UiElement = requirejs('uiElement');
> console.log( UiElement.toString() );
```

should log something like this

```
function ()
{
    var obj = this;

    if (!_.isObject(obj) || Object.getPrototypeOf(obj) !==
        UiClass.prototype) {
        obj = Object.create(UiClass.prototype);
    }

    obj.initialize.apply(obj, arguments);
```

```
10
11      return obj;
12  }
```

When you (or more importantly when Magento or Knockout.js code) uses code like `new UiElement`, **this** is the constructor function that is called. Covering this object in full is beyond the scope of this chapter, but the previously mentioned uiElement series covers it well.

http://alanstorm.com/series/magento-2-uielement-internals/

Next, lets use the extend function to create a **new** constructor function. Run the following code from your javascript console.

```
1  > var OurConstructorFunction = UiElement.extend({
2      defaults:{
3          message:'Hello World!',
4      },
5      sayHello:function(){
6          console.log(this.message);
7      }
8  });
```

Above we've passed `extend` an object with a `defaults` key, and a `sayHello` key. Values in `defaults` will be assigned as *default values to newly constructed objects*, and the non-`defaults` keys will be assigned as methods to the new object.

In other words, if we use `OurConstructorFunction` to instantiate an object

```
1  > var object = new OurConstructorFunction;
```

then that object will have a `helloWorld` property

```
1  > console.log(object.message);
2  "Hello World"
```

and a `sayHello` method

```
1  > object.sayHello();
2  "Hello World"
```

If we return to our RequireJS module

```
1  #File: app/code/Pulsestorm/Nofrillslayout/view/frontend/web
       /template/chapter10/user/view-model.js
2  define(['uiElement'], function(UiElement){
3      return UiElement.extend({
4          defaults:{
5              helloWorld:"Hello World"
```

```
        }
    });
});
```

We can see our new view model constructor function

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/web
    /template/chapter10/user/view-model.js
define(['uiElement'], function(UiElement){
    return UiElement.extend({
        defaults:{
            helloWorld:"Hello World"
        }
    });
});
```

This view model constructor function will ensure a default value for the `helloWorld` property in our view model. If we wanted to add a method to our view model we could do the following to add a `getLines` method

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/web
    /template/chapter10/user/view-model.js
define(['uiElement'], function(UiElement){
    return UiElement.extend({
        defaults:{
            helloWorld:"Hello World"
        },
        getLines:function(){
            return ['You say yes', 'I say no', 'You say
                stop', 'I say go go go'];
        }
    });
});
```

and then we could invoke that method with template code similar to the following

```
<ul data-bind="foreach:getLines()">
    <li data-bind="text:$data"></li>
</ul>
```

We call `getLines` in the `foreach` binding, and then each time through the loop use the `data-bind="text:$data"` to output the value. The `$data` variable is a Knockout.js feature/convention, and contains the current value of the loop (similar to `$_` in perl).

Non-Default Values

Earlier, we covered the `defaults` property, which lets you set default values for your view models.

```
defaults:{
    helloWorld:"Hello World"
},
```

With the above configuration, **by default** an instantiated view model will have a `helloWorld` property with a value of `"Hello World"`. However, it is possible to change these default values. If we update our `x-magento-init` script to include a new value for `helloWorld`

```
#File: app/code/Pulsestorm/Nofrillslayout/view/frontend/
    templates/chapter10/user/main.phtml
{
    "*": {
        "Magento_Ui/js/core/app": {
            "components": {
                "
                    pulsestorm_nofrillslayout_chapter10_viewmodel
                    ": {
                    "component": "Pulsestorm_Nofrillslayout
                        /chapter10/user/view-model",
                    "template": "Pulsestorm_Nofrillslayout/
                        chapter10/hello-view-model",
                    "helloWorld":"Hola Muendo"
                }
            }
        }
    }
}
```

Magento will use this value instead. With the above `x-magento-init`, our template will render with the `Hola Muendo` value instead of hello world. This works because, behind the scenes, Magento's instantiating this object with javascript code that looks something like

```
var viewModel = new UiElement({
    "component": "Pulsestorm_Nofrillslayout/chapter10/user/
        view-model",
    "template": "Pulsestorm_Nofrillslayout/chapter10/user/
        hello-view-model",
    "helloWorld":"Hola Muendo"
});
```

There's also a second form of this that you'll see in Magento's core code, a top level `config` key

```
{
    "*": {
        "Magento_Ui/js/core/app": {
            "components": {
                "
                    pulsestorm_nofrillslayout_chapter10_viewmodel
                    ": {
                    "component": "Pulsestorm_Nofrillslayout
                        /chapter10/user/view-model",
                    "template": "Pulsestorm_Nofrillslayout/
                        chapter10/user/hello-view-model",
                    "helloWorld":"Hola Muendo",
                    "config":{
                        "helloWorld":"Bonjour le monde"
                    }
                }
            }
        }
    }
}
```

With the above code, the `helloWorld` value in the `config` object will override the `helloWorld` value set at the top level. If you're thinking this doesn't make a lot of sense – you're right. The reasons for this are complicated and myriad, and likely the result of one engineer responding to another engineer's choices. If you're writing `x-magento-init` blocks from scratch, we'd recommend staying away from this `config` syntax. However, you'll need to be aware of it if you're trying to debug how core cart functionality works in Magento 2.

Safely Using the uiRegistry

Earlier in this chapter, we used the `uiRegistry` to fetch previously instantiated view models.

```
var uiRegistry  = requirejs('uiRegistry');
var viewModel   = uiRegistry.get('
    pulsestorm_nofrillslayout_chapter10_viewmodel');
console.log(viewModel);
```

While this worked as a debugging exercise, relying on the `uiRegistry`'s `get` method in production code isn't a great idea. Magento's system makes no promises about the order it will instantiate view models in. When you combine

that with the asynchronous nature of RequireJS module loading, that means there's no way to know for sure when a particular view model will be available.

Put another way, just because

```
uiRegistry.get('
    pulsestorm_nofrillslayout_chapter10_viewmodel');
```

works in your dev enviornment doesn't mean it will work in a production enviornment. Fortunately, the `uiRegistry` object has a solution – the `get` method also supports a callback syntax. For example, consider the following program

```
requirejs(['uiRegistry'], function(uiRegistry){
    uiRegistry.get('someUniqueKey', function(
        theRegistryValue){
          console.log("Called the async callback");
          console.log(theRegistryValue);
    });
});
```

Here we're attempting to fetch the `uiRegistry`'s `someUniqueKey` value using `get`, with a javascript function as the second argument to get. Running this program results in **no** output to the console. That's because there's no value set for `someUniqueKey`. However, if (in the same console window) we run the following program that **sets** a value

```
requirejs(['uiRegistry'], function(uiRegistry){
    uiRegistry.set('someUniqueKey', 'The Value');
    console.log("Finished");
});
```

we should see the following in our console.

```
Finished
Called the async callback
The Value
```

That is – Magento invoked the callback function after we set a value. The callback may not be immediately invoked (i.e. our *Finished* message shows up first), but by using `get` with a callback, we can be certain our code will run **only** when there's a value available in the registry.

Wrap Up

Pretty intense, right? After carefully leading you through nine chapters on layout XML, we feel a little guilty dropping a gigantic javascript anvil on your head.

This chapter only touches the surface of how Magento's new, modern javascript features work. When you're ready to continue, we strongly recommend taking a look at the three, free, online series on Magento 2's javascript functionality that we mentioned earlier

Magento 2: Advanced Javascript

https://alanstorm.com/category/magento-2/#magento2-advanced-javascript

Magento 2 UI Components

https://alanstorm.com/category/magento-2/#magento-2-ui

Magento 2: uiElement Internals

https://alanstorm.com/category/magento-2/#magento-2-uielement-internals

While newer greenfield front end technologies (PWA, vue-storefront, etc.) are making their way to the Magento platform, these Knockout.js based UI Components are what ships with stock Magento. Understanding them will be just as important as understanding Magento's layout based XML systems for anyone with more than a fleeting interest in Magento development.

Chapter 12

Appendix

Magento 2 Areas

Areas were a tricky concept to understand in Magento 1, and remain so in Magento 2. To understand areas, you need to consider **how** web applications are typically built. Consider

1. A URL
2. A web application that lives at that URL
3. A web application that has several distinct features living at sub-URLs
4. And those features each require a different set of assets and resources

Magento's area system uses the current URL to identify which "area" of an application a user is in, and uses that area to load different assets and resources. For example, when you're on a site's homepage, or on the cart checkout page, your URL might look like the following.

```
1 http://magento.example.com/
2 http://magento.example.com/checkout
```

These are URLs in the `frontend` area. However, when you're looking at the back office

```
1 http://magento.example.com/admin/admin/dashboard/index
```

you're in a different area named `adminhtml`.

Areas are important to front end developers because each area loads a *different* set of layout handle XML files, and a different set of front end asset files. For example, in the catalog module you can find the `phtml` files for the `frontend` cart application in the following folder

```
1 //notice the "frontend" path portion
2
3 vendor/magento/module-catalog/view/frontend/templates/
```

while the files for the back office `adminhtml` area are in a different folder.

```
1 vendor/magento/module-catalog/view/adminhtml/templates/
```

When Magento need to load a template URN like the following

```
1 Magento_Catalog::path/to/template.phtml
```

the base folder it uses will be determined by the application's current area.

Base Area Folders

One problem with the area system in Magento 1 was the inability to easily share assets between two areas. For example, as an extension or theme developer you might want to share a `phtml` template with both a back office `adminhtml` page **and** and `frontend` cart page.

To solve this problem, Magento 2 introduced the idea of a `base` area. Assets placed in the base folder are available to **both** areas.

For example, if Magento tries to expand the `Magento_Catalog::path/to/template.phtml` URN out to a full file path in the `frontend` cart application, Magento will check for a file in the following location first

```
1 vendor/magento/module-catalog/view/frontend/templates/path/
      to/template.phtml
```

If it doesn't find a file there, Magento will check the the `base` area folder next.

```
1 vendor/magento/module-catalog/view/base/templates/path/to/
      template.phtml
```

Programmatically Determining the Area

Another way Magento 2 improves the area system is a more straightforward mechanism for programmatically checking the current area code. The `Magento\Framework\App\State` single instance object keeps track of the area code. If you use this class in an automatic constructor dependency injection constructor, you can easily check the current area

```
public function __construct(
    \Magento\Framework\App\State $appState
)
{
    $this->appState = $appState;
}
/* ... */
public function someMethod()
{
    var_dump(
        $this->appState->getAreaCode()
    );
}
```

Each sub-system in Magento handles areas a little bit differently. We'll cover the specifics as we come across them – this appendix was meant as a gentle introduction to the topic, and as a way to cover changes to the area system in Magento 2.

PHP Autoloading

Autoloading is the PHP system that ensures when a developer attempts to instantiate an object from a class, that the correct class definition file is loaded.

All programming languages have some sort of mechanism for this – what makes PHP distinct is the autoloading system is end-user-programmer configurable. That is to say, anyone writing PHP code can register an autoloader that might run when another programmer attempts to instantiate a class.

Taking the long historic view, this has led to autoloaders from early PHP frameworks not playing nice with each other when used in the same program. An autoloader from framework A might fail when a programmer attempts to load a class from framework B.

In recent years, the PHP community has standardized its autoloading mechanism using the PSR-0 and PSR-4 standards. These standards were adopted by Composer, which means most modern Composer distributed frameworks (Magento 2 included) use the PSR based Composer autoloaders.

However, as a working PHP programmer, it's still necessary to understand the mechanics of autoloading. Older PHP frameworks still use older, non PSR based autoloaders in their Composer packages. Additionally – Composer's autoloader API offers an irresistible temptation for meta-programmers who want to hook into class instantiation in PHP based systems.

This appendix will describe Magento's PSR-4 based autoloaders. It will also describe a few ways in which Magento uses the autoloader system to implement

functionality unrelated to class autoloading.

PSR-4 Autoloading

A PSR-4 autoloader allows a user to configure a class prefix with a path name. When PHP needs to instantiate classes with this prefix, the autoloader will look for class files in the configured folder. The autoloader uses the non prefixed portion of the class to build the path's full class.

Here's an example. A user might configure the class prefix `Foo\Bar` with the path `path/to/src`. Then, if someone tried to instantiate a class like this

```
1  $object = new \Foo\Baz\Bar\Bap;
```

The PSR autoloader would look for the class's definition file in

```
1  path/to/class/Bar/Bap.php
```

The autoloaders would not try to load a class named `Baz\Bar\Foo`, because this class does not start with the configured prefix `Foo\Bar`.

A PSR-4 autoloader creates the path to the class file by starting with the configured path name, and transforming the non-prefix part of the class name (`Bar\Bap`). This transformation includes turning the namespace separator characters into file path separators, and appending a `.php` file extension.

There's some additional subtleties and rules to PSR-4 autoloading. If you're interested in learning more you should checkout the PSR-4 specification

http://www.php-fig.org/psr/psr-4/

PSR-4 Autoloading in Composer and Magento

Here's how PSR-4 autoloading works in Magento 2 with Composer. In most cases, a Magento 2 module will configure a PSR-4 autoloader in its composer.json file. For example, consider the `Magento_Catalog` module

```
1  #File: vendor/magento/module-catalog/composer.json
2
3  "autoload": {
4      /* ... */,
5      "psr-4": {
6          "Magento\\Catalog\\": ""
7      }
8  }
```

As you can see above, Magento's configured a PSR-4 autoloader with a class prefix of `Magento\Catalog`, and an empty path. This is a standard Composer PSR-4

autoloader configuration. The empty path setting means *from this Composer package's root folder*. In other words, Magento 2 will load the `Magento\Catalog\Model\Product` class from

```
vendor/magento/module-catalog/Model/Product.php
```

This is the pattern Magento follows for all its core modules. Third party developers are not required to follow this pattern, but it's strongly recommended they do so.

Autoloader Module Registration

Earlier we mentioned some frameworks, Magento included, use the autoloader and autoloader related systems to implement other functionality.

Magento needs a way to know which modules are available in a system. Rather than keep a list of modules somewhere in application state, or scan common module directories on every system bootstrap, Magento uses registration.php files. These files are located in the root of each Magento module folder (as well as theme, library, and language pack folders)

Rather than perform a search for these files on every bootstrap, Magento taps into the power of Composer's `files` autoload feature. Consider, again, the `Magento_Catalog` module.

```
#File: vendor/magento/module-catalog/composer.json

"autoload": {
    "files": [
        "registration.php"
    ],
    /* ... */
}
```

The original intent of the `files` autoloader is that developers would use a file autoload to manually setup PHP autoloading for the framework with `__autoload` or `spl_autoload_register`, or that they would simply list the class files that should be automatically loaded here.

However, because PHP loads a file configured in the `files` autoloader on every PHP request, this is a tempting place for framework authors to put other initializations and registration scripts. Magento could not resist this temptation themselves – if you look inside a registration.php file

```
#File: vendor/magento/module-catalog/registration.php
<?php
/**
 * Copyright © Magento, Inc. All rights reserved.
```

```
 * See COPYING.txt for license details.
 */

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

You'll see the code a module developer writes to let Magento know their module's name, and that the module is available for the system to use.

Code Generation Autoloader

The other meta-programming autoloader Magento uses is their code generation autoloader. Magento 2's programming style requires many classes, and many of those classes have boilerplate code that's the same thing over and over again. Rather than force Magento developers to manually create these classes, Magento will automatically create them for us whenever it sees them mentioned. For example, if you mention a Factory class in a constructor

```
public function __construct(SomeObjectFactory $factory)
{
    /* ... */
}
```

and that constructor class does not exist already, Magento will automatically create a class with a base factory implementation for you in either the `var/generation/` or (in Magento 2.2+) `generation/` folders

The "whenever it sees them" line probably piqued your engineering brain. In order to implement this functionality, Magento registers a **custom** autoloader named `Magento\Framework\Code\Generator\Autoloader::load`. Its implementation is beyond the scope of this appendix, but Magento registers this autoloader here

```
#File: vendor/magento/framework/ObjectManager/
    DefinitionFactory.php
public function createClassDefinition()
{
    $autoloader = new Autoloader($this->getCodeGenerator())
        ;
    spl_autoload_register([$autoloader, 'load']);
    return new Runtime();
}
```

and you can start following the autoloader's execution here

```
#File: vendor/magento/framework/Code/Generator/Autoloader.
    php
public function load($className)
{
    if (!class_exists($className)) {
        return Generator::GENERATION_ERROR != $this->
            _generator->generateClass($className);
    }
    return true;
}
```

There's a number of different file types Magento will generate for you with this autoloader

```
#File: vendor/magento/magento2-base/app/etc/di.xml
<argument name="generatedEntities" xsi:type="array">
    <item name="factory" xsi:type="string">\Magento\
        Framework\ObjectManager\Code\Generator\Factory</
        item>
    <item name="proxy" xsi:type="string">\Magento\Framework
        \ObjectManager\Code\Generator\Proxy</item>
    <item name="interceptor" xsi:type="string">\Magento\
        Framework\Interception\Code\Generator\Interceptor</
        item>
    <item name="logger" xsi:type="string">\Magento\
        Framework\ObjectManager\Profiler\Code\Generator\
        Logger</item>
    <item name="mapper" xsi:type="string">\Magento\
        Framework\Api\Code\Generator\Mapper</item>
    <item name="persistor" xsi:type="string">\Magento\
        Framework\ObjectManager\Code\Generator\Persistor</
        item>
    <item name="repository" xsi:type="string">\Magento\
        Framework\ObjectManager\Code\Generator\Repository</
        item>
    <item name="convertor" xsi:type="string">\Magento\
        Framework\ObjectManager\Code\Generator\Converter</
        item>
    <item name="searchResults" xsi:type="string">\Magento\
        Framework\Api\Code\Generator\SearchResults</item>
    <item name="extensionInterface" xsi:type="string">\
        Magento\Framework\Api\Code\Generator\
        ExtensionAttributesInterfaceGenerator</item>
    <item name="extension" xsi:type="string">\Magento\
        Framework\Api\Code\Generator\
        ExtensionAttributesGenerator</item>
</argument>
```

However, be careful. Just because Magento shipped a code generating autoloader doesn't mean that the autoloader is ready for the outside world (or even the inside world) to use.

The Magento Cache

The concept of a *cache* in programming is a relatively simple one. However, as time goes on, the *practice* of caching has become more and more complicated.

Caching is the process of

1. Doing something computationally intensive or time consuming.
2. Saving the result.
3. Using the saved result the next time you need that same thing.

Caching is a popular solution for performance problems because it allows developers to solve their problem without worrying about performance up-front. In isolation, this may seem lazy. However, considered in the life of a working programmer who is writing new algorithms everyday, and who never knows which ones need to be performant and which ones do not, caching is often the only solution that makes sense. Deciding what to cache, when, and for how long is a fundamental part of modern systems development.

Magento 1 featured an infamous number of caches. Magento 2 has kept all of these, and added a few new layers on top.

You can see identifiers (also known as *tags*) for each of Magento's cache types by using the `cache:status` CLI command

```
$ php bin/magento cache:status
                     config: 1
                     layout: 1
                  block_html: 1
                 collections: 1
                  reflection: 1
                      db_ddl: 1
                         eav: 1
      customer_notification: 1
                   full_page: 0
          config_integration: 1
      config_integration_api: 1
                  translate: 1
           config_webservice: 1
```

Covering what each of these cache types does is beyond the scope of this book. We will, however, briefly cover a few front end related caches. We'll also cover some things not on this list, but that are cache-like systems you'll need to be

aware of when working with Magento 2. Finally we'll cover clearing these caches via formal, and informal, means.

layout

[layout]

The `layout` cache stores pre-built XML configuration trees for *layout related XML files*. This means files in your module and theme's `view` folder, and **not** files in your `etc` folder.

block_html

[block_html]

Most `phtml` templates in Magento have a corresponding PHP Block class. A block class author can, if they wish, configure these blocks to be cached. That is, they'll render once using `phtml` code and block class methods, but the next render will simply pull the previous output from cache.

Magento's category navigation block is a prominent example of this – as querying a large category list is often a computationally complex action.

full_page

[full_page]

The full page cache is a new cache type introduced in Magento 2. The full page cache will cache the results of individual Magento page requests to disk. This means with full page caching enabled, the first request for a Magento page will use Magento's controllers and layout system to build the page's output and save the page to cache before serving it. The next request will skip these subsystems, opting to serve the page from the cache.

This feature existed in Magento 1's enterprise versions, and Magento 1 CE had many third party full page caching extensions available. Because this feature ships in all versions of Magento 2, module and theme developers will need to test their feature both with full page caching enabled, and full page caching disabled.

Although this cache type is grouped with Magento's others, and you're able to use standard cache commands to manage (i.e. clear) the full page cache, Magento **does not** store these cached pages in the standard cache storage. Instead, full page cache files are (by default) stored in the

```
1 var/page_cache
```

folder. Finally, it's worth noting that some Magento team members have stated the stock full page cache behavior is intended as a demo, and that it shouldn't be used in production. Regardless of how you decide to deploy, you'll want to be aware of full page cache while you're developing or distributing code to developers who may run with this mode enabled.

Full Page Cache with Varnish

By default, Magento's full page cache stores its cached pages on disk, in the `var/page_cache` folder. However, it's possible to configure Magento to use varnish for its full page cache. Varnish is a web server proxy whose sole job is to save pre-rendered HTTP responses (HTML pages, XML files, JSON files, etc.)

You can "turn on" varnish by navigating to

```
1 Stores -> Configuration -> Advanced -> Full Page Cache
```

Magento's varnish implementation will generate a varnish VCL file for you to use with your varnish system. It will not setup or run varnish automatically for you. Magento 1 had no official varnish support, but if you've ever used the Magento 1 Nexcess Turpentine extension you'll feel right at home. Per our previous comments on Magento not recommending full page caching on disk (the default behavior) for production sites, Magento **does** recommend you run your stores behind varnish proxy servers.

If you're interested in running varnish and don't know how, you'll need to check with your server infrastructure team/partners.

LessCSS Caches

Magento uses a front end tool called LessCSS to generate its cross browser CSS files. On the fly rendering of LessCSS files is another place where you can benefit from a caching system.

Magento stores its LessCSS caches in

```
1 var/view_preprocessed
```

If you're making changes to LessCSS files, you may need to clear out this folder to see your changes.

There's no `bin/magento` command to clear out these LessCSS caches. However, Magento does ship with grunt tasks that will clear out *some* of the LessCSS caches. See the Front End Build System appendix for more details.

Front End Files (Static Assets)

Magento's front end asset files (CSS, JavaScript, etc.) are, for the most part, generated on the fly. As previously mentioned, CSS files are generated by Less-CSS. JavaScript files, while not generated, are stored with Magento's modules and moved *on demand* to the `pub/static` folder. By on demand, we mean as a user requests them, or when you deploy your Magento system (using the `setup:static-content:deploy` command).

Other Cache Entries

Keep in mind that Magento's caching objects are available for third party extension developers to use however they want. Even if you clear **all** the identifiers listed above, another developer may have tagged a cache entry with their own cache tags and not one of the 13 Magento cache types. For these cases, Magento offers a Flush Cache Storage option, available at

```
1 System -> Tools -> Cache Management
```

or

```
1 $ php bin/magento cache:flush
```

This will clear Magento's primary cache storage engine of all entries.

Notes on Clearing Cache

Throughout this book, we'll try to remind you when you'll need to clear your Magento cache to see your changes show up. Magento's main cache storage is located (by default) in the

```
1 var/cache
```

folder. You can clear your cache by removing this entire folder. However – it's also possible (and recommended) to configure Magento to use an in-memory caching solution like redis or memcache. Because of this, it's probably best to develop the habit of clearing your cache using the built in Magento command line command

```
1  $ bin/magento cache:clean
```

This will run through each of Magento's main cache types and clear them out.

Another caveat – if your full page cache engine is varnish, or another third party full page cache implementation, the above command may not clear this

cache. Also, as previously mentioned, the LessCSS files and front end assets only behave like cached entries, they're not an official part of the Magento cache.

Whenever you're facing a problem where it *seems* like you've edited the right file, but your changes aren't showing up, it's always worth reviewing which cache and cache-like systems may be in play, and take the additional steps of

1. Clearing those caches using the official tools
2. Moving on to clearing the storage directly (removing cache folder/files, deleting your redis key entries, etc.)

The Command Line

The "command line" is one of the oldest ways to run programs and communicate with your computer. Despite being a relic of another time, the command line remains popular for developers and other technical professionals who need programs that take clear inputs, and produce clear outputs.

If you're unfamiliar with the command line, you can access it via your Linux or MacOS/OS X based computer by running a terminal program (included with the OS). Once you launch this program, the command line will just sit there, waiting for you to run a command

```
1 $
```

Running a command line environment on a Windows computer is a slightly more complicated affair – the windows command line is different in many ways from a unix/linux environment. Windows also offers the ability to connect to a *nix based computer or virtual machine and run command line programs. We're going to assume you windows folks know what you're doing, or are smart enough to figure it out. i.e. this appendix will assume a *nix environment.

Running a Command Line Program

To run a command line program, all you need to do is type that program's name, and hit enter. For example, the `ls` program will list a directory's current contents

```
1 $ ls
2 ...
```

Command line programs can also, like functions in programming languages, accept arguments. For example, the `cd` command (which changes directories) accepts a single argument (the folder/directory you want to change to)

```
$ cd /path/to/magento
```

Command line programs also often accept options – for example you can use the `-1` option with `ls` to get a single column listing of files

```
$ ls -1
```

Typically, you'll use a single dash is for one letter options, and two dashes for a full-word options. i.e.

```
$ curl --progress-bar http://example.com
```

The single letter options are a remnant of computer past, where every byte of memory was important.

Magento CLI

Many application frameworks include their own command line framework. These frameworks allow a programmer to create command line scripts in the language of the application framework and with easy access to the functions and methods (or, in today's vague parlance, the "API") of the application framework itself.

Magento 2 includes a command line framework, and ships with many useful commands. To see a list of commands in the Magento command line framework, just run the following from your project's root folder

```
$ php bin/magento list
...
```

This command may require some explaining. First, the `php` portion is just us running the command line version of PHP. The command line version of PHP accepts an argument that's a path to the file to execute via php – in this case, `bin/magento`. If you look in the `bin/magento` file

```
#File: bin/magento

#!/usr/bin/env php
<?php
/**
 * Copyright © Magento, Inc. All rights reserved.
 * See COPYING.txt for license details.
 */

if (PHP_SAPI !== 'cli') {
    echo 'bin/magento must be run as a CLI application';
```

```
12      exit(1);
13  }
14
15  try {
16      require __DIR__ . '/../app/bootstrap.php';
17  } catch (\Exception $e) {
18      echo 'Autoload error: ' . $e->getMessage();
19      exit(1);
20  }
21  try {
22      $handler = new \Magento\Framework\App\ErrorHandler();
23      set_error_handler([$handler, 'handler']);
24      $application = new Magento\Framework\Console\Cli('
            Magento CLI');
25      $application->run();
26  } catch (\Exception $e) {
27      while ($e) {
28          echo $e->getMessage();
29          echo $e->getTraceAsString();
30          echo "\n\n";
31          $e = $e->getPrevious();
32      }
33      exit(Magento\Framework\Console\Cli::RETURN_FAILURE);
34  }
```

Here we see the PHP code that bootstraps Magento's command line framework. The second argument to `php` (above, `list`) is *also* the first argument to the `bin/magento` command line script. Many command line programs use this "sub-command" pattern. For example, the git version control system has many sub-commands

```
1  $ git add ...
2  $ git status
3  //etc...
```

When you run the `list` command, you're telling Magento's command line framework to list out all the commands available

Describing each Magento command in full is well beyond the scope of this appendix (or this book, or possibly any single book). However, here's an example of a command that will clear Magento's cache

```
1  $ php bin/magento cache:clean
```

And here's another command that will enable a module

```
1  $ php bin/magento module:enable Foo_Bar
```

You'll notice the `module:enable` command also accepts an argument – `Foo_Bar`. It's arguments all the way down! You can see an example of the arguments, and options, supported by each command by using the `help` command

```
$ php bin/magento help module:enable
```

BE CAREFUL: Command line scripts are incredibly powerful, but they often come without guard rails. In addition, Magento often ships command line scripts that aren't as well tested as more common application code paths. A bug in a command line script or a misuse of a bug-free script could damage your system in a way that might take hours, or even days, to fix. Don't use a command line script that you don't understand (or that you don't find in a tutorial you trust), and it's always a good idea to test out a command that's new to you in a backed up development environment.

Finally, as mentioned, these commands are just PHP code. For example, you can find the source to the `module:enable` command here

```
vendor/magento/magento2-base/setup/src/Magento/Setup/
    Console/Command/ModuleEnableCommand.php
```

Magento 2 Components

In Magento 1, the idea of modules, themes, libraries, and language packs all got a little blurry around the edges. Magento 2 attempts to make these distinctions a bit clearer. Specifically, while Magento 2 still has code modules, themes, code libraries, and language packs, there's a higher level idea of a Magento Component that sits above all of them.

What is a Component

To start, a "naming things is hard" note. Magento Components and UI Components are two different things. We're talking about Magento Components here, so if you know anything about the UI Component system, check that knowledge at the door. Don't worry, it'll be there when you get back.

A Magento Component is, simply speaking, a logical collection of files in a folder hierarchy. A Magento Component is identified by a `registration.php` file in the top level folder of this hierarchy.

For example, consider the following file

```
#File: vendor/magento/theme-frontend-blank/registration.php
\Magento\Framework\Component\ComponentRegistrar::register(
    \Magento\Framework\Component\ComponentRegistrar::THEME,
```

```
4       'frontend/Magento/blank',
5       __DIR__
6   );
```

The files in `vendor/magento/theme-frontend-blank` are part of the blank theme that ships with Magento 2. This `registration.php` identifies the files in this folder as part of this theme. Each `registration.php` contains a call to the static `Magento\Framework\Component\ComponentRegistrar::register` method which

1. Identifies the Component type
2. Provides an identifier for the Component
3. Provides a path to the Component's files

So, above, the Component is identified as a theme via the class constant

```
1   \Magento\Framework\Component\ComponentRegistrar::THEME
```

Its identifying string is `frontend/Magento/blank`, and the code uses the `__DIR__` magic constant to identify the current directory as the folder that contains this Component's files. Regarding `__DIR__`, while it's technically possible to point to another folder, the Magento convention is to always use this constant. Veering from this may result in unexpected behavior.

A Module Component

Let's look at another Component to make sure we understand `registration.php`. Consider the following file

```
1   #File: vendor/magento/module-catalog/registration.php
2   \Magento\Framework\Component\ComponentRegistrar::register(
3       \Magento\Framework\Component\ComponentRegistrar::MODULE
            ,
4       'Magento_Catalog',
5       __DIR__
6   );
```

This folder contains files for the catalog code module. We know this Component is a module because of the following class constant.

```
1   \Magento\Framework\Component\ComponentRegistrar::MODULE
```

This module's identifier is `Magento_Catalog`, and (as recommended) the location of its files are the current directory, indicated via `__DIR__`.

You can see an example of a code library here

```
#File: vendor/magento/framework/registration.php
\Magento\Framework\Component\ComponentRegistrar::register(
    \Magento\Framework\Component\ComponentRegistrar::
        LIBRARY,
    'magento/framework',
    __DIR__
);
```

and a language/translation pack here

```
#File: vendor/magento/language-de_de/registration.php
\Magento\Framework\Component\ComponentRegistrar::register(
    \Magento\Framework\Component\ComponentRegistrar::
        LANGUAGE,
    'magento_de_de',
    __DIR__
);
```

Magento treats the files in each of these folders differently – but they're all Magento Components.

How Magento loads Components

This, of course, leads to the $64,000 question: How does Magento see, or load, these Components. In Magento 1, modules were loaded via files in

```
app/etc/modules
```

and themes were loaded by editing values in the `System -> Configuration -> Design` sections and added to via layout XML files configured in a module's `config.xml` file. Magento 1's single code library lived in `lib/`, and Magento scanned for language packs in a hard coded folder. Like we said – things get blurry at the edges.

One of the goals of the Magento Component system is to remove that blurriness. Another is to enable distribution of these Components via PHP Composer. If you downloaded Magento 2 as an archive, you may wonder why most of its code files are located in the `vendor/magento` folder. The `vendor` folder is where Composer installs files to by default.

So, if Magento's code files are (mostly) in `vendor` – how does Magento see them? That happens thanks to the magic of Composer's "file autoload" feature.

1. Each Composer package has a composer.json file
2. One of the things this file lets you do is run a small PHP script on every page load for a Composer based project

3. The original intent of this script was to allow packages to setup their own PHP Autoloader (via PHP's `spl_autoload_register` function)
4. Magento has hijacked this functionality to register Components

If you're interested in learning more about autoloading, the autoloading appendix is a good place to start, and this online article can provide you with an in-depth exploration of Composer's autoloading mechanisms

http://alanstorm.com/laravel_composer_autoloading/

Let's take another look at the catalog module – but this time, we'll look at its `composer.json` file.

```
1  #File: vendor/magento/module-catalog/composer.json
2  {
3      /* ... */
4      "autoload": {
5          "files": [
6              "registration.php"
7          ],
8          /* ... */
9      }
10 }
```

We've highlighted the section we're interested in. This `autoload.files` configuration tells Composer to load the `registration.php` files located in the root folder of this module. This is the same `registration.php` we viewed earlier.

By creating their `composer.json` files like this, Magento has ensured that as soon as Composer's autoloader finishes loading, that PHP will have called `Magento\Framework\Component\ComponentRegistrar::register` for all the Components that are currently installed via Composer.

While clever, and powerful, this does mean that Magento 2 is tied to Composer in a way that other PHP projects with traditional architectures aren't.

Non-Composer Components

This also raises a few important questions – is it possible to install Magento Components **without** relying on Composer? Can we work on projects without needing to redeploy code to a Composer package every time we want to see a change?

Fortunately, the answer to both is yes. In addition to the above `composer.json` mechanism, Magento will scan certain folders for `registration.php` files, and load those as well. This scanning happens in the following file

```
1  #File: app/etc/NonComposerComponentRegistration.php
2  <?php
```

```
3  $pathList[] = dirname(__DIR__) . '/code/*/*/cli_commands.
       php';
4  $pathList[] = dirname(__DIR__) . '/code/*/*/registration.
       php';
5  $pathList[] = dirname(__DIR__) . '/design/*/*/*/
       registration.php';
6  $pathList[] = dirname(__DIR__) . '/i18n/*/*/registration.
       php';
7  $pathList[] = dirname(dirname(__DIR__)) . '/lib/internal
       /*/*/registration.php';
8  $pathList[] = dirname(dirname(__DIR__)) . '/lib/internal
       /*/*/*/registration.php';
9  foreach ($pathList as $path) {
10     // Sorting is disabled intentionally for performance
           improvement
11     $files = glob($path, GLOB_NOSORT);
12     if ($files === false) {
13         throw new \RuntimeException('glob() returned error
               while searching in \'' . $path . '\'');
14     }
15     foreach ($files as $file) {
16         include $file;
17     }
18 }
```

If you're having trouble reading that, Magento will search for `registration.php` files at

```
1  app/code/[ANYFOLDER]/[ANYFOLDER]/registration.php
2  app/design/[ANYFOLDER]/[ANYFOLDER]/[ANYFOLDER]/registration
       .php
3  app/i18n/[ANYFOLDER]/[ANYFOLDER]/registration.php
4  lib/internal/[ANYFOLDER]/[ANYFOLDER]/registration.php
5  lib/internal/[ANYFOLDER]/[ANYFOLDER]/[ANYFOLDER]/
       registration.php
```

Where `[ANYFOLDER]` is literally any folder. These paths are included partially for backwards compatibility, but also for a developer's convenience. For example, because Magento still scans `app/code` for `registration.php` files, we can have you drop the sample code included with this book into this folder and have it work **without** the need for setting up a Composer module.

Magento 1 developers will want to be careful: There's a few quirks to the system that make these file path based Components behave a little differently than their Magento 1 counter parts.

For example – it's the constant in `registration.php` that controls the type of Component you're loading. This means it's technically possible (although not recommended) to include a theme in `app/code`.

```
app/code/base/default/registration.php
app/code/base/default/...other theme files...
```

or a module in `app/design`

```
app/design/somefolder/Package/Module/registration.php
app/design/somefolder/Package/Module/...other module files
    ...
```

Also – if you're in the habit of renaming folders to `.bak`, or something similar, as a quick backup/testing mechanism, you're in for a surprise. Magento will still scan the following folder and try to load a module

```
app/code/Packagename/Modulename.bak/registration.php
```

Also, don't forget that a module or theme's name comes from the `registration.php` file and **not** the file path. It's theoretically possible to setup a `registration.php` file where the two don't match.

```
#File: app/code/Packagename/Modulename/registration.php

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

While this might "work" – it will likely lead to confusion and *may* cause unexpected bugs. If you're going to use the `app/code` folders it's best to stay within the guide rails of the system (unless you enjoy deep code safari's – in which case, have fun!)

Downloading URLs with curl

Every web page on the internet has a URL

```
http://example.com/hello.html
```

In modern times, URLs often resolve to data documents like XML or JSON files.

```
http://magento.example.com/foo.json
http://magento.example.com/foo.xml
```

Some of these may be actual documents on a server. Others may be generated on the fly by a software application (like Magento). One of the beauties of the internet is web browsers don't care how a document gets made – as long as the server speaks the *HyperText Transfer Protocol* (or HTTP), web browsers are happy to grab the document or data from the server.

As a human being, you use a program like Internet Explorer, Chrome, Firefox, Opera, Netscape, etc. to browse the web. These programs fetch the documents for you.

As a programmer, it's often useful to make an HTTP request **without** a web browser. Browsers have many layers of caching that happen automatically, some of which are hard to turn off. Web browsers also try to render documents as HTML by default, and sometimes you want to see the *raw data* returned by the server. Other times, you may be writing a software application that isn't a web browser, but still needs to use pages and data on the web. Regardless of your motivations, there are many libraries and programs for doing this. One of the most popular and longest standing is a program and library called `curl`.

The `curl` command line program is available for all popular operating systems, and many unpopular ones. It's based on a C library named `libcurl` that programmers embed in their own C based software applications. Chances are you use something that uses `libcurl` everyday.

This book will occasionally use the `curl` command line program to fetch a URL. This appendix is a brief and incomplete tutorial on to use `curl`.

Fetching URLs with CURL

In its simplest form, the `curl` command accepts a single argument. This argument is a URL. When you pass curl a single url, it will use the HTTP protocol to fetch a response from that URL

```
1  $ curl https://alanstorm.com/
2  <!DOCTYPE html>
3  <!--[if IE 7]>
4  <html class="ie ie7" lang="en-US">
5  <![endif]-->
6  <!--[if IE 8]>
7  <html class="ie ie8" lang="en-US">
8  //...
9
10 $ curl https://api.twitter.com/1/statuses/alanstorm.json
11 {
12     "errors": [{
13         "message": "The Twitter REST API v1 is no longer
               active. Please migrate to API v1.1. https://dev
               .twitter.com/docs/api/1.1/overview.",
14         "code": 64
```

```
15      }]
16  }
```

The results of a `curl` call are sent to the terminal's STDOUT stream. i.e. they're shown as output to the terminal. If you want to save these results to a file, you can use standard Unix redirect tools

```
1  $ curl https://alanstorm.com/ > file.html
2  $ curl https://api.twitter.com/1/statuses/alanstorm.json >
       file.json
```

You can also tell `curl` to save a file using the servers file name with the `O` option.

```
1  $ curl -O https://api.twitter.com/1/statuses/alanstorm.json
2  $ cat alanstorm.json
```

Viewing HTTP Headers with CURL

Sometimes you may be surprised to find `curl` returns an empty or unexpected response for a URL that works in the browser

```
1  $ curl http://alanstorm.com
2  <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
3  <html><head>
4  <title>302 Found</title>
5  </head><body>
6  <h1>Found</h1>
7  <p>The document has moved <a href="https://alanstorm.com/">
       here</a>.</p>
```

One reason for this **may** be that the server sent an HTTP Location redirect. You can test for this by looking at the HTTP headers for a response with the `-I` option.

```
1  $ curl -I http://alanstorm.com
2  HTTP/1.1 302 Found
3  Date: Thu, 29 Mar 2018 00:42:28 GMT
4  Server: Apache/2.4.29
5  Location: https://alanstorm.com/
6  Content-Type: text/html; charset=iso-8859-1
```

Ah ha! We did get a `Location`: header redirect. This is an example of curl being very simple and literal. It made the HTTP request, and showed you the response. There are times where this is exactly the behavior you want. However, sometimes you just want curl to follow the redirects automatically. In those cases, use the `L` option

```
$ curl -L http://alanstorm.com
//...
```

Another important header trick: Use the `curl -i` option (lowercase) to fetch both the headers AND the response.

The `curl` command has almost 200 options at its disposal. With these options, you'll be able to recreate almost any web request you need. We're not going to cover them all, but the internet is filled with all sorts of `curl` examples. Whether you're looking to set custom request headers, want to read and write cookies, or plumbing the depths of what HTTP can do, `curl` can probably do it for you.

Magento 2 Dependency Injection

There's no way around this: If you're going to work with Magento 2, you need to have a high level understanding of dependency injection in order to understand Magento 2's automatic constructor dependency injection system. This appendix is meant as a general introduction to dependency injection – if you don't understand everything at first, don't worry. We'll try to address specific, more complicated concerns in the text itself.

Come back! Don't panic! This will be a gentle introduction. You only need to be passingly familiar with these concepts if you're working with Magento's front end code – mainly so you can understand what's going on in the `__construct` method of Magento's blocks, controllers, and other layout related classes.

Understanding the Problem

In PHP, as in many other programming languages, you can create a new object with syntax that looks something like this

```
$object = new SomeClassName;
$object = SomeClassName::createObject();
```

Instantiating an object is a common task in modern programming. However, some programmers have noticed that code like the following

```
function someFunction()
{
    //... does some stuff ...
    $object = new SomeClassName;
    //... does some other stuff with $object ...
}
```

presents long term problems when maintaining a code base. Whenever **anyone** calls this `someFunction` function, the `$object` will always be instantiated as a `SomeClassName` object. For the original programmer who wrote this code, this isn't a problem. However, programmers who come along later might not want to instantiate the `SomeClassName` object. These new programmers may want to instantiate a different object that behaves similarly, or they may want to instantiate a mock object because they're writing unit tests. Because of this, these programmers frown upon writing individual methods or functions that directly instantiate an object.

Instead, the better practice is to pass in the objects your method depends on.

```
1 //type hint for $object to make sure we get the sort
2 //or object this function/method expects
3 function someFunction(SomeClasName $object)
4 {
5     //... does some stuff ...
6
7     //... does some other stuff with $object ...
8 }
```

This makes `someFunction` far more flexible.

Because some programmers like fancy names for things, this is known as *injecting* a dependency into the function, or *dependency injection*. This technique could just as easily be called "always pass in objects instead of directly instantiating them" – but that's a lot more to type.

Where Should Objects be Instantiated

This does raise a question – while it's easy to look at a function or method in isolation and say,

> Hey, we shouldn't instantiate that object here

the objects you inject need to be instantiated **somewhere**. This is where modern dependency injection systems can get a little complicated. One approach would be for a program's `main` function (or equivalent in your language/system of choice) to instantiate all a programs objects

```
1 function main()
2 {
3     $object1 = new SomeClass;
4     $object2 = new SomeOtherClass;
5     //...
6     someFunction($object1);
7
8     someOtherFunction($object1, $object2)
9 }
```

However, in a complex program (or a complex framework) this approach would end up creating a gigantic initial program which would, itself, end up being super complicated. This has led many programming frameworks to create systems for **automatically** injecting dependencies. Magento 2's core team are some of those programmers.

Magento's Automatic Constructor Dependency Injection System

Magento's dependency injection system is large, complex, and the sort of thing you could write an entire book on.

We're going to keep things simple: The following describes how Magento's magic dependency injection system works. If you're interested in how this system was implemented, or its advanced features, the *Magento 2 Object System* series is a good place to start.

http://alanstorm.com/series/magento-2-object-system/

Magento 2's core team have extended PHP's basic object construction features. Whenever Magento 2 instantiates an object, it will look at arguments in the object's constructor function, and **automatically** instantiate objects for these arguments. Consider a class like this

```
1  class SomeClass
2  {
3      public function __construct(
4          \Some\Other\Class $someOtherClass,
5          \Another\Class $anotherClass
6      )
7      {
8          $this->someOtherClass = $someOtherClass;
9          $this->anotherClass   = $anotherClass;
10     }
11 }
```

When Magento instantiates an object from `SomeClass`, it runs code that looks like this pseudo code

```
1  //looks at the type hint for argument 1, instnatiates an
       objet
2  $someOtherClass = new \Some\Other\Class;
3
4  //looks at the type hint for argument 2, instantiates an
       object
5  $anotherClass   = new \Another\Class;
6
7  $ourObject = new SomeClass($someOtherClass, $anotherClass);
```

The actual code is a bit more complicated, and uses something called the Object Manager, but that's a little beyond our scope today. Functionally though, that's what happens whenever Magento instantiates an object for you. These Magento instantiated objects include

1. Controller objects
2. Block objects created via layout handle XML files
3. Objects created via automatic constructor dependency injection

If you directly instantiate a class with the `new` keyword yourself, you won't get this special functionality. **Only** objects that Magento itself instantiates are subject to automatic constructor dependency injection. Of course, Magento recommends that you inject all your dependencies and never use the `new` keyword to create objects yourself. Whether you follow this advice is a topic that, again, is beyond the scope of this article.

Instantiating Interfaces

Let's take a look at another Magento class

```
1  #File: vendor/magento/module-catalog/Ui/DataProvider/
       Product/Form/Modifier/Images.php
2  use Magento\Catalog\Model\Locator\LocatorInterface;
3
4  //...
5
6  public function __construct(LocatorInterface $locator)
7  {
8      $this->locator = $locator;
9  }
```

If we were debugging this class and wanted to know what sort of object Magento injects into `$this->locator`, we'd look at the type hint and say

> Magento's automatic constructor dependency injection instantiates a `Magento\Catalog\Model\Locator\LocatorInterface` object

However – `Magento\Catalog\Model\Locator\LocatorInterface` is a PHP interface! You can't use an interface to instantiate an object! What's going on?!

Here we've stumbled across another feature of Magento's dependency injection system. The system allows you to provide an interface as a type hint and then, via configuration, tell Magento which sort of object to instantiate.

Put another way, Magento has a list of class names to use with interfaces it encounters in constructors. When Magento sees the request to instantiates a `Magento\Catalog\Model\Locator\LocatorInterface`, it looks at its `di.xml` configuration files and finds this

```
#File: vendor/magento/module-catalog/etc/adminhtml/di.xml
<preference
    for="Magento\Catalog\Model\Locator\LocatorInterface"
    type="Magento\Catalog\Model\Locator\RegistryLocator"/>
```

This tells Magento when it encounters a `Magento\Catalog\Model\Locator\LocatorInterface` interface that it should instantiate a `Magento\Catalog\Model\Locator\RegistryLocator` object.

Why Magento uses interfaces is a topic that's a bit too far down the OOP rabbit hole. For now, just accept this as one of those weird things Magento 2 does.

Front End Build System

Although we won't cover it in great detail, Magento uses a "modern" front end build system that's based on NodeJS and the `grunt` build tool. The build system is modern in the sense that it exists (vs. working with plain, non-preprocessed CSS files). However, some bleeding edge front end developers might find both `grunt` and LessCSS beneath them. Regardless of your opinion, these are the systems stock Magento uses, so you'll need to be aware of them.

To use these front end build tools, you'll need to install NodeJS onto your computer. The node onboarding process is pretty straight forward these days – just hop over to

https://nodejs.org

for more information.

You can check if node is installed via the command line and node's `-v` version flag.

```
$ node -v
v6.4.0
```

NodeJS also comes with a command called `npm`. This originally stood for *Node Package Manager*, but these days `npm` can install all sorts of javascript projects to your computer. The `npm` command is similar to `composer.phar` in that it manages code packages for you. However, `npm` handles *javascript* packages.

The `npm` command, like `node`, has a version flag checker

```
$ npm -v
3.10.6
```

Once you have NodeJS and `npm` installed, you'll use these tools to install the grunt cli tool globally. Globally means you'll be able to access grunt no matter

which directory your command line is currently at. A local (non-global) install means you're installing something into a specific node/npm based project.

To install grunt globally, type the following

```
$ npm install -g grunt-cli
```

Once this command finishes running, you should have a `grunt` command available.

```
$ grunt -V
grunt-cli v1.2.0
```

Installing the Local Magento Build Tools

So, you've now got `node`, `npm`, and `grunt` running locally. The next step is to install the `npm` based project Magento ships withs. In the Magento root folder, you'll find a file named `package.json.sample` – copy this file to the name `package.json`

```
$ cd /path/to/magento
$ cp package.json.sample package.json
```

The `package.json` file is `npm`'s version of `composer.json`. It contains a list of javascript packages to install. While Composer will install things to the `vendor` folder, `npm` will install its packages to the `node_modules` folder.

How do you install a node based project? Just run

```
$ npm install
```

Once the command finishes, you'll have a `node_modules` folder. At this point, we'll be able to run grunt locally from our project folder. However, when we do this, we'll get an error

```
$ grunt
A valid Gruntfile could not be found. Please see the
    getting started
guide for more information on how to configure grunt:
    http://gruntjs.com/getting-started

Fatal error: Unable to find Gruntfile.
```

Even though `grunt` has the javascript packages installed locally, `grunt` also needs a configuration file. Fortunately, Magento ships with that configuration file – named `Gruntfile.js.sample`. Just copy this file to the name `Gruntfile.js`

```
$ cp Gruntfile.js.sample Gruntfile.js
```

and you'll be able to run `grunt`. Use the `--help` flag to get a list of the tasks available

```
$ grunt --help
```

For the purposes of this book, the command we'll be running most often is

```
$ grunt clean
```

This will clear out any generated front end files, as well as the cached/preprocessed LessCSS views.

Installing the Pulsestorm_Nofrills Magento Module

This book includes a Magento code module (`Pulsestorm_Nofrills`) which you'll need to install.

There are three broad ways to install a code module into a Magento system. You may

1. Use Magento's Marketplace, whose technology is based on PHP Composer
2. Manually edit your system's composer.json file to install a module hosted elsewhere
3. Manually install code files into the `app/code` folder

Because we're distributing the module files to you directly, the easiest way to install them is via the `app/code` folder. This is a 5 step process

1. Unarchive the files
2. Create (if necessary) an `app/code` folder in your Magento system
3. Move the unarchived files to this folder
4. Run the `module:enable` command
5. Run the `setup:upgrade` command

Unarchive the Files

The `Pulsestorm_Nofrills` module is distributed as a compressed tar archive. Many operating systems support unarchiving these files by double clicking on them. You may also use the command line to extract these files.

```
$ tar -zxvf Pulsestorm_Nofrills.tar.gz
```

Moving the Files

If necessary, create an `app/code` folder in your system. The `app` folder should already exist, the `code` folder may not. Once you have this folder in place, move the module files from the previous step to the `app/code` folder. Once you've moved everything, you should have a file at

```
app/code/Pulsestorm/Nofrillslayout/etc/module.xml
```

This is not the only file in your module – it's just a convenient one to use to make sure you've moved your module files to the correct folder.

Telling Magento About the Module

Finally, we need to tell Magento about the new module. We do this by running the following two command line commands

```
$ php bin/magento module:enable Pulsestorm_Nofrillslayout
//...
$ php bin/magento setup:upgrade
//...
```

If you don't know about the Magento command line, checkout *The Command Line* appendix.

At this point, the module is installed and you should be ready to start!

Interfaces

Interfaces are a feature of the PHP programming language. They originally come from the java programming language, and fall under the broad category of "Object Oriented Programming". By and large a front end developer working with Magento doesn't need to understand interfaces, but if you want to have a better understanding of *why* certain features work a certain way understanding interfaces is important.

Also, if this is the first time you've encountered interfaces you probably won't understand them on your first read through. If you come back in a year you may start to understand their practical application. Interfaces are a deceptively simple, but advanced topic. Don't be discouraged if you don't get them right away.

Implementing Interfaces

On their face, PHP's interface feature is a straightforward thing. A PHP class may *implement* one or many interfaces.

```
class Foo extends Bar implements FirstInterface,
    SecondInterface {...}
```

What does implementing an interface mean? Well, an interface is a collection of abstract method signatures.

```
interface FirstInterface
{
    abstract public function baz();
    abstract public function zip($one, $two);
}
```

When your class implements an interface, PHP won't run your program unless it has defined all the methods in that interface.

That's all relatively straight forward, and the PHP manual has more information on the specifics of how all this works.

http://php.net/manual/en/language.oop5.interfaces.php

What's less clear is *why* anyone would want to use interfaces. If you're writing code to accomplish a simple task, interfaces don't seem to offer any obvious advantage.

That's because they **don't** offer any advantage. An interface isn't for a programmer writing code to complete a simple task. Instead, interfaces are a tool that's best used by a programmer *writing code that other programmers will use.*

An interface allows a system developer to say *ok client programmer, if you want to build your own version of this class, you can. Just make sure it implements all these methods.* Interfaces let the client programmer understand which methods on a class are the important ones, and which are the helpers. By themselves that's all interfaces do, and if you've never encountered them before that can be a tricky concept to get your head around.

Type Systems and Magento 2

Interfaces come to us from the java programming language. They're one of many programming language features that help systems developers create *type* systems for their application or system. In the mists of programming past, variable and object types started as a way of telling the computer how much memory it should set aside for certain data. However, over the years, many smart people have embraced types as a way of writing better, *more correct* programs.

Whether these systems are worth the time or introduce more problems than they solve is another topic for another day (hint: it depends). We mention this mainly as a prelude to explaining the extra systems that Magento's build on top of interfaces.

If you've read the dependency injection appendix, you already have some understanding of Magento's custom object system. Magento doesn't want its developers directly instantiating objects from classes. Instead, they want developers to list their class dependencies in a class's constructor and have Magento's automatic constructor dependency injection system create those objects automatically.

Magento takes this concept one step further with interfaces. In Magento's ideal version of the world, developers using dependency injection aren't asking the dependency injection system to instantiate a class. Instead, they're asking the dependency injection system to instantiate an interface.

If you know anything about interfaces this may confuse you – it's a core tenant of interfaces that you **can't** instantiate an object from them. So what does it mean to ask Magento's object system to instantiate an interface?

Magento keeps a list of which concrete classes should be instantiated for each interface. You can actually add to this list (for your own interfaces), or change this list (for Magento and third party interfaces) via Magento's `di.xml` configuration (a topic for another day and another book).

So if that's the how – **why** would Magento do something like this? In practical terms, by developing this system Magento are encouraging developers to think about interfaces for their classes first. Some experts consider this a best practice. That aside, there is a distinct whiff of

> Why did they climb the mountain? Because it's there!

filtering down from Magento's dependency inject basecamp.

Practically speaking, as a front end developer, you'll be mostly shielded from these sorts of things. However, the more you're involved with an open-source system like Magento, the more you'll be exposed to its internal code. Having a general understanding of the hows and whys will always serve you well, even if you never write a line of object oriented PHP code.

Magento Modes

Once you've installed Magento 2, there's three different *modes* you can run it in.

- Developer
- Default

- Production

These modes impact how certain Magento systems behave, and how "automatically" certain things happen in Magento.

One concrete example of that is front end asset files (JS files, CSS files, images, etc). Magento stores these files inside code modules. In order to make them available to the public for serving via URLs, you'll need to run the

```
1 php bin/magento setup:static-content:deploy
```

command. This command will look at all the installed modules and configured themes, and create front end assets for you in the `pub/static` folder.

When you're running in production mode, if Magento can't find a CSS file, it reports a file 404 and does nothing more.

However, when you're running in developer or default mode, Magento will attempt to automatically copy or symlink files from the module folders to `pub/static` for you. This means if a file isn't there, Magento does some extra work to find it and put it in place for you.

These three modes exists because of the performance and security tradeoffs involved in each of these features.

Speaking generally, production mode does nothing automatic for you. You need to run commands like `setup:static-content:deploy` and `setup:di:compile` to generate static versions of things that are created automatically for you when running in developer mode. This helps with Magento's performance, and creates less of a chance that one of these dynamic systems is hijacked for nefarious purposes.

When you first install a Magento system, it will be in `default` mode. The `default` mode sits somewhere between production and developer, and tries to create a "hands off" first run experience that's still relatively performant.

For most of this book, we'll expect you to be running in `developer` mode.

Changing Modes

There are two ways to change modes of your Magento 2 system.

The first is the command line `bin/magento` program's `deploy:set:mode` command

```
1 php bin/magento deploy:mode:set developer
2 php bin/magento deploy:mode:set production --skip-
      compilation
```

This allows you to swap between modes, and automatically run the generation (i.e. compilation) steps needed for production mode. This command sets the `MAGE_MODE` value in `app/etc/env.php`

```
#File: app/etc/env.php

  'MAGE_MODE' => 'production',
```

In addition to `app/etc/env.php`, it's also possible to set the Magento mode in your `.htaccess` file(s) or nginx equivalent configuration.

```
#File: .htaccess
SetEnv MAGE_MODE "developer"

#File: pub/.htaccess
SetEnv MAGE_MODE "developer"
```

Curiously, values set in the web server configuration will override the value set via `deploy:mode:set` – so if you're seeing strange behavior in your systems be sure to check that you don't have your Magento mode configured differently in each of these files.

Unix Find

This book will occasionally use the venerable unix `find` command to look for files that match a particular pattern. This appendix will briefly cover `find`'s syntax in case you've never run across it before.

Consider the following command, run from a Magento root project folder

```
$ find vendor/magento -name '*.phtml'
```

The first argument to the `find` command – `vendor/magento` above – is the folder you want to search. You can actually run `find` with just this single argument to list out **all** the files in a folder.

```
$ find vendor/magento
vendor/magento/
vendor/magento//composer
vendor/magento//composer/.gitignore
vendor/magento//composer/composer.json
vendor/magento//composer/LICENSE.txt
//... huge list snipped ...
vendor/magento//zendframework1/resources/languages/sk/
    Zend_Captcha.php
vendor/magento//zendframework1/resources/languages/sk/
    Zend_Validate.php
vendor/magento//zendframework1/resources/languages/sr
vendor/magento//zendframework1/resources/languages/sr/
    Zend_Validate.php
```

```
12 vendor/magento//zendframework1/resources/languages/uk
13 vendor/magento//zendframework1/resources/languages/uk/
       Zend_Validate.php
14 vendor/magento//zendframework1/Vagrantfile
```

However, when used with the `-name` option (as we have above and below)

```
1 $ find vendor/magento -name '*.phtml'
```

The `find` command will pare down this list using the provided wildcard expression (`*.phtml` above). This means our command will find all the `phtml` template files in Magento's `composer` source directory.

Another common `find` invocation will look something like this

```
1 $ find . -name '*.phtml'
```

When the first argument is a `.`, `find` will search starting at the current directory. If you're not familiar with unix/linux, `.` usually refers to the *current* directory, whereas `..` refers to the directory **above** the current directory.

Viewing HTML Source

There will be a number of times in this book where we ask you to view the rendered HTML source of your page. For long time web developers, this is a simple, obvious task. However, I've noticed that there's a certain class of young javascript developers who are **only** aware of HTML tags as they relate to their UI abstraction (React, Knockout.js, etc.) of choice.

Given Magento 2 is still a PHP MVC/MVVM framework that relies on server rendered HTML, a quick primer on the differences between a page's raw source and rendered source seems appropriate.

HTTP and the Web Browser

Whenever a user requests a URL from a server (i.e. visits a web page), the web browser connects your computer to server, and gets back an HTML page. In modern web applications, code on the web page itself may make additional requests via ajax (also known as "XHR") for more HTML chunks, or JSON and XML data objects, but that initial visit that returns an HTML page is **always** required.

Most computer based (i.e. not phone or tablets) web browsers still have a "View Source" menu that will let you view the raw source of that initial HTTP request.

For example, in Google Chrome, if you navigate to

```
View -> Developer -> View Source
```

you'll see the HTML file returned by the server.

We say "raw source" because most modern browsers also have a way to inspect the *rendered source* of a page. After a web browser downloads a web page, it will run any javascript programs included on the page. These javascript programs can change the HTML elements included in a page. Additionally, these javascript programs can setup event handlers that respond to certain user actions (clicks, hovers, scrolls, taps, etc.), which can further change the initial raw source.

In Google Chrome, if you browse to

```
View -> Developer -> Developer Tools
```

and click on the `Elements` tab you're looking at the **rendered** source of a page – i.e. **after** any javascript has been run.

Other Tools for Viewing a Page Source

If you're familiar with the unix command line (and if you're not, you will be after doing Magento development for a bit), there's another way you can view the HTML source of a document.

The command line program `curl`, which ships with most *nix computers these days (MacOS included), is a program that can fetch the source of *any* URL. For example, to fetch the raw source of the Google homepage, just type the following command in your unix terminal.

```
$ curl 'http://www.google.com'
<!doctype html><html itemscope="" itemtype="http://schema.o
    ...
```

If you want to send the source to a file, just use a unix > (redirect) character

```
$ curl 'http://www.google.com' > file.html
```

Even the `View Source` menu of a web browser will sometimes add formatting to the HTML document. The `curl` program can show you **exactly** what's being returned from a web server. This can be a bit fiddly sometimes. For example, if you try to fetch `google.com` (and not `www.google.com`), you'll end up with a result that looks like this

```
$ curl 'http://google.com'
<HTML><HEAD><meta http-equiv="content-type" content="text/
    html;charset=utf-8">
```

```
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document has moved
<A HREF="http://www.google.com/">here</A>.
</BODY></HTML>
```

That's because `google.com` redirects to `www.google.com`, and `curl` shows you **exactly** what the server returns (you can work around this specific problem by using the `-L` option).

A `curl` request may also send different HTTP headers than a browser, which may make certain web servers act differently. Despite this fiddly business, or maybe **because** of it, `curl` is a tool that should be in every web developer's tool-belt.

You may start using `curl` to quickly look at a browser's page source, but after a while, you'll start learning the fundamentals of the HTTP protocol, which is the protocol that underlies the entire world wide web. That's a smart thing to do if you're going to make web development your long term home.