CHAPTER 5

Drupal 8 Theming

Drupal themes represent the components of a Drupal site that render content on any browser equipped device in a visually appealing fashion. If Drupal is the cake, then themes are the frosting and decoration—they make your web site beautiful.

The art of theming requires a mixture of visual design skills, including using tools such as Photoshop, as well as experience in developing HTML markup and cascading stylesheets (CSS), using JavaScript, and using the Twig templating engine to connect the theme to the output generated by Drupal.

This chapter covers the details of the role of a Drupal theme, how themes are structured, how themes are installed and enabled, and the process and components required to create a custom Drupal 8 theme, including the yaml files required to describe your theme, the HTML markup, CSS, and Twig components required to assemble pages on your site. It is assumed that you have a basic understanding of HTML, CSS, and JavaScript before attempting to construct a custom theme.

The Role of a Drupal Theme

Drupal themes have one primary responsibility—to provide the means for displaying content that is generated by the various Drupal components such as blocks and views in a visually appealing manner. The theme provides physical containers on the page, typically called *regions,* which are used by site builders to place content, blocks, views, and other visual elements. The theme also joins the physical layout and regions with the cascading stylesheet elements and JavaScript to deliver to the browser a complete package ready to be viewed by the site visitors.

While the theme is responsible for defining regions on a page, CSS, and JavaScript, the theme may also override the visual presentation of the output generated by blocks, views, and modules.

While the theme is responsible for defining the structure and visualization of pages, content, and elements on the page, it's important to talk about the underlying templating engine that is the workhorse of Drupal themes, responsible for integrating the output generated by Drupal and its components with the HTML markup and CSS. That component is the Twig templating engine and it's new to Drupal as of Drupal 8. It replaces the previous templating engine, the PHP template.

The Twig Templating Engine

While HTML, CSS, and JavaScript play a key role in Drupal 8 themes, the true star of the show is the Twig templating engine (`twig.sensiolabs.org`). Twig is a component of the Symfony2 framework, which is the underlying architecture that Drupal 8 is built on. Think of Twig is the "glue" between the output that is generated by a Drupal module and the rendered page that is presented to the site visitors.

T. Tomlinson, *Enterprise Drupal 8 Development*, DOI 10.1007/978-1-4842-0253-1_5

The Twig templating engine uses relatively simple syntax consisting of variables, expressions, and tags. The Twig templating engine converts each of those elements into highly optimized PHP code that binds the output of Drupal's modules to the rendered page. A simple example of using Twig to render a block's title and body is as follows:

```
<h3>{{ block.title }}</h3>
<div>{{ block.content }}</div>
```

While there would likely be conditional logic wrapping the output, for example, checking to see if the title and content existed before rendering it, the example demonstrates the simplicity of Twig and its syntax. The block module exposes `block` to the theme layer, with `title` and `content` as elements within the block that are rendered using the previous syntax. Before exploring the details of Twig syntax, let's first look at the elements required to create a Drupal 8 theme.

The Structure of a Drupal Theme

Drupal 8 themes, like modules, require a specific set of files in a standard directory structure in order to function properly. All themes that are not part of Drupal 8 core reside in the `theme` directory at the root directory of your Drupal 8 site. Within the `theme` directory, if you have not yet done so, create two subdirectories:

- `contrib`: This is where all themes downloaded from `drupal.org` are stored
- `custom`: This is where all themes you create for your site will reside

Focusing on a custom theme, the directory structure required to support the creation of a custom theme is as follows:

```
themename
  config
    install
    schema
  css
  js
  images
  templates
```

Where `themename` is the name of your theme. In the `themename` directory you will find several files (replace `themename` with the actual name of the theme):

- `themename.info.yml`: The only mandatory file for a Drupal 8 theme. This file describes the metadata about your theme, for example the name of your theme, as well as libraries, regions, and the version of Drupal core that is required to use the theme.
- `themename.libraries.yml`: Defines the JavaScript and CSS libraries that are loaded by the theme.
- `themename.breakpoints.yml`: Defines the screen widths where the design needs to change to accommodate different devices.
- `themename.theme`: Contains all of the conditional logic and preprocessing of output that occurs before it is rendered on the page. It may also extend the basic theme settings by creating advanced theme settings.

- `screenshot.png`: Rendered on the Appearance page, giving the site builder a preview of the theme.
- `logo.svg`: The standard logo rendered on a page in the header section of your site. You may provide a standard logo as part of your theme or upload a logo through the Appearance ➤ Settings page.
- `.css`: There may be one to many CSS files in the `css` directory.
- `.js`: There may be one to many JavaScript files in the `js` directory.

We cover the `config` directory and advanced configuration options later in this chapter.

Creating the Theme Files

To demonstrate the process of creating the files associated with a Drupal 8 theme, we start with the creation of the directory structure as described previously, for a new theme called `davinci`. We first create a new directory in the `themes` directory called `davinci`, and then create the same subdirectories as described previously, resulting in the directory structure shown in Listing 5-1.

Listing 5-1. The Davinci Theme Directory

```
└── davinci
    ├── config
    │   ├── install
    │   └── schema
    ├── css
    ├── images
    ├── js
    └── templates
```

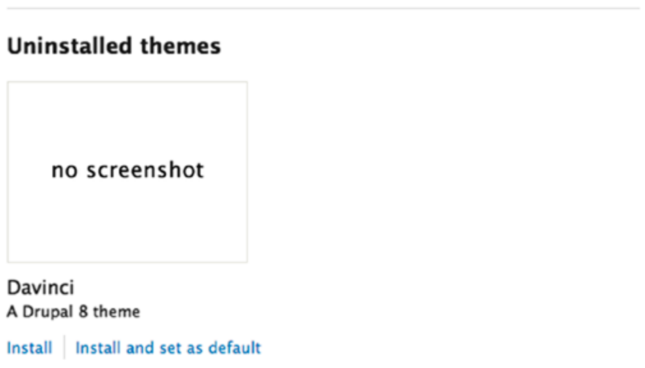
The next step in the process is to create the `.info.yml` file. Using your favorite text editor, create the `davinci.info.yml` file in the root directory of your new theme (`themes/custom/davinci`) with the following code:

```
name: Davinci
type: theme
description: A Drupal 8 theme
core: 8.x
```

The values in this file are as follows:

- `name` defines the value that will appear on the Appearance page.
- `type` defines what this `.info.yml` file defines, in this case a theme.
- `description` is text that appears on the Appearance page and describes your theme to the site administrator.
- `core` defines the version of Drupal that your theme supports.

Although there are more elements that may be added to the `.info.yml` file, those listed previously are all that are required to display the theme on the Appearance page and all that are required to enable the theme. Save the file, rebuild the cache, and visit the Appearance page. You'll now see the new theme, as shown in Figure 5-1.



Uninstalled themes

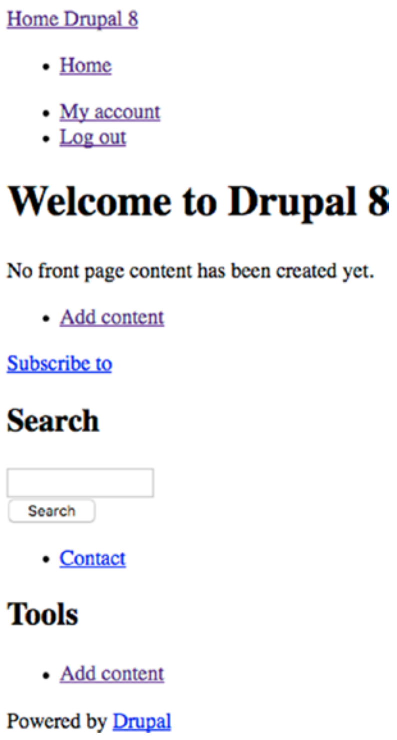
no screenshot

Davinci
A Drupal 8 theme

Install | Install and set as default

Figure 5-1. *The Davinci theme*

Clicking on the Install and Set Default link and visiting the homepage reveals a rather stark design, but it is the starting point and it will only get better from here (see Figure 5-2).



Home Drupal 8

- Home
- My account
- Log out

Welcome to Drupal 8

No front page content has been created yet.

- Add content

Subscribe to

Search

Search

- Contact

Tools

- Add content

Powered by Drupal

Figure 5-2. *The site rendered with the Davinci theme*

Adding Regions to the Theme

One of the power features of themes is the ability to define regions on a page where content, blocks, menus, or other elements that are rendered by modules can be placed. The regions defined by a theme appear on the Structure ➤ Block Layout page, where a site builder can place blocks into each of the defined regions.

For demonstration purposes, we create several regions in the Davinci theme. Figure 5-3 depicts the general layout of the regions that will appear on every page.

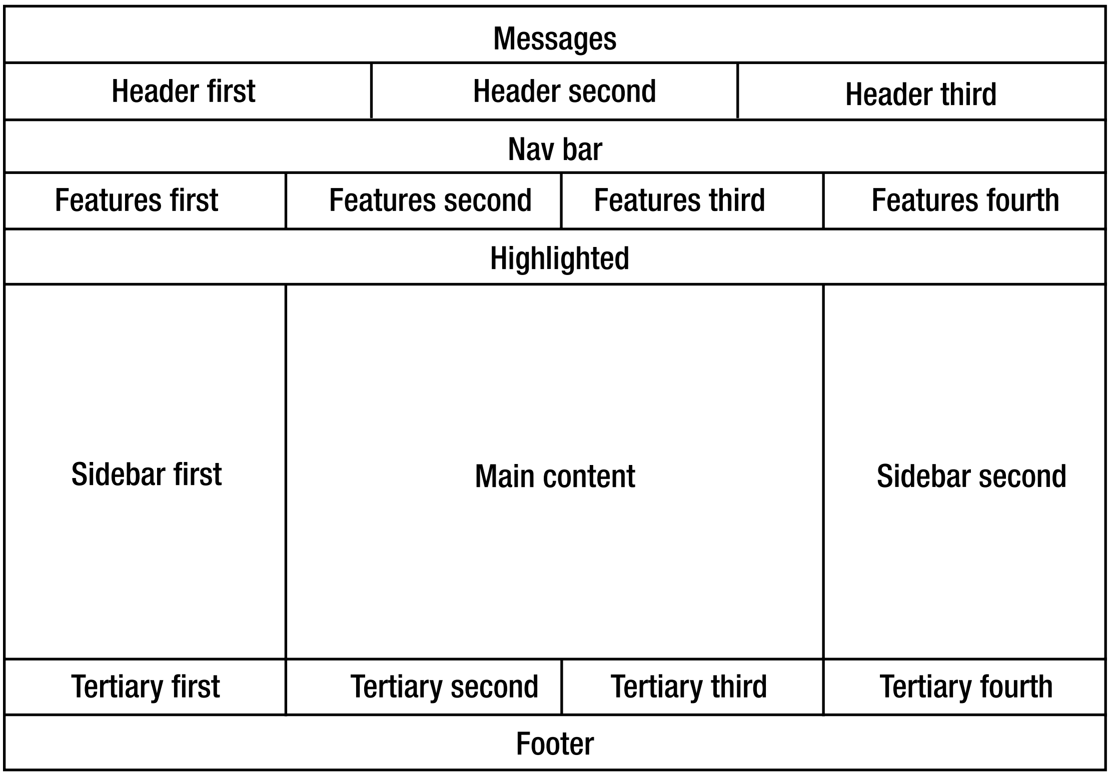


Figure 5-3. *The regions of the Davinci theme*

Not every theme must have as many regions as shown in Figure 5-3, and there may be cases where you need more regions than are provided by the Davinci theme. The choice is up to the designer.

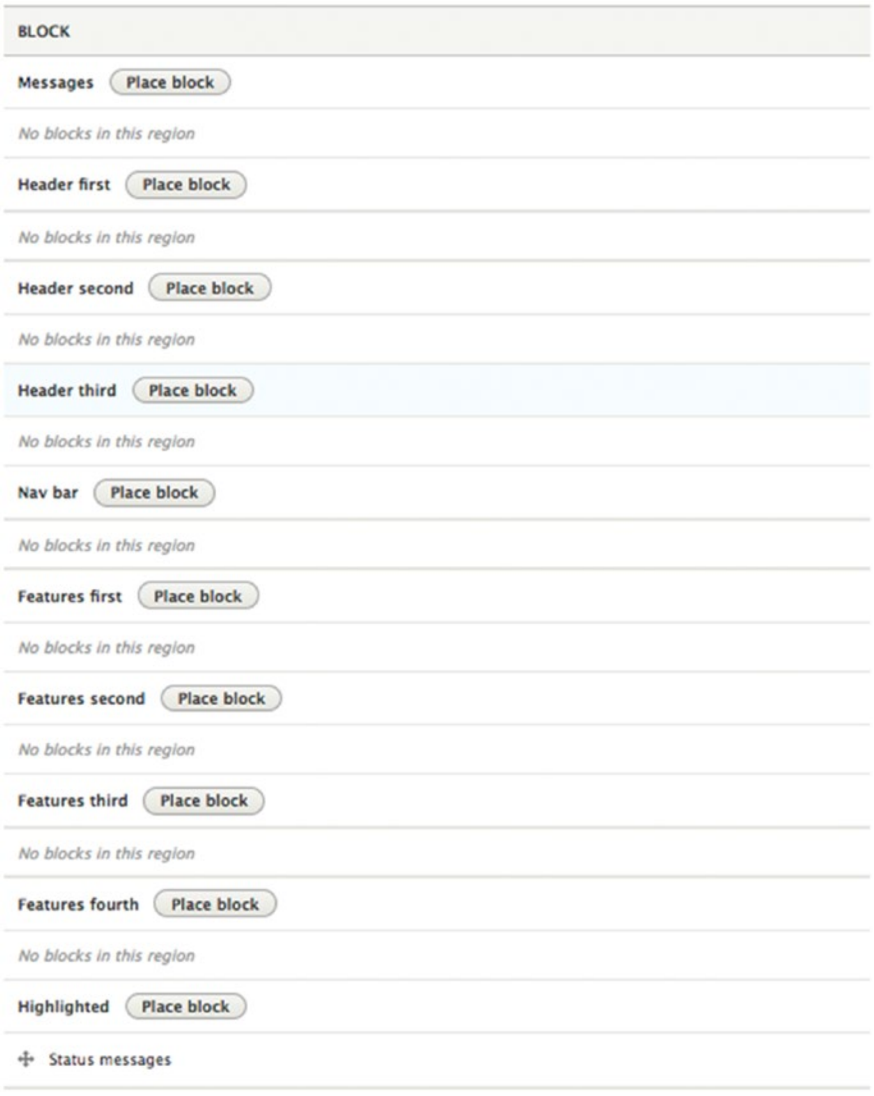
Regions are defined in the `.info.yml` file in a section titled regions. The structure of a region's definition is the internal name of the region, for example `header_first`, followed by the name of the region that will appear on administrative interfaces, such as the Block Layout page. In the case of `header_first`, the value displayed will be `Header first`. Note: Spacing in `.yml` files is important and has meaning. Each of the regions defined in the `regions` section are indented exactly two spaces, which is the yaml syntax for elements within a group. Expand the `davinci.info.yml` file, adding all of the regions shown in Figure 5-3.

```
name: davinci
type: theme
description: A Drupal 8 theme
core: 8.x
regions:
  messages: 'Messages'
  header_first: 'Header first'
  header_second: 'Header second'
  header_third: 'Header third'
  navbar: 'Nav bar'
  help: 'help'
  features_first: 'Features first'
  features_second: 'Features second'
  features_third: 'Features third'
  features_fourth: 'Features fourth'
  highlighted: 'Highlighted'
  content: 'Main content'
  sidebar_first: 'Sidebar first'
  sidebar_second: 'Sidebar second'
  tertiary_first: 'Tertiary first'
  tertiary_second: 'Tertiary second'
  tertiary_third: 'Tertiary third'
  tertiary_fourth: 'Tertiary fourth'
  footer: 'Footer'
  page_top: 'Page top'
  page_bottom: 'Page bottom'
```

Drupal 8 core requires that three regions exist in every theme:

- `content`: The primary container for content on a page
- `page_top` and `page_bottom`: Regions that are hidden by default(they do not appear on the Block Layout page) and are used by modules to place markup and JavaScript at the top and bottom of pages

After updating and saving the file, we rebuild the cache and visit Structure ➤ Block Layouts, where the new regions appear, as shown in Figure 5-4.


BLOCK

Messages Place block

No blocks in this region

Header first Place block

No blocks in this region

Header second Place block

No blocks in this region

Header third Place block

No blocks in this region

Nav bar Place block

No blocks in this region

Features first Place block

No blocks in this region

Features second Place block

No blocks in this region

Features third Place block

No blocks in this region

Features fourth Place block

No blocks in this region

Highlighted Place block

Status messages

Figure 5-4. *A partial listing of the new regions*

At this juncture, we could place blocks and other elements into the regions shown in Figure 5-4; however, before we can see the elements we've placed in the regions, we have to define the templates that will render those regions. But before diving into the details of template files, let's start with an overview of Twig syntax and functionality. Having an understanding of Twig syntax and functionality will make the discussion about the content of template files more meaningful.

Twig Syntax

There are three general categories of "things" that Twig does:

- It "says something"
- It "does something"
- Or it's a comment

Each of these items has a specific syntax. The "say something" syntax is represented by `{{ ... }}`, where the opening and closing braces are identified by the Twig parser as something that Twig needs to do, and the `...` represents a variable that will printed to the page that is being rendered. For example, `{{ name }}` would print the value associated with a variable called `name`.

The "do something" syntax is associated with `if` and `for` statements, setting the value of variables, filters, and other less common functions. The syntax for "do something" is `{% ... %}`. For example:

```
{% filter upper %}
   {{ name }}
{% endfilter %}
```

This prints the value of `name` in uppercase. We cover additional uses of "do something" in the sections that follow. The syntax for comments in Twig templates is `{# ... #}`.

Combining "say something" with "do something" provides all of the functionality required to connect content in Drupal with the rendered page. The next sections expand on the capabilities of each.

Twig Variables

One of the primary functions of Twig is to print the content of a variable on a page. If you have a variable called `first_name`, printing the value of that variable to the page is accomplished through `{{ first_name }}`. Where did `first_name` come from? Most likely a module that generated a value and assigned it to a themable variable named `first_name`.

The `first_name` variable is a simple example, but variables generated by Drupal modules are often more complex such as an array, an object, or a function. Twig handles that by automatically searching for the possible sources of that variable. Using an example of a object named `customer`, with attributes of that customer being `name`, `address`, `city`, `state`, `postal_code`, `email`, and `phone`, we can print the value of the customer's name using `{{ customer.name }}`. When Twig evaluates `customer.name`, it searches for the following:

- `$customer['name']`
- `$customer->name`
- `$customer->__isset('name') && $customer->__get('name')`
- `$customer->name()`
- `$customer->getName()`
- `$customer->isName()`
- `$customer->__call('name')`

In nearly every case, one of the references will return the customer's name. As a frontend developer, you need not worry about how the backend developer stored the value of `name`; Twig handles the details for you.

While most variables are created by modules, there may be scenarios where you need to create and use a variable in your template. To create a variable and assign a value to it, use the following syntax:

```
{% set hello = 'Hello World' %}
```

Twig also supports the creation of key => value arrays, although in Twig they are called *hashes*. To create a hash, use the following syntax:

```
{% set name = { first: 'John', last: 'Doe' } %}
```

In this example, you print the values using `{{ name.first }}` or `{{ name.last }}`. Hashes may also be nested, for example:

```
{% set sports = {
      football:  { team: "Seahawks", city: "Seattle" },
      basketball: { team: "Trailblazers", city: "Portland" },
      soccer: { team: "Timbers", city: "Portland" },
      baseball: { team: "Mariners", city: "Seattle" }
} %}
```

You would print values from these example using a statement such as `{{ sports.football.team }}`.

Discovering Variables

As a frontend developer who creates template files, you will see that it's easy to find any variable you define within the template through a `{% set %}` statement. But most variables are created outside of templates through Drupal modules. While the standard Twig templates included with Drupal core have a well-documented list of variables that are available within that template, those documentation blocks will not include variables created by contributed modules that you have installed on your site.

While you could dig through the custom modules to find instances of where theme variables are created, there is an easier way—using the `{{ dump() }}` function. The `dump` function is available after you have enabled Twig debugging in the `sites/default/services.yml` file. By default, Twig debugging is set to `false`. Change the value to `true` and rebuild cache before attempting to use the dump function. Note: Using the `dump()` function may cause out of memory errors, so use it with caution. You may choose to use `{{ dump(_context|keys) }}` instead, which only shows variables that are available to your template.

If you know the name of a variable, you can use `{{ dump(variable_name) }}` to display the value, replacing `variable_name` with the actual name of the variable. If you don't know which variables are available, you can use the `{{ dump() }}` function, which returns all variables that are known to the template that you are working on. For example, adding the `dump` function to the Bartik themes `node.html.twig` template, saving the revised template, and rebuilding the cache results in the following output, which shows every variable that is available to the `node` template:

```
array(28) { [0]=> string(8) "elements" [1]=> string(19) "theme_hook_original"
 [2]=> string(10) "attributes" [3]=> string(16) "title_attributes"
 [4]=> string(18) "content_attributes" [5]=> string(12) "title_prefix"
 [6]=> string(12) "title_suffix" [7]=> string(12) "db_is_active" [8]=> string(8) "is_admin"
 [9]=> string(9) "logged_in" [10]=> string(4) "user" [11]=> string(9) "directory"
 [12]=> string(9) "view_mode" [13]=> string(6) "teaser" [14]=> string(4) "node"
 [15]=> string(4) "date" [16]=> string(11) "author_name" [17]=> string(3) "url"
 [18]=> string(5) "label" [19]=> string(4) "page" [20]=> string(7) "content" [21]=> string(17)
 "author_attributes" [22]=> string(17) "display_submitted" [23]=> string(14) "author_picture"
 [24]=> string(6) "#cache" [25]=> string(8) "metadata" [26]=> string(22) "theme_hook_suggestions"
 [27]=> string(7) "classes" }
```

Although the output is not very pretty, it does show all of the available variables, such as author_name, url, label, page, etc. Inspecting a specific variable, such as url using {{ dump(url) }} displays the following:

```
string(7) "/node/1"
```

Conditionals, Looping, Filters, and Math Functions in Twig

While the theme layer isn't the place for business logic, there are cases where you will want to "do something" with the information that is available to the template that you are working with. Twig provides the ability to do so through conditional logic (if statements), looping (foreach), and filters (e.g., transform text to uppercase).

Twig Conditionals

You can test the content of variables using if statements in Twig in the form of {% if <variable> <condition> <comparison> %}. The conditions that may be checked in Twig are as follows:

a == b to test that the value of a and b are equal

a != b to test that a is not equal to b

a <> b to test that a is not equal to b

a < b to test if a is less than b

a > b to test that a is greater than b

a <= b to test that a is less than or equal to b

a >= b to test that a is greater than or equal to b

a === b to test that the value and type of variable are the same for a and b

a !=== b to test that a and b are not identical

Examples of each conditional are as follows:

```
// test if a is equal to b
{% if a == b %}
   {{ A equals b }}
{% endif %}

// test if a is not equal to b
{% if a != b %}
   {{ A is not equal to b }}
{% endif %}

// test if a is not equal to b
{% if a <> b %}
   {{ A is not equal to b }}
{% endif %}
```

```
// test if a is greater than b
{% if a > b %}
   {{ A is greater than b }}
{% endif %}

// test if a is less than b
{% if a < b %}
   {{ A is less than b }}
{% endif %}

// test if a is less than or equal to b
{% if a <= b %}
   {{ A is less than or equal to b }}
{% endif %}

// test if a is greater than or equal to b
{% if a >= b %}
   {{ A is greater than or equal to b }}
{% endif %}

// test if a and b are equal and the same type
{% if a === b %}
   {{ A and b are equal and the same type }}
{% endif %}

// test if a and b not equal and the same type
{% if a !== b %}
   {{ A and b are not equal and the same type }}
{% endif %}
```

Twig also supports multiple conditions in a single `if` statement, for example:

```
{% if a < b or b < c %}
   {{ a is less than b or b is less than c }}
{% endif %}

{% if a < b and b < c %}
   {{ a is less than b and b is less than c }}
{% endif %}
```

Twig supports `if else` statements, for example:

```
{% if a < b %}
   {{ a is less than b }}
{% elseif a > b %}
   {{ a is greater than b }}
{% elseif a == b %}
   {{ a equals b }}
{% endif %}
```

Twig supports testing for the existence of a value. For example, to test whether the variable `user.name` is set, use:

```
{% if user.name %}
  {{ user name is user.name }}
{% endif %}
```

You can also test for when a variable is not set:

```
{% if not user.name %}
  {{ user name is not set }}
{% endif %}
```

You can also do string operations such as:

```
{% if 'Hello' starts with 'F' %}
```

or

```
{% if 'Hello' ends with 'N' %}
```

or run contains comparisons such as:

```
{% if 'cd' in 'abcde' %}
{{ 'cd' is in 'abcde' }}
{% endif %}
```

or

```
{% if 1 in [1,2,3] %}
{{ 1 is in 1,2,3 }}
{% endif %}
```

Twig also provides a PHP-like `switch` statement that allows you to write a more legible control statement than using a long list of `if-elseif` statements. The form of the `switch` statement is as follows:

```
{% switch user.type %}
  {% case "administrator" %}
     {{ User is an administrator }}
  {% case "editor" %}
     {{ User is an editor }}
  {% case "anonymous" %}
     {{ User is anonymous }}
  {% default %}
     {{ User type is user.type }}
{% endswitch %}
```

Twig's control statements provide a powerful solution for handing conditional logic in template files. Twig also provides powerful and easy-to-use capabilities for iterating or looping through elements.

Looping in Twig

Twig provides two different mechanisms to loop or iterate over elements that are exposed in a template file. You may use for loops to perform some functionality one to many times using the following syntax:

```
{% for i in range(0,3) %}
    {{ i }}
{% endfor %}
```

In the previous example, the variable i starts with a value of zero and is incremented by 1 until it equals 3. The functionality within the for statement, in this case, will print 0 1 2 3.

Iterating over a list of variables is accomplished through another form of the for statement. In the following example, the variable items is an array of objects, where content is an attribute of the item object. This look assigns each element of the items array to a variable named item, and then the value of the content attribute is printed.

```
{% for item in items %}
    {{ item.content }}
{% endfor %}
```

Either version of the for statement may be nested to perform more complex looping and iterating scenarios.

Twig Filters

Filters are a mechanism for performing transformations and evaluations of a string or the value stored in a variable. For example, {{ 'HELLO WORLD' | lower }} would print hello world. Filters also provide the ability to evaluate certain aspects of variables, such as counting the number of elements in an array through the length filter.

```
{% if users|length < 1 %}
   {{ There are no users }}
{% endif }
```

There are several filters available in Twig, some of the most commonly used filters are as follows:

- abs: Determines the absolute value of a number {{ count|abs }}
- capitalize: Converts the first character of a string to a capital letter; for example, {{ 'hello world'|capitalize }} would print Hello world
- date: Formats a date to a given format {{ published_date|date("m/d/Y") }}. The date filter accepts any date that is supported by the PHP function strtotime, or DateTime and DateInterval instances. The date functional also works with the value of now, which is the current date and time; for example, {{ "now"|date("m/d/Y") }}.
- date_modify: Alters the date value in a variable by, for example, adding one day to the value {{ published_date|date_modify("+1 day")|date("m/d/Y") }}. You can use any of the date modifiers supported by the PHP function strtotime.
- default: Provides the ability to assign a default value to a variable if the variable is undefined or empty {{ person|default('anonymous') }}.

- `escape`: Provides the ability to escape a value using `html`, `js`, `css`, `url`, or `html_attr` contexts for safe insertion into the final output. Examples of using HTML and `js` are `{{ content|e('html') }}` or `{{ content|e('js') }}`.
- `first`: Returns the first element of an array or a string. For example, `{{ 'ABCD'|first }}` returns `A`. For arrays, `{{ [A,B,C,D]|first }}` would also return `A`.
- `format`: This filter applies string transformations using the `printf` notation. For example, `{{ "The user %s is from %s."|format(user.name, user.location) }}` would output `The user John is from New York`.
- `join`: Concatenates values from items in a sequence. For example `{{[A,B,C,D]|join }}` would return `ABCD`. You may also insert values between items being joined, for example `{{ [A,B,C,D]|join('|') }}` would return `A|B|C|D`.
- `keys`: Returns the keys of an array. For example

  ```
  {% for key in array|keys %}
     {{ key }}
  {% endfor % }
  ```

  would iterate over the keys of an array and print them.
- `last`: Similar to the first filter, `last` returns the last element in an array or the last character in a string `{{ '1234'|last}}`.
- `length`: Returns the number of items in an array or the length of a string, for example `{% 'ABCD'|length %}` would return `4`.
- `lower`: Transforms a string to all lowercase. For example `{{ 'HELLO WORLD'|lower }}` would print `hello world`.
- `merge`: Merges two arrays into a single array. For example

  ```
  {% set berries = ['strawberry', 'blackberry', 'raspberry'] %}
  {%  set fruit = ['apple', 'orange', 'grapes'] %}
  {% set salad = berries|merge(fruit) %}
  ```

  would result in a new array named `salad` with the combination of `berries` and `fruit`.
- `number_format`: Transforms a number into a given format using the the same functionality as PHP's number_format function. For example `{{ 1234.567|number_format(2, '.', ',')}}` would return `1,234.56`.
- `replace`: Formats a given string by replacing the placeholders with values that are specified in the replacement pattern. For example, `{{ "All cows are %color%."|replace({'%color%': brown}) }}` would result in `All cows are brown`.
- `round`: Rounds a number. For example, `{{ 3.145|round }}` would output `3`. You may also specify floor or ceiling to force the rounding to always round down or up. For example, `{{ 3.145|round(1,'floor') }}` would output `3.1`.
- `slice`: Extracts a piece of an array or a string given a start position and the number and the length of the slice to return. For example, `{{ 'ABCD'|slice(2,2) }}` would output `BC`.

- `sort`: Sorts an array using PHP's `asort` function. For example:

  ```
  {% for user in users | sort %}
     {{ user.name }}
  {% endfor %}
  ```

  would sort the user's array in ascending order.

- `split`: Splits a string by a given delimiter and returns a list of strings. For example:

  ```
  {% set colors = "red, green, blue"|split(',') %}
  ```

  would result in colors as an array set to `['red', 'green', 'blue']`. You can also set the limit on how may elements to parse, with the remaining elements set to the last element of the array. For example:

  ```
  {% set addresses = "123, 456, 789, 0"|split(',',2) %}
  ```

  would result in an array of `Array[0] = ['123'], Array[1]=['456,789,0']`.

- `striptags`: Removes all SGML and XML tags from a string `{{ content|striptags }}`.
- `trim`: Removes whitespace from the start and end of a string `{{ content|trim }}`.
- `upper`: Transforms all of the letters in a string to uppercase `{{ content|upper }}`.
- `url_encode`: Translates a string as a URL segment or a query string. For example, `{{ "hello world"|url_encode }}` would result in `"hello%20world"`.

Twig Tests

Similar to `if` statements, Twig provides a number of functions that examine various attributes of a variable, array, or string:

- `constant`: Checks to see if a variable has the same value as a constant. For example, `{% if article.status is constant('Article::PUBLISHED') %}`.
- `defined`: Tests to see if the variable is defined in the current context. For example, `{% if user is defined %}`.
- `divisible`: Checks to see if a variable is divisible by a number. For example, `{% if company.members is divisible by (2) %}`.
- `empty`: Tests to see if a variable is empty. For example, `{% if order.number is empty %}`.
- `even`: Checks to see if a number is even. For example, `{% count is even %}`.
- `iterable`: Tests to see if a variable is an array or a transversable object. For example, `{% if orders is iterable %}`.
- `null`: Tests to see if a variable is null. For example, `{{ user is null }}`.
- `odd`: Tests to see if a number is odd. For example, `{{ count is odd }}z`.
- `sameas`: Checks to see if two variables are the same value and type.

Twig Math Functions

You can perform mathematical calculations on Twig variables using the same operators that are available in PHP. Examples include:

- Addition `{{ 1+1 }}`
- Subtraction `{{ 4-2 }}`
- Division `{{ 8/2 }}`
- Remainder `{{ 11 % 5 }}`
- Multiplication `{{ 5 * 5 }}`
- Exponentials `{{ 2 ** 3 }}` (2 to the power of 3)

There are other Twig features and functions that you can find at `twig.sensiolabs.org/documentation`.

Twig Template Files

Twig template files define the HTML markup, content, and CSS selectors that are used to render a field, taxonomy term, block, node, page, or the overall HTML of the site. Twig plays a key role in templates as it is the mechanism for rendering content, whether that content is defined as static text in the template or is generated by a module in Drupal, when the page is loaded.

Drupal 8 provides the ability to define templates for nearly every element that is rendered on a page, including the page itself. The structure of the templates is similar, the primary differences being the scope of the elements being rendered by the template. The specific elements that can be controlled through a template file are as follows:

- Fields, where each individual field rendered on a page may be customized using a field specific Twig template. Not all fields require specialized handling, and in this case, Drupal 8 provides a generic field template that your theme will use to render fields. We cover the naming convention of Twig templates and how the naming convention binds the template to a specific field.
- Taxonomy terms, where each individual taxonomy term may be customized using a taxonomy term specific Twig template. Like the field template, there is a generic taxonomy term template that Drupal will use if your theme does not provide one.
- Nodes, where a node, such as an article, may be customized to represent the layout and structure of your specific use cases through a node-specific Twig template. As with all Twig templates, a node template can be as specific as an individual node ID, across all nodes of a specific type (e.g., Article), or all nodes in general regardless of type or ID. As with other Twig templates, Drupal provides a generic template that will be used if your theme does not provide one.
- Blocks, where each block may be customized using a Twig template. Block templates, like other Twig templates, may be a specific as a single block, or generalized across all block on your site. As with other templates, Drupal provides a generic block template as part of Drupal 8 core.
- Regions. Regions are physical areas on the page where content may be placed. Regions, like other elements, may be customized through a Twig template, include the ability to develop a template for a specific region, or generally across all regions. Like other elements, Drupal 8 provides a generic region template that will be used if your theme does not provide one.

- Pages. The structure of a page may be customized through a page level Twig template. As with other elements, a specific page may be controlled through a page-specific template, or through a generic page template that applies to all pages on your site. Think of page as everything that falls in the `<body>` tag on a typical HTML page. Drupal core provides a generic page template that will be used when one that is applicable to the page being rendered is not found in your theme.
- HTML. This is the generalized template that provides the markup associated with HTML page level elements such as `<head>` and `<title>`.

There are specific naming conventions for template files in order to be identified by Drupal 8. Template files that do not follow the naming convention are ignored. The naming conventions for each type of template file are as follows:

- `html.html.twig`: The primary overarching template file that contains typical elements that would appear in the `<html>` and `<head>` section of a HTML page. For example, all of the CSS and JavaScript files that are loaded on a page that are global in nature.
- `page.html.twig`: The template file associated with the overall page. Think of this as the template that controls everything within the `<body>` tags on a typical HTML page. Page template files may be generic, such as `page.html.twig`, or specific to nodes (`page--node.html.twig`), a specific node (`page--node--1.html.twig`), or an action performed on a node (`page--node--edit.html.twig`).
- `region--[region].html.twig`: The naming convention for templates that are specific to a given region. In the case of the Davinci theme, candidates for region templates include `region--messages.html.twig`, `region--header_first.html.twig`, and `region--content.html.twig`.
- `block.html.twig`: The template file associated with all blocks on your site. You may create block specific templates through the naming convention of `block--module–delta.html.twig`, where `module` is the name of the module that is generating the block and `delta` represents the specific block that is being rendered. (For example, `block--test--news.html.twig` would control how the output generated by the block delta named `news` is generated in the module named `test`.) You may also provide Twig template files for blocks generated by views. As an example, a view named `featured_blogs` with a display ID of `block_1` would use the following name for the template file: `block--views--block--featured-blogs-block-1.html.twig`. Note the replacement of underscores with single dashes in the name of the template file.
- `node.html.twig`: The generic template that is applied to all nodes rendered on the site. There are several variants of the node template, including:
    - `node--viewmode.html.twig`. Simply replace `viewmode` with the appropriate value, such as `node--default.html.twig` or `node--teaser.html.twig`.
    - `node--type.html.twig`. Replace `type` with the content type that is being controlled by this template, such as `node--article.html.twig`.
    - `node--type--viewmode.html.twig`. Combining the previous two examples, the result would be `node--article--teaser.html.twig` or `node--article--default.html.twig`.

    - `node--nodeid.html.twig`: Refers to a specific node as defined through its `nodeid`, such as `node--1.html.twig`.
    - `node--nodeid--viewmode.html.twig`: As in previous examples, `node--1--teaser.html.twig` would refer to the node with an ID of 1 being rendered as a teaser would be controlled through this Twig template file.

- `taxonomy-term.html.twig`: Controls how all taxonomy terms on a site are displayed. More specific control is available through:
    - `taxonomy-term--vocabulary-machine-name.html.twig`, which controls the output of all terms in a specific taxonomy vocabulary.
    - `taxonomy-term--tid.html.twig`, which controls the output of a specific taxonomy term based on its `tid`.
- `field.html.twig:` Controls the output of all fields rendered on a site. To add specificity, you may use one of the following patterns:
    - `field--field-type.html.twig`, replacing `field-type` with, for example, `field--text-with-summary.html.twig`.
    - `field--field-name.html.twig`, replacing `field-name` with the name of the field, for example, `field--title.html.twig`.
    - `field--content-type.html.twig` would control all fields rendered on a content type, for example `field--article.html.twig`.
    - `field--field-name--content-type.html.twig` controls a specific field on a specific content type, for example `field--title--article.html.twig`.

There are other less widely used Twig templates, for example comments, comment wrappers, forums, and search results. Those naming conventions are as follows:

- `comment--node-[type].html.twig` controls how comments are rendered for a specific content type. For example, `comment--node-article.html.twig` would control how comments posted on all articles would be displayed.
- `comment-wrapper--node[type].html.twig` controls the format of the wrapper template for comments. For example, `comment-wrapper--article.html.twig`.
- `forums--[[container|topic]--forumID].html.twig` controls the output of forum containers and topics. Specific templates include:
    - `forums.html.twig` for the highest level and most generic theming across all forums on your site.
    - `forums--containers.html.twig` formats the containers defined in your forums on your site.
    - `forums--forumID.html.twig` formats a specific forum on your site.
    - `forums--containers--forumID.html.twig` formats a specific container on a specific forum on your site.
    - `forums--topics.html.twig` formats all topics across all forums on your site.
    - `forums--topics--forumID.html.twig` formats all topics for a specific formum on your site.

- `search-result.html.twig`: Formats the output of search results on your site. You may add more specific templates in the form of `search-result--node.html.twig` for node-based search results or `search-result--user.html.twig` for user-based search results.

With the list of possible template files defined in this list and a knowledge of Twig syntax, it's time to start examining the inner workings of templates.

Standard Twig Templates

Drupal 8 core comes with standard template files for all of the elements that can be rendered on a site using Drupal 8 core capabilities. This ensures that even if your theme doesn't provide a template file, Drupal knows how to render standard elements such a fields, blocks, nodes, breadcrumbs, forms, links, tables, pages, and other elements. You can find the standard templates in the `/core/modules/system/templates` directory. Take a few moments to navigate to the templates directory to see what is available. The standard templates are shown in Listing 5-2 for reference.

Listing 5-2. Drupal 8 Templates

```
├── admin-block-content.html.twig
├── admin-block.html.twig
├── admin-page.html.twig
├── authorize-report.html.twig
├── block--local-actions-block.html.twig
├── block--system-branding-block.html.twig
├── block--system-menu-block.html.twig
├── block--system-messages-block.html.twig
├── breadcrumb.html.twig
├── checkboxes.html.twig
├── confirm-form.html.twig
├── container.html.twig
├── datetime-form.html.twig
├── datetime-wrapper.html.twig
├── details.html.twig
├── dropbutton-wrapper.html.twig
├── entity-add-list.html.twig
├── feed-icon.html.twig
├── field-multiple-value-form.html.twig
├── field.html.twig
├── fieldset.html.twig
├── form-element-label.html.twig
├── form-element.html.twig
├── form.html.twig
├── html.html.twig
├── image.html.twig
├── indentation.html.twig
├── input.html.twig
├── install-page.html.twig
├── item-list.html.twig
├── links.html.twig
├── maintenance-page.html.twig
├── maintenance-task-list.html.twig
├── mark.html.twig
├── menu-local-action.html.twig
├── menu-local-task.html.twig
├── menu-local-tasks.html.twig
├── menu.html.twig
├── page-title.html.twig
├── page.html.twig
├── pager.html.twig
├── progress-bar.html.twig
├── radios.html.twig
├── region.html.twig
├── select.html.twig
├── status-messages.html.twig
├── status-report.html.twig
├── system-admin-index.html.twig
├── system-config-form.html.twig
├── system-modules-details.html.twig
├── system-modules-uninstall.html.twig
├── system-themes-page.html.twig
├── table.html.twig
├── tablesort-indicator.html.twig
├── textarea.html.twig
├── time.html.twig
└── vertical-tabs.html.twig
```

If you want to override the output of an element and you are creating a custom theme, you can copy the template file from `/core/modules/system/templates` to your `themes` template directory and modify the structure of the template file to meet your needs.

Modifying the page.html.twig Template File

One of the most common modifications to template files is to override the `page.html.twig` template file to meet a specific set of requirements and to render the regions defined in a theme. If we revisit the Davinci theme, we created a number of regions that currently aren't rendering on a page when we view, for example, the homepage of the Drupal 8 site. The reason they aren't rendering is that the `page.html.twig` file that Drupal 8 core provides in `/core/modules/system/templates` doesn't know that these regions exist. Let's fix that situation by copying the `page.html.twig` file from core and placing that copy in the Davinci theme's templates directory. Once it's been copied, open the `page.html.twig` file in your favorite editor to examine the contents of the file.

At the top of the file you will notice a large docblock that describes the variables that are available to the `page.html.twig` file, as defined by core. There are general informational variables, such as `base_path`, `is_front` and `front_page`, as well as content specific variables such as `node`. The default docblock also lists the regions that are available in Drupal 8 core.

```
{#
/**
 * @file
 * Default theme implementation to display a single page.
 *
 * The doctype, html, head and body tags are not in this template. Instead they
 * can be found in the html.html.twig template in this directory.
 *
 * Available variables:
 *
 * General utility variables:
 * - base_path: The base URL path of the Drupal installation. Will usually be
 *   "/" unless you have installed Drupal in a sub-directory.
 * - is_front: A flag indicating if the current page is the front page.
 * - logged_in: A flag indicating if the user is registered and signed in.
 * - is_admin: A flag indicating if the user has permission to access
 *   administration pages.
 *
 * Site identity:
 * - front_page: The URL of the front page. Use this instead of base_path when
 *   linking to the front page. This includes the language domain or prefix.
 *
 * Page content (in order of occurrence in the default page.html.twig):
 * - messages: Status and error messages. Should be displayed prominently.
 * - node: Fully loaded node, if there is an automatically-loaded node
 *   associated with the page and the node ID is the second argument in the
 *   page's path (e.g. node/12345 and node/12345/revisions, but not
 *   comment/reply/12345).
 *
 * Regions:
 * - page.header: Items for the header region.
 * - page.primary_menu: Items for the primary menu region.
 * - page.secondary_menu: Items for the secondary menu region.
 * - page.highlighted: Items for the highlighted content region.
 * - page.help: Dynamic help text, mostly for admin pages.
 * - page.content: The main content of the current page.
 * - page.sidebar_first: Items for the first sidebar.
 * - page.sidebar_second: Items for the second sidebar.
 * - page.footer: Items for the footer region.
 * - page.breadcrumb: Items for the breadcrumb region.
 *
 * @see template_preprocess_page()
 * @see html.html.twig
 *
 * @ingroup themeable
 */
#}
```

The first step in updating the `page.html.twig` file is to revise the list of regions that are available on the page to match the list of regions that were defined in the `davinci.info.yml` file. For reference, those regions are as follows:

```
regions:
  messages: 'Messages'
  header_first: 'Header first'
  header_second: 'Header second'
  header_third: 'Header third'
  navbar: 'Nav bar'
  help: 'help'
  features_first: 'Features first'
  features_second: 'Features second'
  features_third: 'Features third'
  features_fourth: 'Features fourth'
  highlighted: 'Highlighted'
  content: 'Main content'
  sidebar_first: 'Sidebar first'
  sidebar_second: 'Sidebar second'
  tertiary_first: 'Tertiary first'
  tertiary_second: 'Tertiary second'
  tertiary_third: 'Tertiary third'
  tertiary_fourth: 'Tertiary fourth'
  footer: 'Footer'
  page_top: 'Page top'
  page_bottom: 'Page bottom'
```

Using an editor, we update the docblock to reflect the revised list of regions, as shown here:

```
* Regions:
 * - page.message: The messages area.
 * - page.header_first: First header region.
 * - page.header_second: Second header region.
 * - page.header_third: Third header region.
 * - page.help: Help region
 * - page.features_first: First featured region.
 * - page.features_second: Second featured region.
 * - page.features_third: Third featured region.
 * - page.features_fourth: Fourth featured region.
 * - page.highlighted: Highlighted region.
 * - page.content: The main content of the current page.
 * - page.sidebar_first: Items for the first sidebar.
 * - page.sidebar_second: Items for the second sidebar.
 * - page.tertiary_first: First tertiary region.
 * - page.tertiary_second: Second tertiary region.
 * - page.tertiary_third: Third tertiary region.
 * - page.tertiary_fourth: Fourth tertiary region.
 * - page.footer: Items for the footer region.
```

Updating the docblock doesn't change the functionality of the template; it only provides reference to developers who may use this template in the future.

The next step is to update the template to render each of the regions that are currently not part of the existing `page.html.twig` template file. If you examine the markup and Twig elements of template file, you'll see HTML markup and Twig elements as described in the previous sections—for example, `{{ page.header }}`, which outputs everything assigned to the header region on the Admin ➤ Structure ➤ Block Layout page.

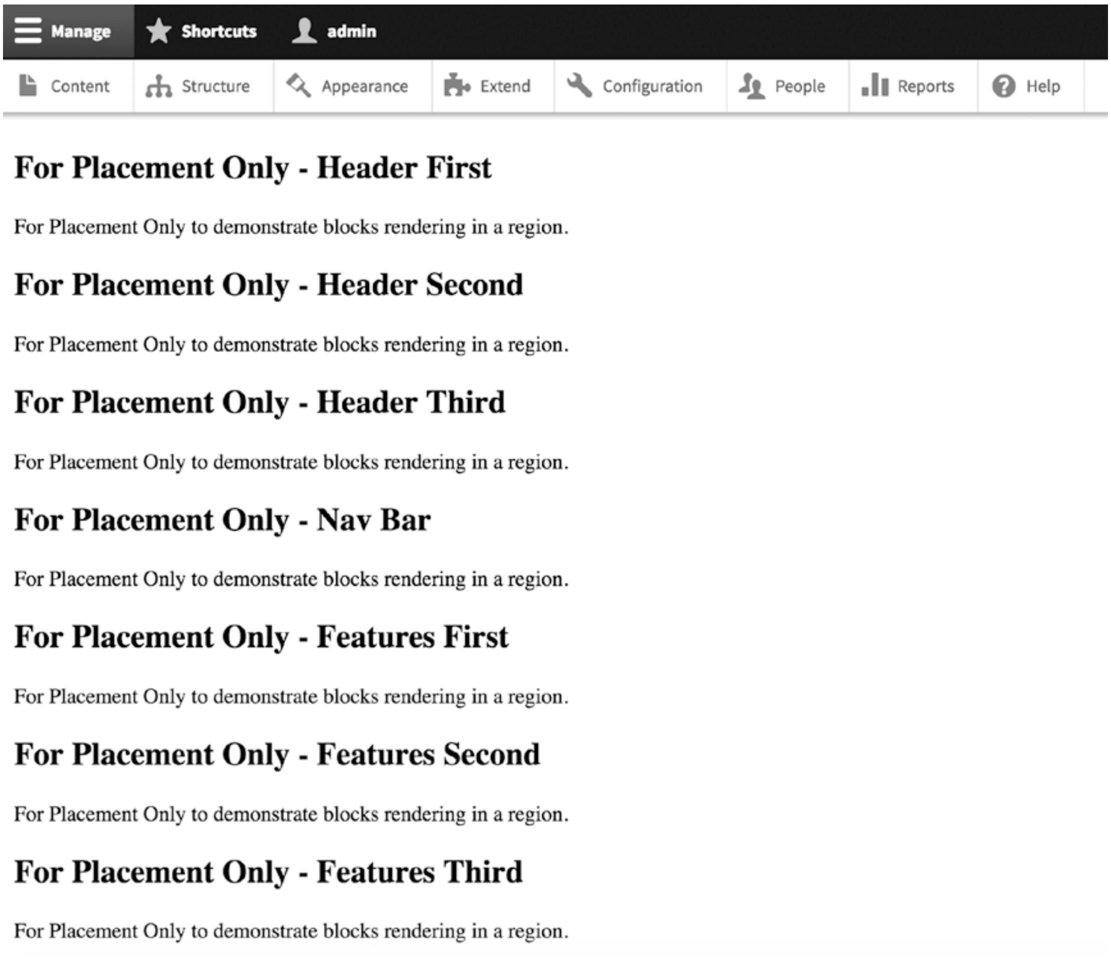
```
<div class="layout-container">

  <header role="banner">
    {{ page.header }}
  </header>

  {{ page.primary_menu }}
  {{ page.secondary_menu }}

  {{ page.breadcrumb }}

  {{ page.highlighted }}

  {{ page.help }}

  <main role="main">
    <a id="main-content" tabindex="-1"></a>{# link is in html.html.twig #}

    <div class="layout-content">
      {{ page.content }}
    </div>{# /.layout-content #}

    {% if page.sidebar_first %}
      <aside class="layout-sidebar-first" role="complementary">
        {{ page.sidebar_first }}
      </aside>
    {% endif %}

    {% if page.sidebar_second %}
      <aside class="layout-sidebar-second" role="complementary">
        {{ page.sidebar_second }}
      </aside>
    {% endif %}

  </main>

  {% if page.footer %}
    <footer role="contentinfo">
      {{ page.footer }}
    </footer>
  {% endif %}

</div>{# /.layout-container #}
```

We changed the names of some of the regions and added new regions that did not exist in the original template. To enable those regions so that they display, we need to edit the template and rename the regions we changed, and add the regions that do not exist in the off-the-shelf version of `page.html.twig`. This section assumes that you know HTML markup and CSS syntax and focuses on the Twig elements that need to be added to this page to fully render all of the regions.

After revising the template, the contents of the file are now set properly to render all of the regions that we defined in the `davinci.html.yml` file. As shown here, we added all of the regions by adding a `{{ page.region_name }}` Twig element, replacing `region_name` with the name of the regions as defined in the `davinci.html.twig` file. Note that the name used to replace `region_name` is the value for the region to the left of the `:` and not the description that appears to the right. For example, `{{ page.header_third }}` is the Twig representation of `header_third: 'Header third'` from the `davinci.html.yml` file. The updated template file, excluding the docblock, is as follows:

```
<div class="layout-container">

  <header role="banner">
    <div class="header_first">
      {{ page.header_first }}
    </div>
    <div class="header_second">
      {{ page.header_second }}
    </div>
    <div class="header_third">
      {{ page.header_third }}
    </div>
  </header>

  <div class="navbar">
    {{ page.navbar }}
  </div>

  {{ page.breadcrumb }}

  {{ page.help }}

  <div class="features">
    <div class="features_first">
       {{ page.features_first }}
    </div>
    <div class="features_second">
       {{ page.features_second }}
    </div>
    <div class="features_third">
       {{ page.features_third }}
    </div>
  </div>

  {{ page.highlighted }}

  <main role="main">
    <a id="main-content" tabindex="-1"></a>{# link is in html.html.twig #}

    <div class="layout-content">
      {{ page.content }}
    </div>{# /.layout-content #}
    {% if page.sidebar_first %}
      <aside class="layout-sidebar-first" role="complementary">
        {{ page.sidebar_first }}
      </aside>
    {% endif %}
    {% if page.sidebar_second %}
      <aside class="layout-sidebar-second" role="complementary">
        {{ page.sidebar_second }}
      </aside>
    {% endif %}

  </main>

  <div class="tertiary">
    {% if page.tertiary_first %}
      <div class="tertiary_first">
        {{ page.tertiary_first }}
      </div>
    {% endif %}
    {% if page.tertiary_second %}
      <div class="tertiary_second">
        {{ page.tertiary_second }}
      </div>
    {% endif %}
    {% if page.tertiary_third %}
      <div class="tertiary_third">
        {{ page.tertiary_third }}
      </div>
    {% endif %}
  </div>

  {% if page.footer %}
    <footer role="contentinfo">
      {{ page.footer }}
    </footer>
  {% endif %}

</div>{# /.layout-container #}
```

After examining the template file, you'll see that the primary changes were the addition of a few CSS classes, a few additional conditionals check to see if a region has a value before rendering it on the page, and the Twig statements to render the additional regions. After saving the template and rebuilding cache, we add a sample block to each of the regions on the theme and test to see if all the regions appear on the homepage (see Figure 5-5).



For Placement Only - Header First

For Placement Only to demonstrate blocks rendering in a region.

For Placement Only - Header Second

For Placement Only to demonstrate blocks rendering in a region.

For Placement Only - Header Third

For Placement Only to demonstrate blocks rendering in a region.

For Placement Only - Nav Bar

For Placement Only to demonstrate blocks rendering in a region.

For Placement Only - Features First

For Placement Only to demonstrate blocks rendering in a region.

For Placement Only - Features Second

For Placement Only to demonstrate blocks rendering in a region.

For Placement Only - Features Third

For Placement Only to demonstrate blocks rendering in a region.

Figure 5-5. *A partial listing of the new regions on the homepage*

While they are stacked on top of each other due to the fact that we haven't applied any CSS formatting to the regions, the updates to the `page.html.twig` template demonstrate that the new regions are rendering content on the homepage. The next step is to apply CSS to classes that we added to format the page as desired. We'll cover adding CSS in a moment, but next let's look at two of the other common templates that most developers want to modify on their sites—the node and block templates.

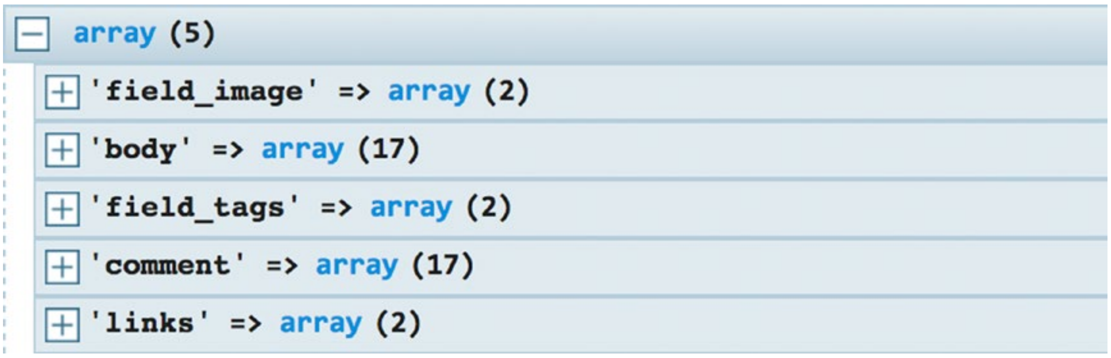
Modifying the node.html.twig Template

Following the same approach as for the page template, copy the default `node.html.twig` template from `core/modules/node/templates/node.html.twig` to my `themes/custom/davinci/templates` directory. Note that the node template is found in the `core/modules/node` directory and not the `core/modules/system` directory, as the node module is the one responsible for theming nodes.

After copying the template file, examine the docblock and look at the variables that are available for you to use in your `node.html.twig` template file. Most of the variables are self-explanatory. For example, `url` provides the value of the URL used to access the node being displayed. But there are others that are not quite so obvious, such as `content` and `attributes`.

Displaying and Hiding Content Fields

Content is the object that contains all of a node's fields that are to be rendered given a specific display mode (e.g., teaser or default). Using `{{ content }}` renders every field that is associated with the current display mode. You may encounter scenarios where you need to render specific fields, not every field associated with content. To render specific fields, use `{{ content.field_name }}` and replace `field_name` with the appropriate field. For example, `{{ content.body }}`. To find the list of fields that are available, visit the Structure ➤ Content Types and click the Manage Display link in the Operations column for the content type that you are working with. Select the display mode that you are working with to view the list of fields. Alternatively, assuming you have debugging turned on and the Devel module and Devel Kint enabled (`drupal.org/project/devel`), you may use `{{ kint(content) }}` in your template file to print a list of all the fields associated with the content object, as shown in Figure 5-6.



```
array (5)
  'field_image' => array (2)
  'body' => array (17)
  'field_tags' => array (2)
  'comment' => array (17)
  'links' => array (2)
```

Figure 5-6. *The Content object displayed by kint*

You may also exclude fields from being displayed using for example `{{ content|without ('comment') }}`. This would render content without the comment field. You may also exclude multiple fields using `{{ content|without('comment', 'field_tags') }}`. This becomes useful when you need to display certain fields from the content object wrapped in specific CSS; for example, you want comments to appear in a column to the right of the body and not below the body. You could accomplish this by using the following:

```
<div class='body-only'>
  {{ content|without('comment') }}
</div>
<div class='comments-only'>
  {{ content.comment}}
</div>
```

Using Attributes

Drupal 8 provides a standard means for generating CSS classes for divs using an attribute array. The base attribute array is created in /core/includes/theme.inc in a template preprocess function (more on preprocess functions later in this chapter):

```
$variables = array(
    'attributes' => array(),
    'title_attributes' => array(),
    'content_attributes' => array(),
    'title_prefix' => array(),
    'title_suffix' => array(),
    'db_is_active' => !defined('MAINTENANCE_MODE'),
    'is_admin' => FALSE,
    'logged_in' => FALSE,
  );
```

The attributes array is a common placeholder for all CSS class definitions that may be applied to a div in your template. For example, you may have a common set of classes that you want to apply to several divs. Instead of entering this over and over again:

```
<div class='red white blue'>
```

The preferred way is to add red white blue to the attributes array and then use this:

```
<div{{ attributes }}>
```

Note that in theme.inc there are multiple attributes that may be augmented for your specific needs, attributes, title_attributes, and content_attributes. You may also create your own additions by creating new variables, and many modules do just that.

Adding classes to the attributes arrays is relatively simple. In the previous example—using red white blue as representative CSS classes that we want to add to several divs—we would use the following:

```
<div{{ attributes.addClass('red white blue') }}>
```

From that point forward, any time that we use `<div{{ attributes }}>`, we will automatically get red white blue added as classes to the divs. You may also remove classes from the attributes array using attributes.removeClass('blue'), replacing blue with the class that you want to remove. Removing a class removes it permanently from the attributes array, meaning it will no longer be applied in future uses of {{ attributes }}.

You may also add attributes—for example, you can create an attribute named id and assign a value to it using attributes.setAttribue(attribute, value).

As mentioned previously, attributes isn't the only array available for assigning classes. You may use the off-the-shelf title_attributes array to add CSS classes to titles and then render titles using, for example, `<h1{{ title_attributes }}>`. You may also use content_attributes, title_prefix, title_suffix, or any other attributes that you define using preprocess functions.

Modifying the block.html.twig Template

The block module provides a base template that will be used by default if your theme does not provide a specific `block.html.twig` template. To override the block template, copy `block.html.twig` from `/core/modules/block/templates` to your `themes` templates directory and examine the contents of that template file. As with other core templates, the docblock describes the variables that are presented by the block module for use in your template file.

```
/**
 * @file
 * Default theme implementation to display a block.
 *
 * Available variables:
 * - plugin_id: The ID of the block implementation.
 * - label: The configured label of the block if visible.
 * - configuration: A list of the block's configuration values.
 *   - label: The configured label for the block.
 *   - label_display: The display settings for the label.
 *   - provider: The module or other provider that provided this block plugin.
 *   - Block plugin specific settings will also be stored here.
 * - content: The content of this block.
 * - attributes: array of HTML attributes populated by modules, intended to
 *   be added to the main container tag of this template.
 *   - id: A valid HTML ID and guaranteed unique.
 * - title_attributes: Same as attributes, except applied to the main title
 *   tag that appears in the template.
 * - title_prefix: Additional output populated by modules, intended to be
 *   displayed in front of the main title tag that appears in the template.
 * - title_suffix: Additional output populated by modules, intended to be
 *   displayed after the main title tag that appears in the template.
 *
 * @see template_preprocess_block()
 *
 * @ingroup themeable
 */
```

Significantly simpler than the node template, the block template really only has two variables that are rendered—the title (which is called `label`) and the body of the block, which is called `content`.

```
<div{{ attributes }}>
  {{ title_prefix }}
  {% if label %}
    <h2{{ title_attributes }}>{{ label }}</h2>
  {% endif %}
  {{ title_suffix }}
  {% block content %}
    {{ content }}
  {% endblock %}
</div>
```

As you examine this code, you'll see the patterns that we discussed previously in this chapter. The template renders the values in the attributes array, if there are any, as CSS classes in the opening `div`. It renders the value stored in `title_prefix` and then checks to see if the label has a value. If it does, it renders the label as an H2 using the CSS classes defined in the `title_attributes` array. It then renders the values of `title_suffix` and then renders the content associated with the block.

■ **Note** The statement `{% block content %}` and its paired `{% endblock %}` should not be confused with the Drupal Block module. The block in this case specifies a code block and has nothing to do with the Block module. It is just an unfortunate collision of common terms.

While there isn't much to the block template, there may be scenarios where you have added custom fields to a block and you need to render those fields in a specific `div`, or there may be other unique scenarios where you need to override the default behavior of the core `block.html.twig` template. Follow the patterns and principles described elsewhere in this chapter. You are on your way to customizing block output.

Modifying the field.html.twig Template

The last template file that we examine is the field template. In most cases you won't override the generic field template but rather you will create a template file for a specific field. As mentioned earlier in this chapter, template files may be created for specific pages, nodes, blocks, and fields by simply following the naming patters associated with that template file to apply a specific template file to a specific element.

In the case of the field templates, the naming conventions are as follows:

- `field--field-type.html.twig`, replacing `field-type` with, for example, `field--text-with-summary.html.twig`.
- `field--field-name.html.twig`, replacing `field-name` with the name of the field, for example, `field--title.html.twig`.
- `field--content-type.html.twig` controls all fields rendered on a content type, for example, `field--article.html.twig`.
- `field--field-name--content-type.html.twig` controls a specific field on a specific content type, for example `field--title--article.html.twig`.

To demonstrate the capabilities of targeting a specific field, we'll create a template for the body field that will wrap the body with additional classes regardless of where it is rendered on the site. To do so, we copy the default template from `core/modules/system/templates/field.html.twig` to the Davinci templates directory and name it `field--body.html.twig`.

The docblock at the top of the template file lists all of the variables that are available for use in this template:

```
* Available variables:
 * - attributes: HTML attributes for the containing element.
 * - label_hidden: Whether to show the field label or not.
 * - title_attributes: HTML attributes for the title.
 * - label: The label for the field.
 * - multiple: TRUE if a field can contain multiple items.
 * - items: List of all the field items. Each item contains:
 *   - attributes: List of HTML attributes for each item.
 *   - content: The field item's content.
 * - entity_type: The entity type to which the field belongs.
 * - field_name: The name of the field.
 * - field_type: The type of the field.
 * - label_display: The display settings for the label.
```

The standard template renders those fields using the following structure:

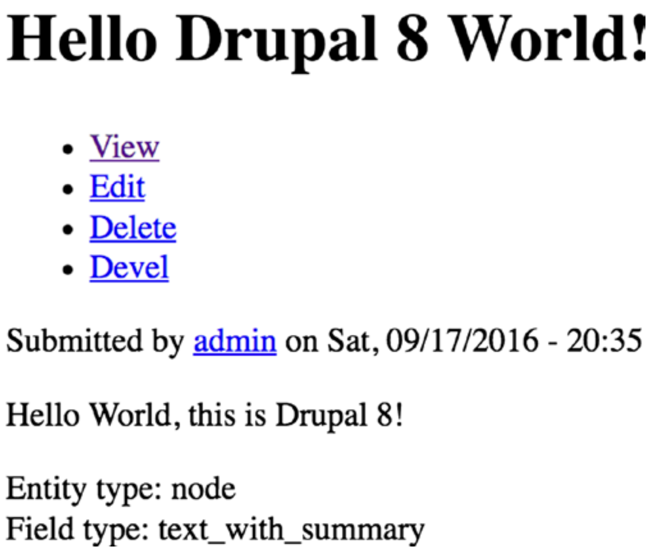
```
{% if label_hidden %}
  {% if multiple %}
    <div{{ attributes }}>
      {% for item in items %}
        <div{{ item.attributes }}>{{ item.content }}</div>
      {% endfor %}
    </div>
  {% else %}
    {% for item in items %}
      <div{{ attributes }}>{{ item.content }}</div>
    {% endfor %}
  {% endif %}
{% else %}
  <div{{ attributes }}>
    <div{{ title_attributes }}>{{ label }}</div>
    {% if multiple %}
      <div>
    {% endif %}
    {% for item in items %}
      <div{{ item.attributes }}>{{ item.content }}</div>
    {% endfor %}
    {% if multiple %}
      </div>
    {% endif %}
  </div>
{% endif %}
```

Using the knowledge of Twig gained throughout this chapter, you can see that the template renders one-to-many instances of a field, displaying the contents of the field and the label associated with that field. Since Drupal provides the ability to create one-to-many instances of field content for a given node, the template addresses this scenario through the `for` items in items loops, displaying each instance of the field.

For demonstration purposes, we are going to update the body template to do one thing, display the `entity_type` and `field_type`. While likely not a high value change to the template, it does demonstrate the ability to override a specific field. Since we can never be certain whether a content type may be set up to accept more than one instance of the body field, we update the template in multiple places just to ensure that we catch every scenario. After the updates, the template appears as shown here. Note that we've added `{{ entity_type }}` and `{{ field_type }}` to the template file in multiple places.

```
{% if label_hidden %}
  {% if multiple %}
    <div{{ attributes }}>
      {% for item in items %}
        <div{{ item.attributes }}>{{ item.content }}</div>
      {% endfor %}
    </div>
  {% else %}
    {% for item in items %}
      <div{{ attributes }}>{{ item.content }}</div>
        <div>
          Entity type: {{ entity_type }}
        </div>
        <div>
          Field type: {{ field_type }}
        </div>
    {% endfor %}
  {% endif %}
{% else %}
  <div{{ attributes }}>
    <div{{ title_attributes }}>{{ label }}</div>
    {% if multiple %}
      <div>
    {% endif %}
    {% for item in items %}
      <div{{ item.attributes }}>{{ item.content }}</div>
    {% endfor %}
    {% if multiple %}
      </div>
    {% endif %}
    <div>
      Entity type: {{ entity_type }}
    </div>
    <div>
      Field type: {{ field_type }}
    </div>
  </div>
{% endif %}
```

After saving the template and rebuilding the cache, the resulting output is as shown in Figure 5-7.


Hello Drupal 8 World!

- View
- Edit
- Delete
- Devel

Submitted by admin on Sat, 09/17/2016 - 20:35

Hello World, this is Drupal 8!

Entity type: node
Field type: text_with_summary

Figure 5-7. *The addition of the entity_type and field_type output to the field template*

There are virtually limitless things you can do with template files. The limitations are a) can it be done with Twig and b) does the module that generates the output provide access to those values through a variable that is accessible to Twig? These issues are the topic of the next section.

Exposing Variables to Twig

For most modules, one of the primary objectives is to generate output that can be rendered on a page through a Twig template. Facilitating that process requires a few simple steps, including the ability to define what template files your module uses to render that content. To demonstrate how the connection between a module and a Twig template works, we'll use Drupal 8's Forum module as the example.

Navigate to `/core/modules/forum` and open the `forum.module` file with your favorite editor. Search for `hook_theme` and you'll find the following function.

```
/**
 * Implements hook_theme().
 */
function forum_theme() {
  return array(
    'forums' => array(
       'variables' => array('forums' => array(), 'topics' => array(), 'topics_pager' =>
       array(), 'parents' => NULL, 'term' => NULL, 'sortby' => NULL, 'forum_per_page' =>
       NULL, 'header' => array()),
    ),
    'forum_list' => array(
      'variables' => array('forums' => NULL, 'parents' => NULL, 'tid' => NULL),
    ),
    'forum_icon' => array(
      'variables' => array('new_posts' => NULL, 'num_posts' => 0, 'comment_mode' => 0,
      'sticky' => 0, 'first_new' => FALSE),
    ),
    'forum_submitted' => array(
      'variables' => array('topic' => NULL),
    ),
  );
}
```

The forum_theme function returns an array of various elements. We focus on the first element of the array named forums. You can see that the forums array includes another array named variables, and within that array you will find additional arrays named forums, topics, topics_pager, parents, term, sortby, forum_per_page, and header. Each of those variables is registered with the Twig theme engine and is available to the template files that are used by the Forum module.

Those variable, now registered with the theme engine, are ready to receive values through a preprocess function. In the case of the Forum module, that preprocess function is named template_preprocess_forums. In this function, the output generated by the module is assigned to the variables that were defined in the forum_theme function. Once they are populated, they are ready to render through a template file.

Search the forum.module for template_process_forums and you'll find the function that is listed here. The first section is the docblock that specifies which template file is used to render a forum and the variables that are populated and exposed to the template file. Again, those variables were originally defined and registered in the forum_theme function. While somewhat complex, the function demonstrates one simple concept—place values into the various variables that are registered by forum_theme and exposed to the template.

```
/**
 * Prepares variables for forums templates.
 *
 * Default template: forums.html.twig.
 *
 * @param array $variables
 *   An array containing the following elements:
 *   - forums: An array of all forum objects to display for the given taxonomy
 *     term ID. If tid = 0 then all the top-level forums are displayed.
 *   - topics: An array of all the topics in the current forum.
 *   - parents: An array of taxonomy term objects that are ancestors of the
 *     current term ID.
 *   - term: Taxonomy term of the current forum.
 *   - sortby: One of the following integers indicating the sort criteria:
 *     - 1: Date - newest first.
 *     - 2: Date - oldest first.
 *     - 3: Posts with the most comments first.
 *     - 4: Posts with the least comments first.
 *   - forum_per_page: The maximum number of topics to display per page.
 */
function template_preprocess_forums(&$variables) {
  $variables['tid'] = $variables['term']->id();
  if ($variables['forums_defined'] = count($variables['forums']) ||
  count($variables['parents'])) {
```

```
if (!empty($variables['forums'])) {
  $variables['forums'] = array(
    '#theme' => 'forum_list',
    '#forums' => $variables['forums'],
    '#parents' => $variables['parents'],
    '#tid' => $variables['tid'],
  );
}

if ($variables['term'] && empty($variables['term']->forum_container->value) &&
!empty($variables['topics'])) {
  $forum_topic_list_header = $variables['header'];

  $table = array(
    '#theme' => 'table__forum_topic_list',
    '#responsive' => FALSE,
    '#attributes' => array('id' => 'forum-topic-' . $variables['tid']),
    '#header' => array(),
    '#rows' => array(),
  );

  if (!empty($forum_topic_list_header)) {
    $table['#header'] = $forum_topic_list_header;
  }

  /** @var \Drupal\node\NodeInterface $topic */
  foreach ($variables['topics'] as $id => $topic) {
    $variables['topics'][$id]->icon = array(
      '#theme' => 'forum_icon',
      '#new_posts' => $topic->new,
      '#num_posts' => $topic->comment_count,
      '#comment_mode' => $topic->comment_mode,
      '#sticky' => $topic->isSticky(),
      '#first_new' => $topic->first_new,
    );

    // We keep the actual tid in forum table, if it's different from the
    // current tid then it means the topic appears in two forums, one of
    // them is a shadow copy.
    if ($variables['tid'] != $topic->forum_tid) {
      $variables['topics'][$id]->moved = TRUE;
      $variables['topics'][$id]->title = $topic->getTitle();
      $variables['topics'][$id]->message = \Drupal::l(t('This topic has been moved'),
      new Url('forum.page', ['taxonomy_term' => $topic->forum_tid]));
    }
    else {
      $variables['topics'][$id]->moved = FALSE;
      $variables['topics'][$id]->title_link = \Drupal::l($topic->getTitle(),
      $topic->urlInfo());
      $variables['topics'][$id]->message = '';
    }
```

```
        $forum_submitted = array('#theme' => 'forum_submitted', '#topic' => (object) array(
          'uid' => $topic->getOwnerId(),
          'name' => $topic->getOwner()->getDisplayName(),
          'created' => $topic->getCreatedTime(),
        ));
        $variables['topics'][$id]->submitted = drupal_render($forum_submitted);
        $forum_submitted = array(
          '#theme' => 'forum_submitted',
          '#topic' => isset($topic->last_reply) ? $topic->last_reply : NULL,
        );
        $variables['topics'][$id]->last_reply = drupal_render($forum_submitted);

        $variables['topics'][$id]->new_text = '';
        $variables['topics'][$id]->new_url = '';

        if ($topic->new_replies) {
          $page_number = \Drupal::entityManager()->getStorage('comment')
            ->getNewCommentPageNumber($topic->comment_count, $topic->new_replies, $topic,
            'comment_forum');
          $query = $page_number ? array('page' => $page_number) : NULL;
          $variables['topics'][$id]->new_text = \Drupal::translation()->formatPlural
          ($topic->new_replies, '1 new post<span class="visually-hidden"> in topic %title
          </span>', '@count new posts<span class="visually-hidden"> in topic %title</span>',
          array('%title' => $variables['topics'][$id]->label()));
          $variables['topics'][$id]->new_url = \Drupal::url('entity.node.canonical', ['node'
          => $topic->id()], ['query' => $query, 'fragment' => 'new']);
        }

        // Build table rows from topics.
        $row = array();
        $row[] = array(
          'data' => array(
            $topic->icon,
            array(
              '#markup' => '<div class="forum__title"><div>' . $topic->title_link .
              '</div><div>' . $topic->submitted . '</div></div>',
            ),
          ),
          'class' => array('forum__topic'),
        );

        if ($topic->moved) {
          $row[] = array(
            'data' => $topic->message,
            'colspan' => '2',
          );
        }
        else {
          $new_replies = '';
          if ($topic->new_replies) {
            $new_replies = '<br /><a href="' . $topic->new_url . '">' . $topic->new_text . '</a>';
          }
          $row[] = array(
            'data' => [
              [
                '#prefix' => $topic->comment_count,
                '#markup' => $new_replies,
              ],
            ],
            'class' => array('forum__replies'),
          );
          $row[] = array(
            'data' => $topic->last_reply,
            'class' => array('forum__last-reply'),
          );
        }
        $table['#rows'][] = $row;
      }

      $variables['topics'] = $table;
      $variables['topics_pager'] = array(
        '#type' => 'pager',
      );
    }
  }
}
```

With the variables defined and populated with information that is ready to render through a template file, the next step is to define what template files should be used to render the variables. This is the job of the theme suggestion's hook. Within the `forum.module` file, search for `forum_theme_suggestions_forums` and you'll find the function that defines the various suggestions for the name of the theme file that Twig should use to render the output generated by the Forum module.

```
/**
 * Implements hook_theme_suggestions_HOOK().
 */
function forum_theme_suggestions_forums(array $variables) {
  $suggestions = array();
  $tid = $variables['term']->id();

  // Provide separate template suggestions based on what's being output. Topic
  // ID is also accounted for. Check both variables to be safe then the inverse.
  // Forums with topic IDs take precedence.
  if ($variables['forums'] && !$variables['topics']) {
    $suggestions[] = 'forums__containers';
    $suggestions[] = 'forums__' . $tid;
    $suggestions[] = 'forums__containers__' . $tid;
  }
  elseif (!$variables['forums'] && $variables['topics']) {
    $suggestions[] = 'forums__topics';
    $suggestions[] = 'forums__' . $tid;
    $suggestions[] = 'forums__topics__' . $tid;
  }
  else {
    $suggestions[] = 'forums__' . $tid;
  }

  return $suggestions;
}
```

The order of the suggestions defined in the function is important, as that is the order of precedence that Twig uses when searching for an applicable template to render what is being sent to the theme layer to render on the page. As shown, the `$suggestions` array is populated with various options based on whether the forum is associated with a taxonomy term ID or not, and moves from general templates applied to all forums, `forums__containers`, to specific containers within a term ID, `'forums__containers__'.$tid`.

Examining the template files completes the picture of how values from a module are rendered on a page. Let's pick the simplest template, `forums.html.twig`. In this template file, you'll find three simple output statements—`{{ forums }}`, `{{ topics }}`, and `{{ topics_pager }}`.

```
{#
/**
 * @file
 * Default theme implementation to display a forum.
 *
 * May contain forum containers as well as forum topics.
 *
 * Available variables:
 * - forums: The forums to display (as processed by forum-list.html.twig).
 * - topics: The topics to display.
 * - topics_pager: The topics pager.
 * - forums_defined: A flag to indicate that the forums are configured.
 *
 * @see template_preprocess_forums()
 *
 * @ingroup themeable
 */
#}
{% if forums_defined %}
  {{ forums }}
  {{ topics }}
  {{ topics_pager }}
{% endif %}
```

If you trace all the way back to the `forum_theme` function, you'll see that those variables were defined here:

```
'variables' => array(
    'forums' => array(),
    'topics' => array(),
    'topics_pager' => array(),
    'parents' => NULL,
    'term' => NULL,
    'sortby' => NULL,
    'forum_per_page' => NULL,
    'header' => array())
```

And were populated in the preprocess function:

```
$variables['forums'] = array(
  '#theme' => 'forum_list',
  '#forums' => $variables['forums'],
  '#parents' => $variables['parents'],
  '#tid' => $variables['tid'],
);
$variables['topics'] = $table;
$variables['topics_pager'] = array(
  '#type' => 'pager',
);
```

While there are many pieces to the overall solution for how content makes it way to the physical page, the process is relatively simple and the patterns are well defined. For additional details on preprocess functions and theme hooks, visit drupal.org/docs/8.

Applying CSS to Your Theme

With all of the non-styling related pieces in place, the next step in beautifying your theme is to apply styling to the CSS elements defined in your template files. There are three steps in Drupal 8 for creating CSS:

1. Create the stylesheet(s) that will be loaded and used to render elements as you want them to appear.
2. Instruct the theme layer how to find the CSS files that you created.
3. Update the .info.yaml file.

Creating the Stylesheets

Stylesheets in Drupal 8 are stored in the theme's directory following a scalable and modular architecture for CSS (SMACSS) style categorization of its CSS rules (visit smacss.com for a complete overview of SMACSS):

- Base: CSS reset/normalize plus HTML element styling
- Layout: Macro arrangement of a web page, including any grid systems
- Component: Discrete, reusable UI elements
- State: Styles that deal with client-side changes to components
- Theme: Purely visual styling ("look-and-feel") for a component

Following SMACSS, the directory structure in the theme's directory where stylesheets are stored is as follows:

```
├── css
│   ├── base
│   ├── component
│   ├── layout
│   ├── state
│   └── theme
```

Since the sample stylesheet deals with general look-and-feel, we'll create a file named `styles.css` in the `css/theme` directory using any editor. In that file, we enter the following and then save the file.

```
h1 {
  text-transform: uppercase;
  text-decoration: underline;
}
```

We could at this point continue to build out other stylesheets following the SMACSS guidelines to address all of the styling requirements for the theme. However, to keep the concept simple, we'll next focus on creating the libraries file.

Creating the libraries.yml File

The next step is to create a `.libraries.yml` file that's used by Drupal to identify all of the CSS and JavaScript files that are associated with my theme. The library file for the Davinci theme will at this point only define the global stylesheets that are to be loaded. The library file will be expanded later in this chapter to address additional requirements. You can create the `davinci.libraries.yml` file in the root directory of the Davinci theme using any editor. In that file, enter the following:

```
global-styling:
  version: 1.x
  css:
    theme:
      css/theme/styles.css: {}
```

In the file, the `global-styling` is a unique library that will be loaded when instructed to do so in the `davinci.info.yml` file. Associated with `global-styling` are various attributes such as the version number of this library, and in this case, the CSS that is going to be used and applied to the theme and where it resides in relation to the theme, in the `css/theme` subdirectory. The structure of the .yml file is important, so be mindful of the indentation, as spacing means something to the yaml parser.

Loading the Libraries Through the .info.yml File

After saving the libraries file, the final step is to instruct the theme to load the `global-styling` library. Open the `davinci.info.yml` file and update the file to appear as shown here. Note the addition of the `libraries` statement and, below that, statement the instruction to load the library associated with the Davinci theme named `global-styling`, which is what we just created in the `davinci.libraries.yml` file in the previous step.

```
name: davinci
type: theme
description: A Drupal 8 theme
core: 8.x
libraries:
 - davinci/global-styling
regions:
  messages: 'Messages'
  header_first: 'Header first'
  header_second: 'Header second'
  header_third: 'Header third'
  navbar: 'Nav bar'
  features_first: 'Features first'
  features_second: 'Features second'
  features_third: 'Features third'
  features_fourth: 'Features fourth'
  highlighted: 'Highlighted'
  content: 'Main content'
  sidebar_first: 'Sidebar first'
  sidebar_second: 'Sidebar second'
  tertiary_first: 'Tertiary first'
  tertiary_second: 'Tertiary second'
  tertiary_third: 'Tertiary third'
  tertiary_fourth: 'Tertiary fourth'
  footer: 'Footer'
  page_top: 'Page top'
  page_bottom: 'Page bottom'
```

After saving all the files and rebuilding cache, you're ready to see if the CSS changes took effect. The results are shown in Figure 5-8.

Figure 5-8. *Demonstrating the successful application of CSS*

Success! The demonstration shows that the CSS file was successfully loaded and applied to a node's `H1` title. You can expand on this by adding stylesheets to the `global-styling` library, or you can create additional libraries outside of `global-styling`, by adding the libraries through the `.info.yml` file. For additional details, visit `drupal.org/node/2216195`.

Adding JavaScript to Your Theme

Adding JavaScript to your theme is nearly identical to the process of adding stylesheets to the theme. It's done through the `.libraries.yml` file and updates to the `.info.yml` file. We'll create a simple JavaScript file named `annoying-cow.js` that does one thing—it generates an alert box that says "Moo!" Being an annoying cow, it will pop up on every page of the site. We will create the JavaScript file in the `js` subdirectory of the Davinci theme. The content of the JavaScript file is simply:

```
(function ($, Drupal) {
  alert("Moo!")
})(jQuery, Drupal);
```

After you save the file, the directory structure of the Davinci theme appears as shown here:

```
├── config
│   ├── install
│   └── schema
├── css
│   ├── base
│   ├── component
│   ├── layout
│   ├── state
│   └── theme
│       └── styles.css
├── davinci.info.yml
├── davinci.libraries.yml
├── images
├── js
│   └── annoying-cow.js
└── templates
    ├── block.html.twig
    ├── field--body.html.twig
    ├── node.html.twig
    └── page.html.twig
```

Next we will define a library in the `davinci.libraries.yml` file that will be used to load the `annoying-cow.js` file. Edit the existing file by adding the `annoying-cow` library to the bottom of the file, as shown here:

```
global-styling:
  version: 1.x
  css:
    theme:
      css/theme/styles.css: {}

annoying-cow:
  version: 1.x
  js:
    js/annoying-cow.js: {}
```

Finally, we update the `libraries` section of the `davinci.info.yml` file to load the `annoying-cow` library.

```
libraries:
 - davinci/global-styling
 - davinci/annoying-cow
```

After saving all the files and rebuilding cache, reload the homepage. The annoying cow is shown in Figure 5-9.

Figure 5-9. *Demonstrating that the added JavaScript works*

Adding JavaScript and CSS Libraries to Template Files

In the previous examples, we added CSS and JavaScript to the Davinci module on a global basis, meaning that every page will load the CSS and the JavaScript that we added through the libraries section of the `davinci.info.yml` file. This approach works but isn't the optimal solution, as not all CSS or JavaScript is required on every page.

To resolve the issue, Twig provides the ability to load libraries on a per-template basis through the `attach_library` function in the form of `{{ attach_library('library_name') }}`. In the case of the Davinci theme, we could add the `node.css` file, which is loaded only when the `node.html.twig` template is used to render a node. We first need to add a new library to the `davinci.libraries.yml` file to define the library:

```
node-styling:
  version: 1.x
  css:
    theme:
      css/theme/node.css: {}
```

After creating the new library, `node-styling`, we can then add that stylesheet to the `node.html.twig` template by inserting `{{ attach_library('davinci/node-styling') }}` at the top of the template (after the docblock). Save all the files and rebuild the cache. At that point, the CSS defined in `node.css` is now applied to all nodes that are rendered on the site, and not the anything other than a node.

The same concept can be applied to JavaScript. You may load JavaScript on an as-needed basis following the same approach outlined for CSS.

Working with Breakpoints

With responsive themes being the default standard for most organizations, any theme that you create should take advantage of breakpoints in Drupal 8 themes. The Breakpoint module is part of Drupal 8 core and is enabled by default when you install Drupal 8 (other than the minimal installation profile). The Breakpoint module keeps track of the height, width, and resolution of the device viewport where the site is being rendered. Themes and modules can define breakpoints and then use them to define how the elements on the page and the page itself are rendered on different devices.

A *breakpoint* is a label and a media query where the label is a human readable label for the breakpoint that is being defined, and the media query is the element that defines what the breakpoint applies to. For example, a breakpoint for mobile devices as defined in the Davinci theme might look like the following:

```
davinci.mobile:
  label: mobile
  mediaQuery: 'all and (max-width: 559px)'
  weight: 0
  multipliers: 1x
```

In this case, the breakpoint of `mobile` will be applied to all device viewports that have maximum display widths of 559px. Anything greater than 559px will not be considered a mobile device in the case of the Davinci theme. The `weight` defines the order of precedence and the multipliers will be defined later in this section.

Breakpoints are defined in a `.breakpoints.yml` file that is stored in the theme's root directory. Let's create a `davinci.breakpoints.yml` file with the following content:

```
davinci.mobile:
  label: mobile
  mediaQuery: 'all and (max-width: 559px)'
  weight: 0
  multipliers:
    - 1x
davinci.narrow:
  label: narrow
  mediaQuery: 'all and (min-width: 560px) and (max-width: 850px)'
  weight: 1
  multipliers:
    - 1x
davinci.wide:
  label: wide
  mediaQuery: 'all and (min-width: 851px)'
  weight: 2
  multipliers:
    - 1x
```

Multipliers are a measure of the viewport's device resolution, defined as that ratio between the physical pixel size of the active device and the device-independent pixel size. For example, Apple's retina displays have a multiplier of 2.x. By adding a new breakpoint of `davinci.mobile_retina` you can address the specific CSS attributes required to render the site properly on a retina display:

```
davinci.mobile_retina:
  label: mobile_retina
  mediaQuery: 'all and (max-width: 559px)'
  weight: 0
  multipliers:
  - 2x
```

While the definition of breakpoints in the `.breakpoints.yml` file, they currently have no impact on the CSS that you write and the media queries will also need to be added to your CSS files as they would without the breakpoints module. At some point in the future, a bridge between CSS and the `breakpoints.yml` file may be made, but at the time that I wrote this book, that bridge did not yet exist. In the mean time, create your breakpoints here as well as in your CSS files.

Creating Advanced Theme Settings

Many Drupal themes provide configurable settings that may be managed through the theme's administrative settings on the appearance page. The Bartik theme, for example, provides the ability to set the colors for nearly every major element in the theme through administrative settings that are exposed on the settings page without having to modify CSS.

To add custom theme settings, the first step is to modify the theme settings form by adding a PHP function to the yourthemename.theme file named yourthemename_form_system_theme_settings_alter function, replacing yourthemename with the name of your theme.

We'll demonstrate the process by adding a site slogan setting to the Davinci theme by first creating a davinci.theme file in the Davinci themes root directory. Within the davinci.theme file, we'll add the function to add the Site Slogan field to the theme configuration page and add a preprocess page function to expose the value stored in the Site Slogan field as a variable that can be accessed on the page template files.

The format of the form field follows the standard Drupal form API setting for a text field, with the addition of a theme_get_setting call to retrieve the value of site_slogan. Note that the setting is stored in the same name as the $form element.

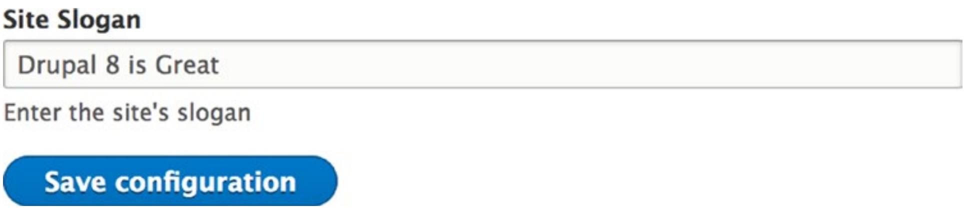
```
<?php

function davinci_form_system_theme_settings_alter(&$form, \Drupal\Core\Form\
FormStateInterface $form_state) {

   $form['site_slogan'] = array(
     '#type'          => 'textfield',
     '#title'         => t('Site Slogan'),
     '#default_value' => theme_get_setting('site_slogan'),
     '#description'   => t("Enter the site's slogan"),
   );

}

function davinci_preprocess_page(&$variables) {

   $variables['site_slogan'] = theme_get_setting('site_slogan');

}
```

The next step in the process is to create a davinci.settings.yml file in the config/install directory of the Davinci theme. In this file, we'll create a simple statement that sets the default value for the site_slogan setting.

```
site_slogan: Drupal 8 is Great
```

The final step in the process is to insert the site_slogan into the page.html.twig template file so that it will be displayed on every page. To do so, pick the correct spot in the page.html.twig file and insert {{ site_slogan }}.

After saving all of the files and rebuilding cache, visit the appearance page and click the Settings link for the Davinci theme. On the Appearance settings page, you can see that the new Site Slogan field appears at the bottom of the form, with the default value we set in the davinci.settings.yml file (see Figure 5-10).

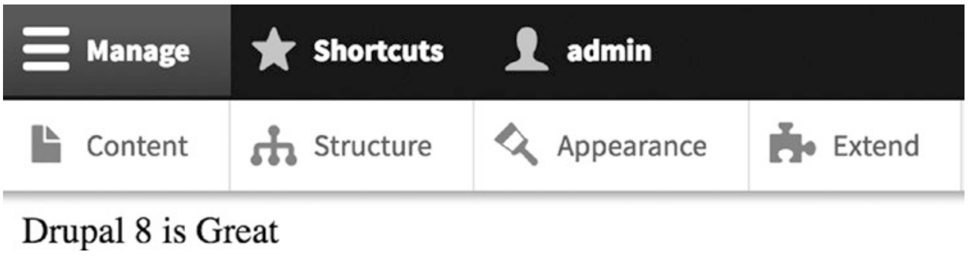

Site Slogan

Drupal 8 is Great

Enter the site's slogan

Figure 5-10. *The Site Slogan settings field*

If you visit a page on the site, you can now see the site slogan printed on the page in the spot where we added `{{ site_slogan }}`, as shown in Figure 5-11.



Drupal 8 is Great

Figure 5-11. *The site slogan rendering on a page*

You may also encounter situations where you need to expose a theme setting value to a node, block, field, or other element. You may do so by adding a `themename_preprocess_type(&$variables)` function, replacing `themename` with the name of your theme and `type` with `node`, `block`, `field`, or other element name. Within that function setting, the variable that will be exposed to the template file for that element type.

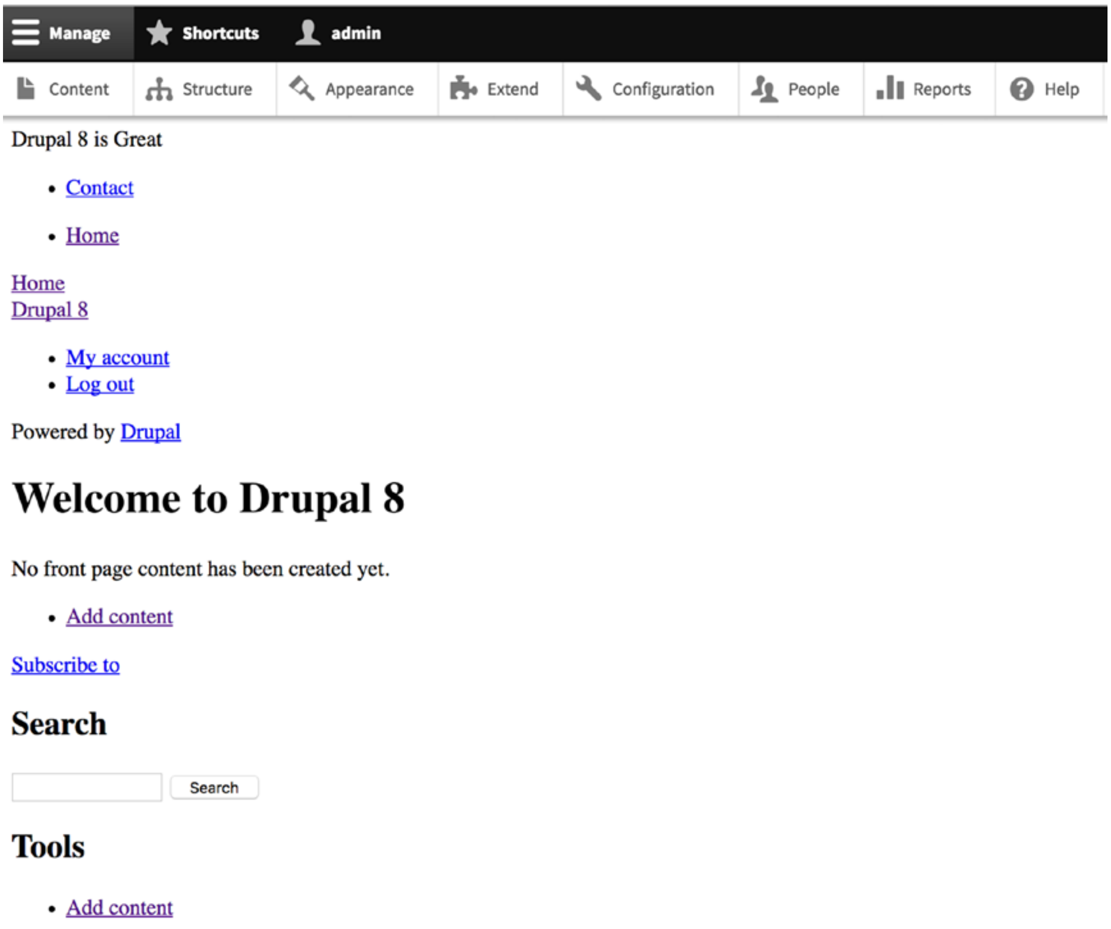
Using Subthemes

One of the enterprise aspects of Drupal 8 themes is the concept of a *subtheme*. Subthemes allow you to create an enterprise theme that every site in your organization can inherit and then extend to meet each individual site's branding and look-and-feel requirements. Subthemes significantly shorten the overall development time because you don't have to recreate a theme from scratch every time a department or group in your organization wants a new Drupal 8 site. Subthemes also to some extent enable you to enforce corporate branding standards across all sites. There are still ways to override nearly everything in a subtheme, and that is where corporate policies come into play.

The process of setting up a subtheme is relatively simple; you only need to add a single statement to the `.info.yml` file of your theme to specify the name of the base theme that you are starting with. The only requirement is that the base theme must reside in your Drupal 8 installation. To demonstrate, we'll modify the Davinci theme, using Classy as the base theme. Classy is included as a theme in Drupal 8 core and can be found at `core/themes/classy`. To the `davinci.info.yml` file, add the following statement:

```
base theme: classy
```

After saving the file and rebuilding the cache, visit the homepage. You'll immediately see subtle differences in the look and feel of the theme, as Davinci now inherits all of the styling and template files from the Classy theme. Figure 5-12 shows the homepage before assigning Classy as the base theme and Figure 5-13 shows the immediate impact of using Classy. While the changes are subtle due to the minimalist nature of the Classy theme, this does demonstrate the power of using subthemes.



Drupal 8 is Great

- Contact
- Home

Home
Drupal 8

- My account
- Log out

Powered by Drupal

Welcome to Drupal 8

No front page content has been created yet.

- Add content

Subscribe to

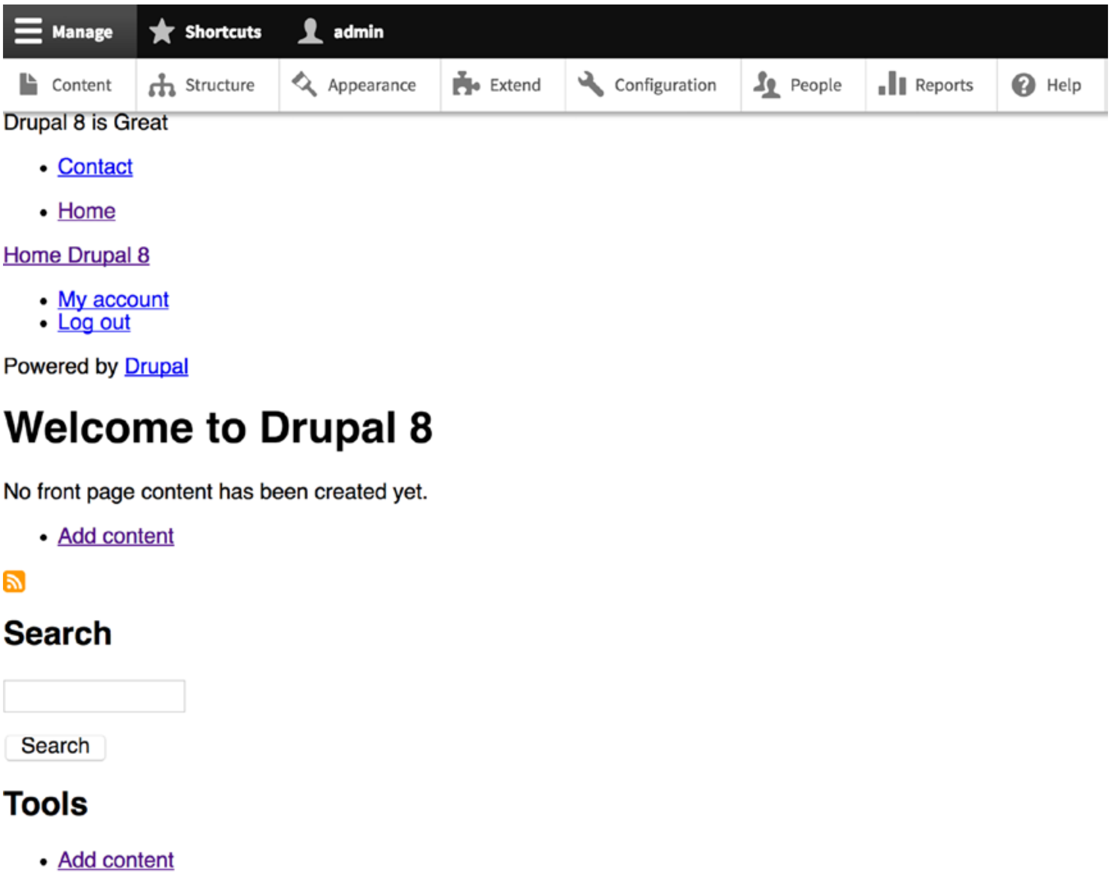
Search

Search

Tools

- Add content

Figure 5-12. *The Davinci theme before applying Classy*



Drupal 8 is Great

- Contact
- Home

Home Drupal 8

- My account
- Log out

Powered by Drupal

Welcome to Drupal 8

No front page content has been created yet.

- Add content

Search

Search

Tools

- Add content

Figure 5-13. *The Davinci theme after applying Classy*

Summary

This chapter covered a lot of ground, starting with the basics of Drupal 8 theming and advancing through the steps of expanding on the capabilities of the Davinci theme. There are even more features and functionally in Drupal 8 theming that you might want to explore at `drupal.org/docs/8/theming`.

The next chapter focuses on leveraging the content that you created on your Drupal 8 site, detailing the process for sharing content with other Drupal and non-Drupal sites, as well as sharing content with new global audiences and empowering your site visitors through enterprise search.