

Learning Liquid

Hints, Tips, and Tricks for Getting Started
with Shopify Theme Development



Introduction

Liquid — Shopify’s templating language — is the backbone of every Shopify theme and is used to create a bridge between HTML files and the data contained within a Shopify store. This ultimately allows you to retrieve and display data (like the name of a product or a series of product images) on your, or your client’s, online store. Needless to say, learning Liquid is the cornerstone of every great Shopify developer’s success!

Whether you’re just starting out with Shopify themes, or you’re a seasoned Shopify Expert, there are always new and useful things to learn that will help with your everyday theme development. We’ve compiled a comprehensive list of hints, tips, and tricks for using Liquid that will help take your Shopify theme development skills to the next level.

In this guide, we’ll cover:

- An overview of the Liquid templating language.
- How to set up a “local” development environment.
- Techniques for getting the most out of your Shopify templates.
- How to use Liquid to improve your images.
- Plus, even more tips, tricks, and hacks for customizing Shopify stores using Liquid!

Table of contents

1 / An Overview of Liquid: What You Need to Know	5
2 / How to Setup a “Local” Shopify Theme Development Environment	21
3 / How URLs Map to Shopify Template	28
4 / The <code>product.liquid</code> Template	32
5 / How to Use Alternate Templates	36
6 / The Power of Alternate Layout Files	41
7 / Using Link Lists in Your Shopify Theme	45
8 / Using Snippets in Your Shopify Theme	53
9 / Using Sections and Blocks in Your Shopify Theme	61
10 / How to Use <code>all_products</code> in a Shopify Theme	69
11 / Manipulate Images with the <code>img_url</code> Filter	74
12 / Ways to Customize the <code>img</code> Element	81
13 / Creating Useful CSS Hooks in Liquid	84
14 / Using Liquid’s <code>case/when</code> Control Tags	87
The Shopify Partner Program	90

CHAPTER 1

An Overview of Liquid: What You Need to Know

If you're new to developing with the Shopify platform, you might be wondering what all the talk about Liquid actually refers to. In this chapter, we'll explain all you need to know about Liquid, how it fits into Shopify theme building, and the core concepts that will enable you to start building powerful and immersive ecommerce templates. Let's begin with a little history.

Liquid was developed by Shopify co-founder and CEO Tobias Lütke and is now available as an [open source project](#) on GitHub. Today, it's used in many different software projects, from content management systems to flat file site generators — and of course, Shopify.

Liquid: language or engine?

Some refer to Liquid as a template language, while others may call it a template engine. It doesn't really matter which label you apply — in many ways both are right. Personally, we like to call it a template language. It has a syntax (like traditional programming languages), has concepts such as output, logic, and loops, and it interacts with variables (data), just as you would with a web-centric language such as PHP.

However, that's really where the similarities end. There's a lot you can't do with Liquid — by design. For example, it has no concept of “state,” it doesn't let you get deep under the covers of the platform, and can occasionally seem counterintuitive for seasoned coders. However, it has been very well thought out, and what might at first seem like a limitation is usually intended and for good reason.

Liquid's function

Liquid, like any template language, creates a bridge between an HTML file and a data store — in our context, the data is of course a Shopify store. It does this by allowing us to access variables from within a template with a simple to use, and readable, syntax.

In Shopify, each template allows us to access certain variables without having to do any heavy lifting. For example, the `product.liquid` template allows us access to all the details relating to the currently viewed product which, in turn, allows us to output this data without having to know anything about the actual product itself. These variables are known as Liquid variables. You can also use Liquid to retrieve data that isn't made available to us. For example, you can ask Shopify to populate a variable you create with all the products in a particular collection.

Once we know the names of the variables we have access to, or created, we can use Liquid constructs such as “output” and “loops” to display the data in our templates.

The Shopify platform understands what data to retrieve, and how to display it depending on the Liquid code you have in your template. It might be a simple case of displaying the name of a product, or something slightly more complex such as showcasing a series of product images.

The great benefit of a template language such as Liquid is that you, as the designer, don't need to know anything about the data itself. As such, your templates are 100 percent agnostic and can be applied to multiple stores without any knowledge of the stores' content.

Liquid's file extension and delimiters

Liquid files have the extension of `.liquid`. A `.liquid` file is a mix of standard HTML code and Liquid constructs. It's an easy to read syntax, and is easy to distinguish from HTML when working with a template file. This is made even easier thanks to the use of two sets of delimiters.

The double curly brace delimiters `{{ }}` denote output, and the curly brace percentage delimiters `{% %}` denote logic. You'll become very familiar with these as every Liquid construct begins with one or the other.

Another way of thinking of delimiters is as “placeholders.” A placeholder can be viewed as a piece of code that will ultimately be replaced by data when the compiled template is sent to the browser. This data is determined entirely by the theme designer as a result of the Liquid code in the template. As such, Liquid templates, much like templates that intersperse PHP and HTML, serve as representations of what will be rendered.

Output

Let's examine the syntax for "output." As the name suggests, output in Liquid will literally output a piece of data from a store into a template.

Here's a quick example of an output placeholder that you'll typically find in the `product.liquid` template:

```
<h2>{{ product.title }}</h2>
```

When rendered, this would output the name of the currently viewed product in place of the `{{ }}`. For example:

```
<h2>American Diner Mug</h2>
```

Output, unless manipulated with a filter (which we'll look at shortly), is simply a case of replacing the entire placeholder with a text string from your store.

Objects and properties

This example also introduces us to the Liquid dot syntax. This is common in many template and server side languages. Taking our `shop.name` example we can break it up into two parts.

The first element preceding the `.` is the object. In this case, it's the shop object. This is a variable that represents all the data relating to the shop that we have defined in the Shopify Admin. These data items include:

- `shop.address`
- `shop.collections_count`
- `shop.currency`
- `shop.description`
- `shop.domain`
- `shop.email`

- `shop.enabled_payment_types`
- `shop.metafields`
- `shop.money_format`
- `shop.money_with_currency_format`
- `shop.name`
- `shop.password_message`
- `shop.permanent_domain`
- `shop.policies`
- `shop.products_count`
- `shop.secure_url`
- `shop.types`
- `shop.url`
- `shop.vendors`

The items following the `.` represent properties of the shop object. A property could be as simple as the name of the store (as per our example above) or it could be a list of items, such as the kinds of payment types enabled in the store.

Collection properties

You'll notice from the list above that a number of the properties are plural, such as:

- `shop.enabled_payment_types`
- `shop.metafields`
- `shop.types`

These properties represent Liquid collections. Instead of returning a string of data such as the name of the shop, they'll return an array of data — in other words, it's a list of items we can access via a Liquid loop.

When first using Shopify and Liquid, it's easy to get confused by collections. We'll therefore refer to “product collections” and “Liquid collections,” the former being a logical grouping of products defined in the Shopify Admin, and the latter being a list of items we can access in Liquid code.

Finally, it's worth saying that each one of the list items in our Liquid collection can also have properties. A good example of this is `product.images`. This represents a list of all the images that have been added to a particular product.

Each of the images in the list has multiple properties associated with it:

- `image.alt`
- `image.attached_to_variant?`
- `image.id`
- `image.product_id`
- `image.position`
- `image.src`
- `image.variants`
- `image.height`
- `image.width`
- `image.aspect_ratio`

In order to access these properties, we have to use a Liquid loop.

Liquid loops

Loops are used extensively in Shopify themes, and are thankfully very easy to understand. If you have done any form of basic programming, the concept of loops will likely be very familiar to you.

Using a loop, often known as a `for` loop, allows us to output the same piece of code a known number of times in our template. As mentioned above, a typical example would be to output all the images associated with a product.

Let's have a look at an example using the `product.images` Liquid collection we discussed earlier.

Our aim with this loop is to output all of the images for a particular product. Here's a very simplistic loop that will output each image inline:

```
{% for image in product.images %}

{% endfor %}
```

Let's break it down into steps to fully understand it.

Step 1

```
{% for image in product.images %}
```

The first line introduces us to the second style of delimiter, the curly brace percentage syntax `{% %}`. Here, we're using a Liquid `for` loop. Loops work with Liquid collections, and allow us to iterate over each item in our list in turn. If the product we're currently viewing had six images associated with it, our `for` loop would loop six times, if it had 10 then it would loop 10 times, and so on. Only once every list item has been looked at (or unless we instruct it otherwise) will the next part of the template be considered.

It's worth noting that unless we specifically ask how big our loop will be, we don't know how many loops will occur — only that Liquid will go over each item in our list, in turn. The loop will finish after the last iteration, and it's at this point that the template will carry on with its processing.

In order to access the properties of each list item, we designate a variable to represent the current item in the loop. In our example above, it's `image`. While this is an obvious choice, and will help other designers understand your logic in the future, it can literally be anything. For example, we could use `alltheimagesintheworld`, in which case it would look as follows:

```
{% for alltheimagesintheworld in product.images %}
```

This is, of course, a silly example to make a point — `image` makes much more sense, but we just want to emphasize the fact that this variable has no relation to the Liquid collection.

Step 2

```

```

The second line of our code example consists of part HTML and part Liquid. You'll also notice that the `src` attribute is populated with a Liquid output tag.

This introduces us to the concept of filters, which are denoted by the `|` (pipe) character — we'll look at these in more detail shortly. In our example, the filter is taking the image variable (the current item in our loop) and is creating a fully qualified URL to the 100px size version of the image, which was created when the product image was added in the Shopify Admin.

We'll look at filters, denoted by the `|` character, next but suffice to say that this short construct will populate the `src` attribute with the fully qualified URL to the 100px version of the current image in our list. The filter does all the work of creating the `src` attribute for us.

Step 3

```
{% endfor %}
```

The final line of our example is our closing endfor statement. This tells the template to carry on after all the loops have been executed.

If we had three images in our `product.images` object, the final output would look something like this:

```



```

Loops are really useful and something you'll encounter daily in your theme development. Outputting images and product variants are two commonly found examples.

Liquid filters

Another very powerful feature of Liquid is output filters, which we used in the code example above. Filters serve three main purposes:

- They manipulate output data in some way.
- They allow our themes to be agnostic.
- They save theme designers time by reducing the amount of code we need to write.

Filters are always used in conjunction with a Liquid output. Let's have a look at some filters, starting with the date filter.

When outputting a blog post, you'll likely want to let the reader know when it was published:

```
<p class="date-time">{{ article.published_at | date: "%a, %b %d, %y" }}</p>
```

You'll notice the `|` character in the middle of the output tag. On the left side of the pipe, we have the article object with its associated `published_at` property, and on the right we have the date filter with an argument to denote the date format — in this case `%a, %b %d, %y`.

Without the filter, Shopify would simply output the date the blog article was published in the format in which it's stored in the database, which may not be humanly readable. However, by adding in the `|` and including the `date` filter, we can manipulate the format so it outputs in a format we want. In this case the date would output in this format:

Tue, Apr 22, 14

Adding a stylesheet

Put simply, filters allow us to take a piece of data from our store and change it. What we start with on the left-hand side gets piped through our filter and emerges on the right-hand side changed. It's this final manipulated data that is then output in the template.

Here's an example of how to add a stylesheet in Liquid:

```
{{ 'style.css' | asset_url | stylesheet_tag }}
```

Here, we're using two filters with the ultimate aim of creating a fully formed style element in a layout file.

We start on the left with the name of our CSS file, which resides in the theme's assets folder. Next we apply our first filter — in this case the `asset_url` filter. This is an incredibly useful filter and one you'll use a lot. We've mentioned before how Shopify themes, thanks to

Liquid, are agnostic. They don't need to have any knowledge of the store they are installed on, and the same theme can be applied to multiple stores. However, this can cause issues when trying to reference assets as we need a way of knowing where a certain asset (image, JS file, CSS file) is on the network.

Thankfully the `asset_url` comes to our rescue. By using this filter, Shopify will return the fully qualified path to the assets folder for the theme and append the name of our asset at the end. Just remember it won't actually check that the file exists — it's up to us to ensure that the first part of the tag, in our case `style.css`, is in the assets folder.

Here's how that might look when output:

```
//cdn.shopify.com/s/files/1/0087/0462/t/394/assets/  
shop.css?28178
```

The final filter in the chain, `stylesheet_tag`, takes the URL and wraps it in a style element which is then output in our layout file. Here's the final result:



```
<link href="//cdn.shopify.com/s/files/1/0087/0462/t/394/  
assets/shop.css?28178" rel="stylesheet" type="text/css"  
media="all">
```

Each filter takes the output from its preceding filter and in turn modifies it. When there are no further filters to pass data into, the result is output as HTML into the template.

There are many really useful filters. Here are just a few you'll find yourself using:

- `asset_url`
- `stylesheet_tag`

- `script_tag`
- `date`
- `pluralize`
- `replace`
- `handle`
- `money`
- `money_with_currency`
- `img_url`
- `link_to`

Liquid logic

The final aspect of Liquid we need to look at is logic.

Here's an example:

```
{% if product.available %}  
<h2>Price: £99.99</h2>  
{% else %}  
<h2 class="sold-out">Sorry - sold out</h2>  
{% endif %}
```

In this case, we're controlling the output to our template using a simple if, else, endif statement. In many ways, if statements are like questions. Depending on the answer to the question, a different piece of markup will be output — or in some cases no markup at all.

In the above example, if the answer to our if statement question is true (`product.available` returns `true` or `false`), we

render the words “This product is available”. If it’s false, our template carries on and outputs the text following our `{% else %}` clause — in this case “Sorry, this product is sold out”.

Another way of looking at logic is that it allows us to control the flow of a template and ultimately make decisions on which data is displayed. This is why these Liquid tags are commonly referred to as “control flow” tags.

It’s worth noting that unlike output tags, the inclusion of logic tags in your templates does not result in anything being directly rendered — rather, they allow us to control exactly what is rendered.

You’ll find yourself using if statements a lot in Shopify theme development. Here’s another example:



```
{% if cart.item_count > 0 %}
  <p>You have {{ cart.item_count }} item(s) in your cart
</p>
{% else %}
  <p>There’s nothing in your cart :( Why not have a
  <a href= “/products”>look at our product range</a></p>
{% endif %}
```

This example demonstrates how you can either display the number of items in a visitor’s cart or output a link to your products.

The full list of control flow tags that you can use to create conditions are:

- `if`
- `unless`
- `else`
- `elsif`

- `case`
- `when`

Operators

You'll notice in this example we're using the greater than `>` operator. As the `cart.item_count` variable returns the number of items in the current user's cart, we can check to see if it's greater than zero, i.e. it has items in it.

If this returns true we can output the message with the current item count; if not we can output `<p>There's nothing in your cart :(` Why not have a `<a href="/products">look at our product range</a></p>` instead.

We could actually refactor our example with a filter. By using the pluralize filter, we can output `item` or `items` depending on the number of items in the cart. The bonus here is that we don't have to know the count in order for Shopify to output the right designation:

```
{% if cart.item_count > 0 %}  
  <p>You have {{ cart.item_count }} {{ cart.item_count |  
    pluralize: 'item', 'items' }} in your cart</p>  
{% else %}  
  <p>There's nothing in your cart :( Why not have a  
  <a href= "/products">look at our product range</a></p>  
{% endif %}
```

You'll notice that the refactored example now includes the `pluralize` filter which takes two parameters. The first is the singular word and the second the plural.

While we've used the `>` operator in the above example, there are a wide range of comparison operators in Liquid, including:

Operator	Function
<code>==</code>	equals
<code>!=</code>	does not equal
<code><</code>	greater than
<code>></code>	less than
<code>>=</code>	greater than or equal to
<code><=</code>	less than or equal to
<code>or</code>	condition A or condition B
<code>and</code>	condition A and condition B
<code>contains</code>	includes the substring if used on a string, or element if used on an array

Whitespace control

Whitespace control in Liquid enables you to remove whitespace rendered by Liquid output. In Liquid, you can use a hyphen in your tag syntax, `{{- , -}}`, `{%- , -%}` to strip whitespace from the left or right side of a rendered tag.

By default, even if your Liquid code doesn't have output, Liquid in a template will still render an empty line in the final HTML.

For example:

```
{% assign my_variable = "coffee" %}
{{ my_variable }}
```

outputs to:

```
coffee
```

However, when you include hyphens in your tag syntax, this whitespace gets stripped out from the rendered HTML.

For example:

```
{%- assign my_variable = "coffee" -%}  
{{ my_variable }}
```

outputs to:

```
coffee
```

If you're someone who likes their HTML to render without whitespace, as a rule you can choose to add hyphens to all your Liquid tag syntax.

Liquid cheat sheet

If you're anything like us, you'll have a hard time committing all these Liquid filters, operators, and structures to memory. Thankfully, we released a [Shopify Liquid Cheat Sheet](#) for you, which is a searchable database of all the Liquid objects, filters, and tags, that links out to the developer documentation. It's an indispensable resource, which we strongly encourage you to bookmark and become familiar with.

Summary

We've covered a lot of ground in this chapter, but hopefully it's given you a solid introduction to Liquid. Here's a reminder of what we covered:

- Liquid is a template language that allows us to display data in a template.
- Liquid has constructs such as output, logic, and loops, and deals with variables.
- Liquid files are a mixture of HTML and Liquid code, and have the `.liquid` file extension.
- Liquid files used in a Shopify theme are agnostic and have no concept of the store they are currently being used in.

- The two types of delimiters used in Liquid.
- How to output data from a store in a template.
- How to manipulate data with filters, and how to link a stylesheet in Shopify.
- How to loop over a Liquid collection to output multiple items.
- The use of logic and Liquid's control flow tags.
- The different types of operators used for comparison.

How to Setup a “Local” Shopify Theme Development Environment

Many developers and designers use and love the online Shopify Theme Editor — it’s easy to work with and is conveniently located within the Shopify Admin itself. But if you’re looking to develop Shopify themes locally, you should know that you’re not limited to the online theme editor.

To set up a “local” Shopify theme development environment you’ll need to use Theme Kit — a cross platform tool that allows you to interact easily with the Shopify platform, while using all of your own development tools.

Once Theme Kit is set up, you can more easily integrate workflow tools like Git into your theme development — giving you the confidence to work on a Shopify Theme with a team of developers, work within your favorite text editor, and have a more localized experience when editing themes. Theme Kit isn’t a truly local development environment, in that it still requires a connection to Shopify servers. If you’re looking for a local offline development tool, checkout Motifmate, which supports an offline option.

Install Theme Kit

Theme Kit is a cross-platform tool for building Shopify Themes, created by Shopify employees. Theme Kit is a single binary that has no dependencies. Once you download Theme Kit, and with a tiny bit of setup, you're off to the theme-creation races.

Some of Theme Kit's notable features include:

- Uploading themes to multiple environments.
- Fast uploads and downloads.
- The ability to watch for local changes and upload automatically to Shopify.
- Works on Windows, Linux, and macOS.

If you're working on Mac, you can use homebrew to install Theme Kit by running the following commands:

```
brew tap shopify/shopify  
brew install themekit
```

If you're working on Windows, you will need to have the Chocolatey Package Manager installed before running the following command:

```
choco install themekit
```

If you are on a Linux based system, you can use the following installation script to automatically download and install the latest Theme Kit for you:

```
curl -s https://shopify.github.io/themekit/scripts/  
install.py | sudo python
```

Troubleshooting older versions and testing

Theme Kit installation

Before you run any Theme Kit commands, make sure you're using the most up-to-date version of Theme Kit. If it's your first time installing Theme Kit, you can ignore the following instructions.

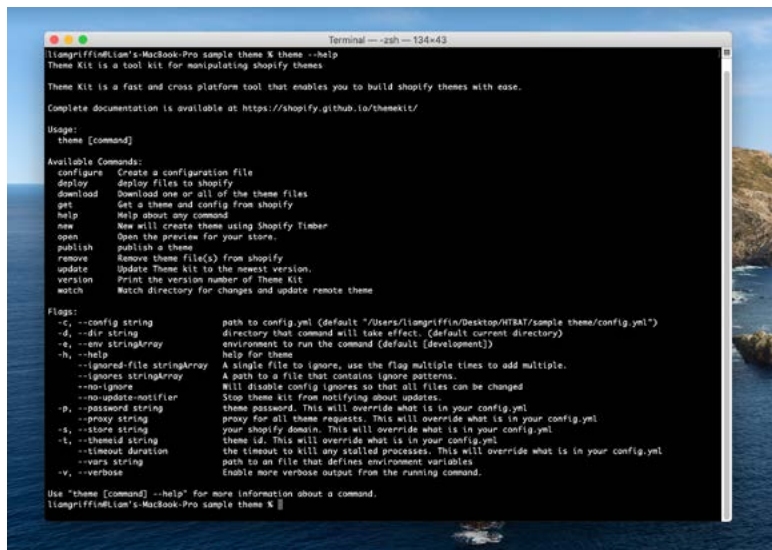
Make sure you're using the most up-to-date version of Theme Kit (you can find versions here). To update Theme Kit, run:

```
theme update
```

To test that Theme Kit is installed and working, and to see available commands, type:

```
theme --help
```

Running `theme --help` should show you all available commands and definitions for Theme Kit.



```
llangriffin@llan's-MacBook-Pro sample theme % theme --help
Theme Kit is a tool kit for manipulating shopify themes.

Theme Kit is a fast and cross platform tool that enables you to build shopify themes with ease.

Complete documentation is available at https://shopify.github.io/themekit/

Usage:
  theme [command]

Available Commands:
  configure  Create a configuration file
  deploy     Deploy files to shopify
  download   Download one or all of the theme files
  get        Get a theme and config from shopify
  help       Help about any command
  new        New will create theme using Shopify Timber
  open       Open the preview for your store.
  publish    Publish a theme
  remove     Remove theme file(s) from shopify
  update     Update Theme kit to the newest version.
  version    Print the version number of Theme Kit
  watch      Watch directory for changes and update remote theme

Flags:
  -c, --config string      path to config.yml (default "/Users/llangriffin/Desktop/HTBAT/sample theme/config.yml")
  -d, --dir string         directory that command will take effect. (default current directory)
  -e, --env stringArray    environment to run the command (default [development])
  -h, --help               help for theme
  --ignored-file stringArray
                           A single file to ignore, use the flag multiple times to add multiple.
  --ignore stringArray     Will disable config ignores so that all files can be changed
  --no-update-notifier     Stop theme kit from notifying about updates.
  -p, --password string    theme password. This will override what is in your config.yml
  --proxy string           proxy for all theme requests. This will override what is in your config.yml
  -s, --store string       your shopify domain. This will override what is in your config.yml
  -t, --themeid string     theme id. This will override what is in your config.yml
  --timeout duration       the timeout to kill any stalled processes. This will override what is in your config.yml
  --vars string            path to an file that defines environment variables
  -v, --verbose            Enable more verbose output from the running command.

Use "theme [command] --help" for more information about a command.
llangriffin@llan's-MacBook-Pro sample theme %
```

Setting up API credentials

Once Theme Kit is installed, we'll need a few things to connect our local theme to your existing Shopify store. We'll need an **API key**, **password**, and **theme ID**.

API key and password

We'll need to set up an **API key** to add to our configuration, and create a connection between our store and Theme Kit. The **API key** allows Theme Kit to talk to and access your store, as well as its theme files.

To do so, we need to log into the Shopify store, and create a private app. In the Shopify Admin, go to **Apps** and click on the **Manage private apps** link at the bottom of the page. From there, click **Generate API credentials** to create your private app. You'll need to provide a title — we usually provide the name of the client and environment for clarity. Make sure to set the permissions of **Theme templates and theme assets** to have **Read and write access** in order to generate the appropriate API credentials, then click **Save**.

Shopify will load a new page, which will provide you with a unique **API key** and **password**.

Theme ID

To connect an existing theme, we need the theme's ID number. There are a few ways to get your theme's ID number. The quickest way is to go to the Theme Editor, by clicking on **Actions > Edit Code**, and copy the theme ID number from the URL — it will be the last several digits after `mystore.myshopify.com/admin/themes/`

If you want to build a theme from scratch, you can do that by running the following in your command line, which creates a new, basic theme in the directory you run it in:

```
theme new --password=[your-password] --store=[your-store.myshopify.com]
```

Hooking it all up with config.yml

Now we can use all the previous information to create a `config.yml` file in our theme, and then download the whole theme locally. The `config.yml` is vital because it's the file that creates a local connection to your Shopify store's theme.

Create a directory for your theme to live in by running:

```
mkdir [your-theme-name]
```

Then, step into that directory using the following command:

```
cd [your-theme-name]
```

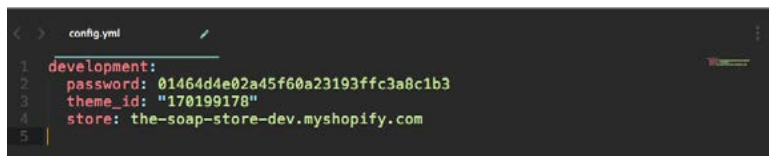
To create the `config.yml` file, run the following command from inside your theme directory, replacing the `[square bracket placeholders]` with your theme's information:

```
theme configure --password=[your-password] --store=[your-store.myshopify.com] --themeid=[your-theme-id]
```

For example:

```
theme configure
--password=01464d4e02a45f60a23193ffc3a8c1b3
--store=the-soap-store.myshopify.com
--themeid=170199178
```

This will automatically create a `config.yml` file for you. You can also manually create a `config.yml` file in the directory with a text editor, which would look something like this:

A screenshot of a code editor with a dark background. The file name 'config.yml' is shown in the top left. The code is as follows:

```
1 development:
2   password: 01464d4e02a45f60a23193ffc3a8c1b3
3   theme_id: "170199178"
4   store: the-soap-store-dev.myshopify.com
5
```

It's also helpful to add `ignore_files` to the `config.yml` file, to avoid overwriting an existing theme's theme settings as well as other files you don't want to overwrite on your store with your local settings.

For example, adding the following nested under the environment in `config.yml` will ignore `settings_data.json` :

```
ignore_files:  
  - config/settings_data.json
```

Then, you can run the following command to download and setup your existing theme in the current directory:

```
theme download
```

Push updates to your theme

Now that the connection has been established to the Shopify Theme, you can run the following command in your theme directory:

```
theme watch
```

Theme Kit will now watch for any changes made to your local files, and automatically push them to your theme. To close the watch connection, simply type `ctrl + c` .

If you're looking for more reading on using Theme Kit, check out [the documentation and other amazing features](#).

Summary

We went through several steps to set up Theme Kit to develop locally. Here's a reminder of what we covered:

- Shopify is a hosted platform, so Theme Kit allows you to sync local theme files with your Shopify store.
- How to install Theme Kit on OSX, Linux, and Windows.
- How to troubleshoot older versions of Theme Kit and confirm Theme Kit is installed.
- How to generate API credentials required to sync your theme with Theme Kit.
- How to generate a `config.yml` file using the password for your API key, Theme ID, and Shop URL.

CHAPTER 3

How URLs Map to Shopify Template

One of the (many) features we love about working with Shopify themes is the simple folder structure. Each store can be powered by a single layout file and a handful of templates, meaning you can achieve a lot with a little — power in simplicity.

However, if you are new to Shopify themes, you may not know exactly when each template gets rendered, or be aware that the same template gets used in various places around the store.

This chapter will focus on building an understanding of what conditions each template is rendered under in a store.

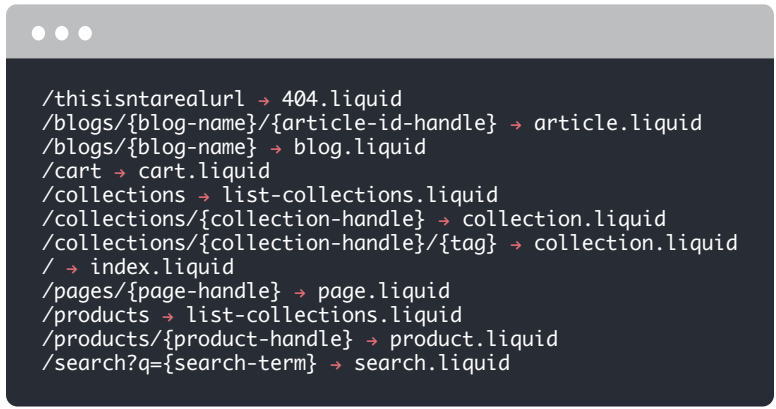
URL template mapping

Internally, Shopify has its own routing table which determines what template is displayed based on the URL requested by the user. If you have ever used one of the popular development frameworks, you might be familiar with the concept of URL routing. Put simply: it's a way of determining which template to send to the browser based on the requested URL.

We mentioned earlier that there are only a handful of templates required to power a store. Each of these templates serves one or more URL — in other words, we're able to utilize the same templates for multiple URLs. From a design perspective, this enables us to reduce our overhead when building a new store.

URLs to templates

Here's an overview of which template is rendered as determined by the URL:



```
/thisisntarealurl → 404.liquid  
/blogs/{blog-name}/{article-id-handle} → article.liquid  
/blogs/{blog-name} → blog.liquid  
/cart → cart.liquid  
/collections → list-collections.liquid  
/collections/{collection-handle} → collection.liquid  
/collections/{collection-handle}/{tag} → collection.liquid  
/ → index.liquid  
/pages/{page-handle} → page.liquid  
/products → list-collections.liquid  
/products/{product-handle} → product.liquid  
/search?q={search-term} → search.liquid
```

Password protected

You might have noticed that the `password.liquid` template isn't included in the list. This template is only seen if you choose to password protect your storefront, and as such will override all other URLs.

If your store is password protected and you don't have a `password.liquid` template in your theme, Shopify will render its default password login page instead.

Alternate templates

It's also worth remembering that the above routing table can be affected by alternate templates — something we'll cover in a later chapter.

URL parameters

As you'll see above, a number of the routes have elements of the URL path wrapped in `{ }`. We have included this to denote a variable that will have an impact on the data loaded into a template.

For example, if we take the `/collections/{collection-handle}` URL pattern, a different set of data will be loaded into the template and sent to the browser if we requested `/collections/bikescompared` instead of `/collections/cars`.

You'll also notice that a number of different URL patterns share the same template file. For example, `/products` and `/collections` will both render the `list-collections.liquid` template. Likewise, `/collections/`, `/collections/{collection-handle}/`, and `/collections/{collection-handle}/{tag}` all make use of `collection.liquid`.

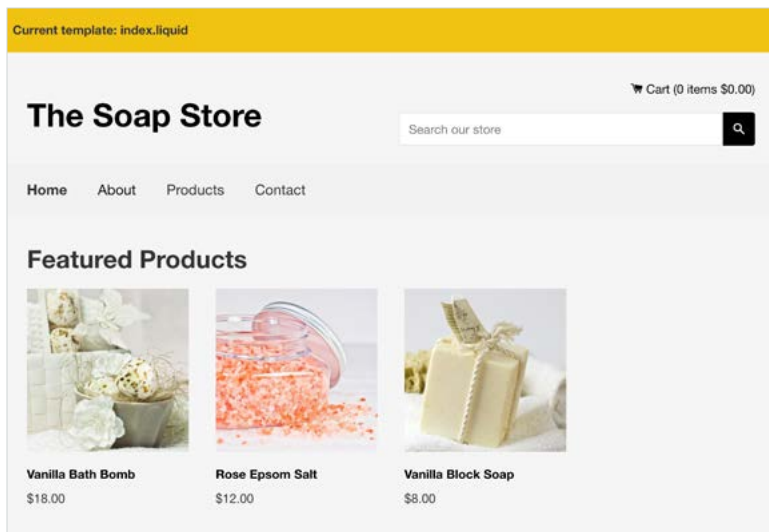
Final note

If you're ever unsure which template is being rendered, there's a really simple way to check.

All you need to do is add `{{ template }}` to your `theme.liquid` file and start browsing your store. This global Shopify variable will output the currently rendered template minus the `.liquid` extension. It's a neat way to be doubly sure your templates are working as expected.

Here's a handy example that you can use in your own theme development with the output shown in the screenshot below:

```
<p style="background: #f1c40f; padding: 1em; font-weight: bold;">Current template: {{ template }}.liquid</p>
```



The **product.liquid** Template

So far in our book we've looked at how URLs are mapped in our Shopify templates. In this chapter, we'd like to take a more in-depth look at one particular template — **product.liquid**.

If you are new to Shopify themes, **product.liquid** is the template that is rendered by default whenever a customer views a product detail page. In the next chapter, we'll look at how to create alternate product templates, however in this chapter we'll stick to the basic template, which resides in the templates folder within a Shopify theme.

By way of an example, we're going to use the `product.liquid` template from our own starter theme “[Birthday Suit](#).”

Here it is in its entirety:

```
<h2>{{ product.title }}</h2>
<p>{{ product.description }}</p>
{% form 'product', product %}
  <select name="id">
    {% for variant in product.variants %}
      <option value="{{ variant.id }}"> {{ variant.title }}-
        {{ variant.price | money }}</option>
    {% endfor %}
  </select>

  <button type="submit" {% unless product.available %} dis-
    abled="disabled" {% endunless %}>
    {% if product.available %}
      <p>Add to Cart</p>
    {% else %}
      <p>Sold Out</p>
    {% endif %}
  </button>
{% endform %}
```

As you'll see, there's very little HTML in this template. This is on purpose, as it's intended to be a starting block for your own theme. If you download a theme from the Shopify Theme Store, you'll notice that the `product.liquid` template will be more involved.

Let's examine what's happening in detail. We begin by using Liquid output to display the product's title and description:

```
<h2>{{ product.title }}</h2>
{{ product.description }}
```

As the description is entered via the Shopify Admin, we don't need to wrap this output with further HTML. Of course, if you need to add in a wrapper element of some sort, you can.

Moving down the template, we come to the Liquid form tag, which creates an HTML `<form>` element along with the required `<input>` elements to submit the form to a particular endpoint — in this case to add a product to the cart. There are a range of different parameters which can be added to the `form` tag, depending on the type of form you are using. You can learn more about the `form` tag in our [developer docs](#).

Next we add a select element that will output a drop down option to pick a product variant. Within this element we create a `for` loop to iterate over all the current product's variants and display the variant's price, with currency formatting — thanks to the `money` filter.

Lastly, to allow buyers to submit this form, we'll create a `button` input with the attribute `type="submit"` inside the form tags. We use the `unless` control flow tag to set up a condition disables the add to cart button if the product is unavailable. If the product has inventory, and so is available, the button will read “Add to cart,” otherwise it will read “Sold out.”

We complete the template by closing out the form with the `{% endform %}` Liquid tag.

```
{% for variant in product.variants %}
{% if variant.available == true %}
<option value="{{variant.id}}"> {{ variant.title }} for
{{ variant.price | money_with_currency }}</option>
{% else %}
<option disabled="disabled"> {{ variant.title }} -
sold out!</option>
{% endif %}
{% endfor %}
```

This template makes use of both the `product` and `variant` objects. They have a large range of properties that you can display in this template, and are worth investigating as you develop your Shopify theme skills.

Extending the template

Of course this example is relatively simplistic and is intended as a starting point for your own development. There's a lot more you could include in this template:

- Adding in Liquid code to display product and variant images.
- Using the Shopify JavaScript snippet option_selection.js to allow better display of variant options.
- Using the `| t` filter for retrieving translated strings from your theme's locale file.
- Including sections or snippets to pull in code from other files.

How to Use Alternate Templates

If you're new to Shopify theme building, your first impression might be that every collection, page, and product page is controlled by a single template. Luckily, this isn't the case and there are, in fact, a number of ways you can apply different, or alternate, templates to these various page types.

This chapter will run you through the basics of creating your first alternate template so that you can start customizing your Shopify themes even further.

Creating an alternate template

Creating an alternate template is straightforward. There are two approaches.

If you're using [Theme Kit](#), or are uploading your theme using a ZIP file, you can simply add a file to your theme's templates folder using the following filename syntax:

```
default_template_name.*.liquid
```

For example, an alternate page template could be called:

```
page.about.liquid
```

Or, for an alternate product template, you could use:

```
product.shoes.liquid
```

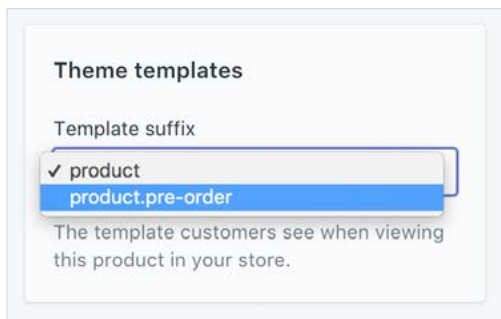
The name itself is irrelevant — the more obvious the better so your clients can recognize its purpose easily.

The second approach is to create an alternate template within the Shopify Admin itself. Here's how:

1. From your Shopify admin, click **Online Store**, then click **Themes**.
2. Find the theme you want to edit, click **Actions**, then click **Edit Code**.
3. Under the templates folder, click the **Add a new template** link.
4. Choose the appropriate option for your new template and give it a meaningful name.
5. **Edit** and **save** your new template as you normally would.

A full description and run through is available in the [Shopify Documentation](#).

Selecting an alternate template



Once an alternate template exists, a new drop-down menu will appear in the relevant **edit page** in the Shopify Admin. This will allow you to select which template you would like applied to the collection, page, or product. Shopify will use the base template by default so you won't need to change every existing item — just the ones you wish to be rendered with the new alternate template.

Switch templates via the URL

Finally, there's one other option for template selection that you have at your disposal. That is being able to select a particular template using the view querystring.

Here's an example for you to review (these links are for demo purposes only):

```
http://store.myshopify.com/products/blue-t-shirt?view=special
```

In this instance, Shopify will load a product template called:
`product.special.liquid`

This technique works for all templates. Here's a collection page example:

```
http://store.myshopify.com/collections/  
computers?view=list
```

In this case, Shopify will load a collection template called:

```
collection.list.liquid
```

If the template requested does not exist, Shopify will fail gracefully and use the default template, or the template specified in the admin.

A really common use case for this technique is for switching between a list and grid view within a product collection.

Alternate templates with sections

Depending on your theme and the type of template that you've created, you might need to create a new section file. If your new template contains the code that you want to edit, creating a section won't be necessary. In this case, you can simply edit your new template as needed.

However, if your new template includes a Liquid tag for a section that contains the code you want to edit, you will need to create a new section for that alternative code.

For example, if you're creating an alternate product template, you'll need to create a new section, since most of the code that makes up the product page is stored in a section file, rather than the template file.

In your template file, you might find:

```
{% section 'product-template' %}
```

Replace `product-template` with the name of the new section that you will create next. For example, if you call your new section `product-alternative`, your code should look like this:

```
{% section 'product-alternative' %}
```

Start implementing alternate templates today

Alternate templates are a great example of the power of Shopify themes. By taking a few minutes to understand how to create them, apply them, and even switch them via a querystring, you start to expose the power of the platform and offer your clients, and their customers, even richer ecommerce experiences.

The Power of Alternate Layout Files

Our previous chapter focused on how to create and use alternate templates when creating Shopify themes. Let's now turn our attention to Liquid layout files.

If you aren't familiar with layouts you'll find the default file, `theme.liquid`, in the layouts folder within your theme directory. If you've never seen one before you might be wondering what's going on!

The `theme.liquid` file can be thought of as the master template for your store. Effectively it's a wrapper for all our other templates found in the templates folder. As a general rule, elements that are repeated in a theme (ex: site navigations, header, footer, etc.) will often be placed inside `theme.liquid`.

It's entirely up to the theme designer to decide how much, or little, code is included in a layout file. For example, some developers often prefer to have certain elements of a layout file included as a snippet, or section, as this allows them to be re-used in alternate layout files — a topic we'll cover shortly. Just remember that all rendered pages in a Shopify theme, unless stated, will be based on the default `theme.liquid` layout file.

The benefits of layout files

One of the main benefits of layout files is that they enable us to follow the DRY (Don't Repeat Yourself) principle. By having all our common elements in a single file, it allows us to make global changes very easily. Another benefit is that our templates (`product.liquid` , `collection.liquid` , etc.) aren't cluttered with markup that is repeated across the store.

Creating a layout file

Regardless of how much HTML you include in a layout file, there are two important Liquid tags that you must include in a Shopify layout file:

1. `{{ content_for_header }}` must be placed between the opening and closing `<head>` tag. This inserts the necessary Shopify scripts into the `<head>` which includes scripts for Google Analytics, Shopify analytics, for Shopify Apps, and more.
2. `{{ content_for_layout }}` must be placed between the opening and closing `<body>` tag. This outputs dynamic content generated by all of the other templates (`index.liquid` , `product.liquid` , etc.).

`theme.liquid` , along with its two required placeholders tags, are required in order for Shopify to validate a theme.

Alternate layouts

One layout file isn't going to cover every eventuality, and there will be situations where you'll require a completely different layout. You could start hiding elements with CSS, but that feels a little wrong — the far better approach is to create an alternate layout complete with different HTML markup.

A good example of this is a specific landing page for a product or a newsletter signup page that doesn't require the same "site furniture" as the rest of the site. In these situations, it's possible to designate that the landing page renders with an "alternative" layout file.

Creating an alternative layout is very straightforward. The first thing to do is create a new file and give it a relevant name and the `.liquid` extension. Next, save it in the layouts folder in your theme directory. In this file, place any HTML you need (e.g. HTML declarations, CSS, JS links, etc.) along with the two placeholders discussed above.

In order to use this layout file, and effectively override the default `theme.liquid` layout file, we use the following Liquid syntax as the first line in any template file (`index.liquid` , `product.liquid` , etc.):

```
{% layout 'alternative' %}
```

In this instance, the default `theme.liquid` will not be applied, but rather the layout called `alternative.liquid` .

It's also possible to request that the layout file isn't applied.

The syntax to request that a layout file isn't applied is:

```
{% layout none %}
```

This needs to be the first line at the top of the relevant template (`index.liquid` , `product.liquid` , etc.). A use case for this might be when rendering output from your store in an alternative syntax such as JSON.

Using snippets to be even more DRY

If we know that a theme will be using multiple layouts, we often remove code out of the layout file and into a snippet. This means that we can reuse code across multiple layouts. For example, we often have the following structure:

snippets/html-header.liquid	Contains all the essential head items right up to the opening body tag.
snippets/html-footer.liquid	Contains any relevant script tags and the closing body tag.
snippets/header.liquid	The main header that is used across the majority of the site.
snippets/footer.liquid	The main footer that is used across the majority of the site.

In order to use these, our base layout file would look as follows:

```
{% render html-header %}
{% render header %}
{% render footer %}
{% render html-footer %}
```

The benefit of this approach is that when you come to create an alternate layout file, you don't need to recreate all your HTML header and footer content — meaning you can update it all from two files. If you're only using one or two layouts, it's perhaps overkill. We'll be looking closer at how snippets work with Shopify themes in chapter 8.

Start using alternate layouts in your Shopify theme development workflow

Alternate layout files can be extremely handy when you require radically different markup for a particular page or set of pages. Coupled together with the use of alternate templates, it's a powerful tool in your theme building toolbox and literally gives you endless possibilities to customize the look and feel of a store.

CHAPTER 7

Using Link Lists in Your Shopify Theme

One of the most underused features in Shopify are link lists. As its name suggests, a link list is a simple collection of links. The link items can be created to point to a page, collection, or product within Shopify, or to a URL outside of the store's domain.

It's possible to use a link list for a variety of reasons. In this chapter, we'll examine a common theme use case: nested navigation using an unordered list. By using link lists and Liquid, we'll have full control over the menu from within the admin, giving flexibility to the merchant running the store.

Creating a nested navigation

Shopify added the ability to create a nested navigation menu, up to three levels deep from a single page, by using a new menu editing interface. Previously, menus were created using multiple menus and the handle for each menu to tie it to its parent menu link. At the time of writing, all newly created stores have the new nested menus user interface, where you can easily drag, drop, and nest menu items.

While it's common to include the navigation in a layout file, the default one being `theme.liquid`, you can test out the nested navigation concept in any template.

Creating menus

We'll begin by creating a new menu, our parent menu, by heading to the **Navigation tab** in the Shopify Admin, which resides under the **Online Store** link in the sidebar.

All new stores have a predefined default menu called “Main Menu.” To add items to the list, simply click the **add another link** button, and give your new item a “link name” and a destination. The **select drop down** will allow you to easily link to internal sections, such as a particular product or collection. Alternatively, you can enter your own URL (either internal or external) by choosing “web address” from the options.

Once we have this in place, we can start to consider the Liquid code we'll need to output this in our theme.

You can drag and drop nested menu items to create a multi-level navigation, and with some JavaScript and CSS easily style it into a “super-menu” or “drop-down menu.”

Outputting the menu

In order to output the menu in a theme file, we'll need to know the handle of the menu. As discussed in other chapters, handles are unique identifiers within Shopify for products, collections, link lists, and pages.

Let's begin by outputting all the items from the Main Menu link list. We can use a simple `for` loop we've used many times before to output the link list items in turn:

```
<ul>
{% for link in linklists.main-menu.links %}
  <li><a href= "{{ link.url }}">{{ link.title }}</a></li>
{% endfor %}
</ul>
```

This Liquid syntax isn't new to us, however it's worth examining the opening Liquid section:

```
{% for link in linklists.main-menu.links %}
```

Once again, we're using the variable `link` to hold the data relating to each item in the link list, as we loop over all the items. In order to access the data, we need to access all the `links` in the link list with a handle of `main-menu`.

Remember, the default **Main Menu** that exists in a Shopify store has the handle of `main-menu`, which is why it's being used above. If our menu had a handle of `uk-brands`, the syntax would be refactored as:

```
{% for link in linklists.uk-brands.links %}
```

Each link item has properties including:

- url
- title

In our example above, `{{ link.url }}` will output the url we entered or generated in the admin, and `{{ link.title }}` will output the text we attributed to the link.

Multi-level navigation

Now that we have the basic Liquid structure in place for a single level menu, we need to consider how to create a sub-menu for our top level items. Firstly, we need to head back to the Shopify Admin and create our first sub-menu.

It might not be 100 percent clear initially, but every link in a link list, in addition to the menu itself, has a unique handle that we have access to in Liquid.

Let's have a look at an example. If our `main-menu` has three levels of links, it could look as follows:

- Home
- About Us
- Women
 - ▷ Accessories
 - Earrings
 - Scarves

What's great about using nested menus in Shopify is that nested menu items can be obtained directly from their parent link using Liquid.

This greatly simplifies the markup required to render a nested menu — meaning you don't need to know the handle of the parent to render its children.

Here's an example of how we can use these related handles to output a three-level deep nested menu:

```
<ul class="parent">
{% for link in linklists.main-menu.links %}
<li><a href="{{ link.url }}">{{ link.title }}</a>
{% if link.links != blank %}
  <ul class="child">
    {% for child_link in link.links %}
    <li><a href="{{ child_link.url }}">{{ child_link.
title }}</a>
    {% if child_link.links != blank %}
      <ul class="grandchild">
        {% for grandchild_link in child_link.links %}
        <li><a href="{{ grandchild_link.url }}">{{ grand
child_link.title }}</a></li>
        {% endfor %}
      </ul>
    {% endif %}
    </li>
    {% endfor %}
  </ul>
{% endif %}
</li>
{% endfor %}
</ul>
```

You'll notice that we're now introducing an `if` statement in our refactored example, directly after we output the first level of our main menu:

```
{% if link.links != blank %}
```

This `if` statement checks to see if a child-link for the current link item in our loop exists. If it does exist, the template moves forward and loops over all the items in the sub menu.

Additionally, in this example we handle `child_link` sub-menu and a `grandchild_link` sub-menu the same way, by checking with an `if` statement to see if there's a child-link for the current link

item, and if it does exist, the template loops through and outputs the sub-menu.

In the example above, `child_link` is just a `for` loop variable we use to represent the current item in the loop; it could easily be replaced with `sub_link`, and `grandchild_link` with `sub_sub_link`. We've used `child` and `grandchild` in this case to illustrate the hierarchy of the nested navigation a bit more clearly.

Final touches

We'd like to quickly mention one extra link property that will be very useful when creating menus — `link.active` and `link.child_active`. These are both boolean properties (`true/false`) that allow you to easily tell if the current page is active, as well as if it's nested items are active. Here's the syntax:

```
{% if link.active %} class="active {% if link.child_active %}child- active{% endif %}"{% endif %}
```

In this example, we'll add a CSS class of `active` if the current page URL is the same as the list item, and a class of `active-child` if

the current page is also part of the active nested item. Here's the full code example for completeness:

```
<ul class="parent">
  {% for link in linklists.main-menu.links %}
  <li {% if link.active %}class="active {% if link.child_active %}child-active{% endif %}"{% endif %}><a href="{{ link.url }}">{{ link.title }}</a>
  {% if link.links != blank %}
    <ul class="child">
      {% for child_link in link.links %}
      <li {% if child_link.active %}class="active {% if child_link.child_active %}child-active{% endif %}"
      {% endif %}><a href= "{{ child_link.url }}">
      {{ child_link.title }}</a>
      {% if child_link.links != blank %}
        <ul class="grandchild">
          {% for grandchild_link in child_link.links %}
          <li {% if grandchild_link.active %} class="active
          {% if grandchild_link.child_active %}child-active{% endif %}"{% endif %}><a href= "{{ grandchild_link.url }}">{{ grandchild_link.title }}
          </a></li>
          {% endfor %}
        </ul>
      {% endif %}
      </li>
      {% endfor %}
    </ul>
  {% endif %}
</li>
{% endfor %}
</ul>
```

Summary

Link lists are a very powerful element of the Shopify platform. Having the ability to create an array of list items that can be changed in the admin gives you lots of flexibility. We've seen theme developers use them far beyond menu structures. Knowing how to create nested navigation that can then be styled with CSS is a great tool to have at your disposal.

CHAPTER 8

Using Snippets in Your Shopify Theme

If you've worked with server-side languages, you'll already be familiar with the concept of `partials` or `includes`. In Shopify, `includes`/`partials` are known as `snippets`.

To help you understand how Shopify uses them, here's a brief overview:

- Snippets are files containing chunks of reusable code.
- They reside in the snippets folder.
- They have the `.liquid` extension.
- They are most often used for code that appears on more than one page but not across the entire theme.
- They are included in a template using the Liquid tag `render`. For example:

```
{% render 'snippet name' %}
```
- You don't need to append the `.liquid` extension when referencing the snippet name.
- Examples of snippets include social links and pagination blocks.

Advanced snippet use

Snippets are extremely useful and allow you to keep repeated code in a single file. Above all, this has the benefit of enabling us to update all instances of that code from one file.

We use snippets a lot when designing themes. The main benefit we find is that they allow us to concentrate on discrete chunks of code, as opposed to dealing with long files. Given that there's no performance disadvantage, we find that it's just a nice way of working.

Of course, everyone has a different workflow style. But beyond the aesthetic and organizational benefits of snippets, there are other reasons that you might wish to consider using them.

Conditional loading of snippets

One example of an alternative use of a snippet is conditional loading. For example, let's say we wanted to show a special offer box for a set of particular products, each of which has `coffee cup` in its product handle.

Every object within Shopify has a unique `handle`. In other platforms, such as WordPress, this is known as a `slug`. A handle is a URL-safe representation of an object. Every product has a handle that is automatically created based on the product title, but you have the potential to manipulate the handle in the admin to be whatever you like.

Given their unique nature, handles are easy to use in Shopify templates for comparison. By using a simple Liquid `if` statement, we can check for the current products handle and make a decision on whether or not to include the snippet.

Here's an example to explain the concept that would be placed in `product.liquid`:

```
{% if product.handle contains "coffee-cup" %}
  {% render "special-offer" %}
{% endif %}
```

As you can see, this `if` statement checks for the currently viewed products handle. If the returned value contains `coffee-cup`, the template will include the snippet `special-offer`. If the returned value doesn't match, the template will simply carry on rendering.

This is a very simplistic example of conditional loading, but it shows how we can use the power of Liquid to output different markup dependent on the product. By using this simple method, you can create exceedingly flexible themes.

Naming conventions

As mentioned earlier, the snippets folder acts as one big bucket for all of your theme's snippet files. As a result, we tend to prefix our files with their function to make working with them cleaner and easier.

For example:

- `product-limited-edition-coffee-cup.liquid`
- `product-showcase.liquid`
- `collections-coffee-cups.liquid`

You'll notice that these are very much in line with the template naming conventions, making them much easier to integrate into your workflow.

Variable scope

When a snippet is rendered in a template file, the code inside it does not automatically have access to the variables assigned using variable tags within the snippet's parent template. Similarly, variables assigned within the snippet can't be accessed by the code outside of the snippet.

However, what if we'd like to make use of a snippet, and reference a variable that is a template variable? In order to achieve this, we simply use the Liquid tag `{% assign %}` and list the variables as parameters on the render tag.

Here's an example:

```
{% assign my_variable = 'apples' %}  
{% render 'name', my_variable: my_variable, my_other_  
variable: 'oranges' %}
```

The snippet will now have access to both the `apples` variable and the `oranges` variable. We could also make a Liquid collection

available in the following format: `{% assign all_products = collections.all.products %} {% render 'snippet' %}`

Using `with`

To round out our look at snippets, let's spend some time looking at an example that uses the render tag parameter `with`. This approach really shows off the power of snippets and allows us to create reusable code that can be used in a variety of contexts.

To set the scene, let's say we have a snippet that allows us to output a product collection in a template. Here's a very basic example that we could save as `collection-product-list.liquid`:

```
<ul>
{% for product in collections.all.products %}
  <li><a href="{{ product.url }}">{{ product.title }}</a>
{% endfor %}
</ul>
```

Since the collections variable is global, this will work as intended in any template. This snippet simply outputs an unordered list of links to every product in the store.

What if we wanted to make it possible to work with any individual product collection? In order to achieve this, we need to refactor the snippet to:

```
<ul>
{% for product in collection-product-list %}
  <li><a href="{{ product.url }}">{{ product.title }}</a>
{% endfor %}
</ul>
```

You'll notice that instead of looping over every item in the `collections.all.products.liquid` collection, we have a placeholder variable that has the same name as our snippet, minus the `.liquid` extension.

Let's have a look at how we make use of this more generic snippet:

```
{% assign c = collections.all.products %}
{% render 'collection-product-list' with c %}
```

Firstly, we're assigning the `collections.all.products` to a Liquid variable. In this instance, it's called `c` but can be named however you see fit.

Next, we move onto the `render` tag and reference the snippet without the `.liquid` extension, following it with `with c`. The `with` parameter assigns a value to a variable inside a snippet that shares the same name as the snippet. While this might sound confusing at first, have a quick look at the example above which has the following line:

```
{% for product in collection-product-list %}
```

Effectively what is happening is that the variable `c` is referenced within the snippet by `collection-product-list`. As such, our snippet

will now function with any product collection we pass in using the `with` parameter.

Extending our generic snippet

It's also possible to pass in more than one variable to our snippet. A good example of this is the number of products to show. We can achieve this in our snippet by using a limit clause on the `for` loop.

Here's the refactored snippet:

```
<ul>
{% for product in collection-product-list limit:
limit_count %}
  <li><a href="{{ product.url }}">{{ product.title }}</a>
{% endfor %}
</ul>
```

And here's how we would pass in a value for our limit clause to reference:

```
{% assign c = collections.all.products %}
{% render 'collection-product-list' with c, limit_
count: 2 %}
```

When the snippet is rendered, it will exit after the second loop. This makes our snippet even more generic and will allow us to use it in a variety of situations.

Note that omitting this variable will mean all the items in the Liquid collection are iterated over. Also, if the `limit_count` is higher than the number of items in the list, it will exit the `for` loop after the final list item.

You can pass in further variables by comma-separating them after the `with` parameter. For example:

```
{% render 'collection-product-list' with c, limit_count: 2, heading_text: 'All Products' %}
```

You can output the **heading_text** within the snippet in the following way: `{{ heading_text }}`

Start using snippets today

While snippets might at first seem to be just another simple tool in your arsenal, it's possible to turn them into a very powerful part of your theme that allows you to create boilerplate markup to be used in a variety of contexts.

Using Sections and Blocks in Your Shopify Theme

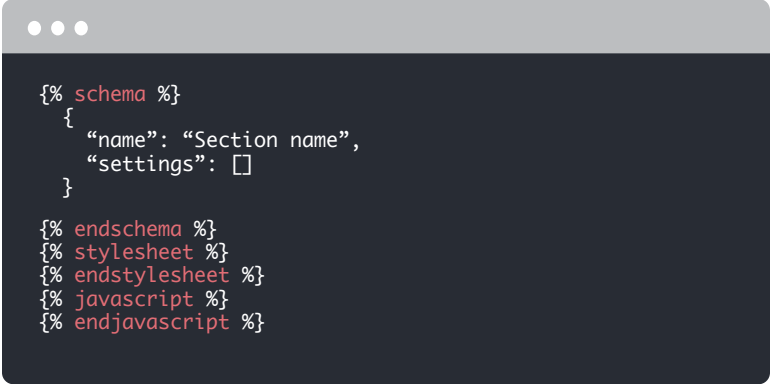
Sections are modular, customizable elements of a page, which can have specific functions. Sections are similar to snippets, in that they are partials, but they allow customization options on the Online Store Editor.

Here's an overview of some of their features:

- Sections can be statically included in a theme's templates, or dynamically added to the theme's homepage from the Online Store Editor.
- Sections are included in template files using the `{% section 'section_name' %}` Liquid tag, or automatically added to homepages when there are presets in the section file.
- Sections support three Liquid tags, which are not usable outside of section files: `{% schema %}`, `{% javascript %}`, and `{% stylesheet %}`.
- While there are Liquid tags for adding section-specific CSS, by default a section will pull its styling from the main stylesheet, such as `theme.scss.liquid`.
- Sections can include basic and specialized input types.

Creating a static section in your Shopify theme

When you create a new section from the theme file editor, a scaffold is automatically created with schema, CSS, and JavaScript tags. Within the schema tags we would add JSON, which would define how the Theme Editor “reads” our content. The CSS and JavaScript tags can be used to add styling or functions specific to this section, but by default the section will pull its styles from the main stylesheet of the theme. This is what a section scaffold would look like:



```
{% schema %}
{
  "name": "Section name",
  "settings": []
}

{% endschema %}
{% stylesheet %}
{% endstylesheet %}
{% javascript %}
{% endjavascript %}
```

To add content to a section, you’ll want to add HTML and Liquid tags to the very top of the file. Sections use the Liquid syntax `{{ section.settings.name }}` to be identified as fields or custom content. Liquid tags can then be defined within the schema, so the section can be customized in the Online Store Editor. You can see the different input values that can be added to [the schema settings in our documentation](#).

One example section we could create is a custom text box, with a personalizable heading and rich text box. You can see that the Liquid tags in the HTML correspond with the IDs within the settings of the schema section:

```

<div id="textsection">
  <div class="simpletext">
    <h1> {{ section.settings.text-box }} </h1>
    <h3> {{ section.settings.text }} </h3>
  </div>
</div>

{% schema %}
{
  "name": "Text Box",
  "settings": [
    {
      "id": "text-box",
      "type": "text",
      "label": "Heading",
      "default": "Title"
    },
    {
      "id": "text",
      "type": "richtext",
      "label": "Add custom text below",
      "default": "<p>Add your text here</p>"
    }
  ]
}

{% endschema %}
{% stylesheet %}
{% endstylesheet %}
{% javascript %}
{% endjavascript %}

```

In the example above, we've created a plain text box and a rich text box, but you can add a wide range of output types depending on your requirements. Other valid input types include `image_picker`, `radio`, `video_url`, and `font`.

Within the schema tags, `id` refers to the Liquid tag being defined, `type` is assigning the kind of output we're creating, `label` is defining a name for this output, and `default` is assigning a placeholder.

To add this section to a specific template (eg: `product.liquid` or `page.liquid`), we would add `{% section 'name_of_`

`section' %}` to the required Liquid template file. This works similar to how you would include a snippet in a page template, but the syntax is slightly different.

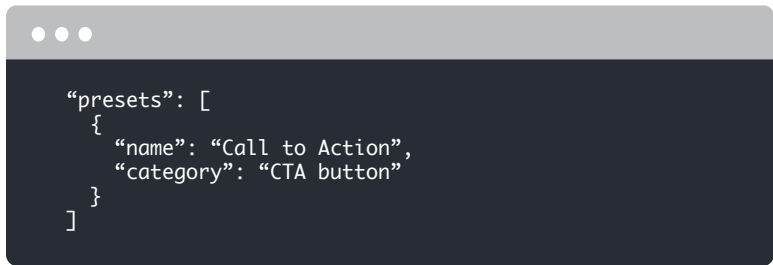
Creating a dynamic section on your Shopify theme

Unlike static sections, dynamic sections can be moved into different positions on the homepage. This drag and drop functionality means that when you build custom dynamic sections, a wide range of options for personalizing homepages are available.

To make a dynamic section, we need to add `presets` within the `schema` tags of the section file. Presets will define how the section appears in the Theme Editor, and the `presets` must have a `name` and `category`.

Once these presets are added, the sections will automatically be available to be added to the index page. Presets are not included in the base file when you add a new section, but adding them manually is straightforward.

For example, presets for a call-to-action button section could look like this:



```
"presets": [
  {
    "name": "Call to Action",
    "category": "CTA button"
  }
]
```

Once these presets are added to the end of the schema file, the theme will automatically recognize this as a dynamic section, which

can be added to the index page. This means that when we access the Online Store Editor and add a section to the homepage, an option for “Call to Action” would appear. This section can now be moved around the page into different positions.

Adding blocks to sections

Blocks are containers of settings and content that can be added, removed, and reordered within a section. What makes blocks different from sections is that elements can be moved around *within* a section. A range of different types of blocks can be added to sections, and the positions of these blocks can be changed, all from the Theme Editor. A block could be an image, video, custom text, or any of the input setting type options seen below:

Value	Application
<u>text</u>	Single-line text fields
<u>textarea</u>	Multi-line text areas
<u>image_picker</u>	Image uploads
<u>radio</u>	Radio buttons
<u>select</u>	Selection drop-downs
<u>checkbox</u>	Checkboxes
<u>range</u>	Range sliders

When we're creating blocks, we wrap our block objects in the Liquid logic loop `{% for block in section.blocks %}` so that the block will render on the Online Store Editor. The structure for this would look like:

```
{% for block in section.blocks %}
  <!-- output block content -->
{% endfor %}
```

We use Liquid tags to denote a block object, and the attributes of this block are defined in the schema array of each section file. The syntax of a block object would look like `{{ block.settings.id }}`, where `id` would be the attribute referenced using JSON in the schema array. For example, a block to add an image could be `{{ block.settings.image }}`.

Within the array, a block must be assigned a `name` and a `type`. A block's `type` can be any value set by the theme developer. A block has settings in the same format as `settings_schema.json`, for example:



```
{% schema %}
{
  "blocks": [
    {
      "type": "quote",
      "name": "Quote",
      "settings": [
        {
          "id": "content",
          "type": "text",
          "label": "Quote"
        }
      ]
    }
  ]
}
{% endschema %}
```

Within the main schema settings, we can assign the max number of blocks in the section. We have set this to three, but it can be any number. Depending on the type of output, you may want to limit or “cap” the possible number of blocks differently, so that a page does not get cluttered.

Below this, because it's a dynamic section, we have `presets`, which will allow this section to be added to the index page. We can define how many blocks appear by default, by adding blocks within the `presets`. This means two call to action buttons will appear, and since we set the max to three, an additional block can be added.

Using “case/when” control flow tags with blocks

By making use of the case/when control flow tags, we can set up different options for including types of output. For example, if we wanted a section to have block options for custom text or a newsletter signup form, the code for these block options could look like this:

```
{% for block in section.blocks %}
  {% case block.type %}

    {% when 'text' %}
      <div class="grid__item {{ column_width }}">
        <h3 class="h4">{{ block.settings.title }}</h3>
        <div class="rte">{{ block.settings.richtext }} </div>
      </div>

    {% when 'newsletter' %}
      <div class="grid__item {{ column_width }}">
        <h3 class="h4">{{ 'layout.footer.newsletter_title'
          | t }}</h3>
        <p>{{ 'layout.footer.newsletter_caption' | t }} </p>
        {% render 'newsletter-form' %}
      </div>

  {% endcase %}
{% endfor %}
```

With great power, comes great responsibility

Now that you've seen how easy it is to add sections to your themes, you can add endless options to your clients' stores.

However, it's worth keeping in mind the possible risks of repeating blocks, especially for elements such as images and videos. Over-repeating these could result in slow page loading times and a poor user-experience for customers, which could have a negative effect on conversions.

But by implementing blocks carefully, and considering their context, you can create a winning formula for your client

CHAPTER 10

How to Use `all_products` in a Shopify Theme

This chapter will explore a way to access product information without having to loop over a collection or be on a product detail page.

We can achieve this by using `all_products` . Here's a quick example:

```
{{ all_products[ "coffee-cup" ].title }}
```

Let's have a look at what's happening. The syntax is pretty simple: `all_products` takes a quoted product handle as its argument.

Liquid handles

If you aren't familiar with handles, the [Shopify Help Center](#) provides a great explanation:

Handles are used to access the attributes of Liquid objects. By default, a handle is the object's title in lowercase with any spaces and special characters replaced by hyphens (-). Most objects in Shopify (products, collections, blogs, articles, menus) have handles.

In the above example, we have a handle of `coffee-cup` , which represents the product available at `yourstore.com/products/coffee` . We follow that by `.title` . When rendered, this will output the title of the product with the handle of `coffee-cup` .

Using `all_products` we can access any property of the product:

- `all_products[ "coffee-cup" ].available` `all_products[ "coffee-cup" ].collections`
- `all_products[ "coffee-cup" ].compare_at_price_max`
- `all_products[ "coffee-cup" ].compare_at_price_min`
- `all_products[ "coffee-cup" ].compare_at_price_varies`
- `all_products[ "coffee-cup" ].content`

- `all_products["coffee-cup"].description` `all_products["coffee-cup"].featured_image`
- `all_products["coffee-cup"].first_available_variant`
- `all_products["coffee-cup"].handle`
- `all_products["coffee-cup"].id` `all_products["coffee-cup"].images`
- `all_products["coffee-cup"].image`
- `all_products["coffee-cup"].options` `all_products["coffee-cup"].price`
- `all_products["coffee-cup"].price_max` `all_products["coffee-cup"].price_min`
- `all_products["coffee-cup"].price_varies`
- `all_products["coffee-cup"].selected_variant`
- `all_products["coffee-cup"].selected_or_first_available_variant`
- `all_products["coffee-cup"].tags` `all_products["coffee-cup"].template_suffix`
- `all_products["coffee-cup"].title` `all_products["coffee-cup"].type`
- `all_products["coffee-cup"].url` `all_products["coffee-cup"].variants`
- `all_products["coffee-cup"].vendor`

Note that some of the returned values will be a Liquid collection, and because of this would need to be “looped” over. Let’s use the images collection as an example:

```
{% for image in all_products["coffee-cup"].images %}

{% endfor %}
```

This example would output all of the images associated with the coffee-cup product.

More than one handle

You can go one step further and create a simple Liquid array of handles that you can use to output specific products.

Here’s an example:

```
{% assign favorites = "hand-made-coffee-tamper|
edible-coffee-cup" | split: "|" %}
<ul>
{% for product in favorites %}
<li>{{ all_products[product].title }}</li>
{% endfor %}
</ul>
```

Using the Liquid assign tag, we create a new variable called **favorites**, which are product handles separated by a `|` character. The `|` is used as a delimiter to divide the string into an array that we can loop over using `for`.

We now have access to both products in turn and can output any property associated with it — in the example above we simply display the title.

When to use `all_products`

`all_products` is a great option when you need to pull out a couple of products in a particular template. Of course, if you are outputting a lot of products, a collection is still the best way forward, as you won't have to manually know all the different product handles. However, `all_products` makes a great option when you need to output a single or small number of products that won't change frequently.

Manipulate Images with the `img_url` Filter

In this chapter, we'll look at how to use the `img_url` filter and examine the recently added parameters that allow you to manipulate images within Shopify in new and exciting ways.

Let's begin by looking at the function of the `img_url` filter. In its basic form, it will return the URL of an image. In order to do this, you need to pass in a mandatory size parameter. It's also quite a versatile filter as it can be used with the following objects, which have images associated with them:

- product
- variant
- line item
- collection
- article
- image

We'll focus on using the product object in this chapter.

Here's an example:

```
{{ product.featured_image | img_url: '100x100' }}
```

In the example above, the `img_url` filter has a parameter of `100x100`. This value corresponds to a particular size of image that was created automatically by Shopify, after it was uploaded via the Shopify Admin.

In this case, the image will be no bigger than 100x100 pixels.

If you upload a square image, it will be perfectly resized. However, if your original image is longer on one side than the other, Shopify will resize accordingly so that the longest side will be 100 pixels.

In other words, all resizing is proportional unless you `crop` the image.

Here's the list of sizes with their corresponding image names:

1024 x 1024 (width and height)	1024x1024
Width only	100x
Height only	x100
Largest / original image	master

You can also chain the `img_url` filter with the `img_tag` filter to output the full `<img>` element:

```
{{ product.featured_image | img_url: '100x100' |  
img_tag }}
```

So far, we've looked at the basic function of the `img_url` filter. There wasn't much more you could do with the `img_url` until a new set of parameters were added, making it possible to resize and crop images from within your template files.

Additional parameters

Before moving on, it's worth noting that the following techniques can be used with a range of filters in addition to `img_url`. They are:

- `product_img_url`
- `collection_img_url`
- `article_img_url`

We'll use `img_url` in all the following examples, but we want to highlight that the techniques work with the three other filters, too.

1. Size

Let's begin by looking at how we can resize an image. In order to do this, we replace the image "name" with a specific size in pixels.

Here's an example:

```
{{ product.featured_image | img_url: '450x450' }}
```

[View generated image](#)

The “names” mentioned above will of course work as they always have. However, using the above syntax puts the control of the image dimensions in your hands. In this example, we've specified both the width and height (in that order).

You can also specify only a width, or only a height.

Width only:

```
{{ product.featured_image | img_url: '450x' }}
```

[View generated image](#)

Height only:

```
{{ product.featured_image | img_url: 'x450' }}
```

[View generated image](#)

When only specifying a single value, Shopify will calculate the other dimension based on the original image size, keeping the original image's aspect ratio intact.

Going back to our original example, you might think that it would result in a `450x450` version of your image being rendered.

This, however, isn't always the case.

This request would result in a perfect square only if both of the following conditions are met:

1. The original image was `450px` or greater on both axes.
2. Both sides are of the same length.

If both conditions are true, a `450x450` square image will be rendered. If not, Shopify will resize it using the same logic as if you've specified only height or width. The longest side wins out in this situation and is scaled accordingly.

2. Crop

Thankfully, creating perfect squares won't require you to upload square images. All that it requires is the addition of another new parameter called `crop`. You specify a `crop` parameter to ensure that the resulting image's dimensions match the requested dimensions. If the entire image won't fit in your requested dimensions, the crop parameter specifies which part of the image to show.

Valid options include:

- top
- center
- bottom
- left
- right

Here's an example building on the one we discussed earlier:

```
{{ product.featured_image | img_url: '450x450',  
crop: 'center' }}
```

[View generated image](#)

3. Scale

As well as dimensions, we can also request a certain pixel density using the scale parameter.

The two valid options are:

- 2
- 3

You can simply add this as another argument to the `img_url` filter as follows:

```
{ product.featured_image | img_url: '450x450', crop: 'center', scale: 2 }}
```

[View generated image](#)

This would result in a resized image of `900x900` pixels. Again, this will only be scaled up if the original image is large enough. If this isn't the case, the closest image in size will be returned.

4. Format

There's one final parameter you can add, which is `format`.

Valid options for this are:

- `jpg`
- `pjpg`

Here's an example incorporating `format`:

```
{{ product.featured_image | img_url: '450x450', crop: 'center', scale: 2, format: 'pjpg' }}
```

[View generated image](#)

This would result in the image being rendered as a progressive JPG — these load as a full-sized image with gradually increasing quality, as opposed to a top-to-bottom load. It's a great option to have depending on your needs.

Shopify can do the following format conversions for you:

- PNG to JPG

- PNG to PJPG
- JPG to PJPG

It's not practical to convert a lossy image format like JPG to a lossless one like PNG, so those conversions aren't possible.

Caching

Finally, it's worth noting that once the requested image has been created, it will be cached and made available on the Shopify CDN (Content Delivery Network). Consequently, there's no need to worry about the image being created every time your template is rendered.

Conclusion

Thanks to these new parameters, it's now possible to implement responsive image techniques in your templates. Whether you want to start using the `srcset` and `sizes` attributes, or the `<picture>` element, you can start offering the most appropriate image for screen size, resolution, and bandwidth.

Ways to Customize the `img` Element

Now we're going to have a look at the humble HTML `img` element.

When creating a Shopify theme, you can add any number of images, in any format, and at any size to the `assets` folder within your theme directory. Typically, these images are used for backgrounds, sprites, and branding elements.

Referencing these images in a theme is very straightforward.

Let's assume we have a `logo.png` in our `assets` folder. We can output this image in any template using the following Liquid syntax:

```
{{ 'logo.png' | asset_url | img_tag: 'Logo' }}
```

This approach uses two Liquid filters to create a fully formed HTML `<img>` element. The first, `asset_url`, prepends the full path to the assets folder for the current store's theme, while the second, `img_tag`, uses this URL and creates an HTML `<img>` element complete with the `alt` attribute. If omitted, the `alt` attribute will be blank.

Here's the end result:

```

```

You'll notice that the `src` attribute references the Shopify CDN. Every image that you add, regardless of its type, will be pushed out to the Shopify CDN. You'll never need to worry about the location of your images, as the `asset_url` filter will work this out for you when the page is rendered.

Adding classes to the `img` element

In the example above, we added in the `alt` attribute. It's also possible to add a further parameter that allows you to add classes to the `<img>` element. Here's our example refactored:

```
{{ 'logo.png' | asset_url | img_tag: 'Logo',  
'cssclass1 cssclass2' }}
```

This would result in the following output:

```

```

More control

There will of course be occasions where you need a little more control over the markup. By simply omitting the `img_tag` filter, we can build up our markup as we wish.

Here's an approach that would allow you to add your own attributes such as an `id` :

```

```

We hope you found these examples useful in your own theme building.

Creating Useful CSS Hooks in Liquid

Many of us use the `<body>` class for CSS as well as JavaScript hooks and, just like in WordPress, it's pretty easy to add a number of useful classes to our `<body>` element in Shopify.

Here are a few ideas that you might find useful placing in your main (or alternate) layout file.

Add the currently rendered template name to the body class

```
<body class="{{ template | handleize }}">
```

In this example, we're using the `template` object to return the name of the currently used template. This can be really useful when you need to target a specific alternate template, for example. Some examples of this are:

```
<body class="index">  
<body class="product">  
<body class="collection">
```

Add the currently rendered product handle to the body class

Building on this, we may wish to add the current product `handle` to our body class. To keep things neat and tidy, we can use an `if` statement to conditionally add the product handle only when we're viewing a product:

```
<body class="{{ template }}{% if template ==  
"product" %}{{ product.handle }}{% endif %}">
```

Note how we include the space before the `{{ product.handle }}` output tag. This is to ensure that classes are separated by spaces and rendered individually.

If you are using alternate product templates, you may wish to use the `contains` operator instead:

```
<body class="{{ template }}{% if template contains  
"product" %}{{ product.handle }}{% endif %}">
```

Add the current page title to the body class

Some themes also add the current page title to the body element in the form of an `id`. Building on the above, our code would now look as follows:

```
<body id="{{ page_title | handleize }}" class="{{
template }}" {% if template == "product" %} {{ product.
handle }} {% endif %} ">
```

Note that in this example we're using the Liquid filter `handleize` to ensure that the `id` or `class` that we add is URL safe and therefore easy to reference in our CSS and JS files. For example, it will turn a page title of "Blue Jeans" into "blue-jeans".

Add the currently viewed collection's name to the body class

For good measure, we could even add in a check for collections:

```
<body id="{{ page_title | handleize }}" class="{{
template }}" {% if template == "product" %} {{ product.
handle }} {% endif %} {% if template == "collection" %}
{{ collection.handle }} {% endif %} ">
```

It's pretty easy to adjust this logic for your own purposes.

Again, you may wish to use the `contains` operator if you are utilizing alternate templates.

Summary

Hopefully you've seen how flexible Liquid is in the above examples. Being able to add a variety of classes to the `<body>` element gives us useful hooks that we can use in CSS and JavaScript.

Using Liquid's **case/when** Control Tags

We're sure many of you are more than familiar with Liquid control tags such as **if** and **else**, but are you familiar with **case/when**?

Here's how the Shopify Help Center describes it:

Case/when *creates a switch statement to execute a particular block of code when a variable has a specified value. **case** initializes the switch statement, and **when** statements define the various conditions.*

Here's an example:

```
{% assign handle = 'cake' %} {% case handle %}
{% when 'cake' %}
  This is a cake
{% when 'cookie' %}
  This is a cookie
{% else %}
  This is not a cake nor a cookie
{% endcase %}
```

In this instance, the output will be determined when the variable called `handle` is equal to “cake” or is equal to “cookie”. If neither condition evaluates to `true`, it will output the text after the last `else` clause.

If you omit the `else` clause and the handle variable never evaluates to `true`, no output will be output. This is because the `else` clause acts as a fallback in the above example.

Real world example

As nice as our example is, you might be wondering when you might use this in your own theme development.

One example we've seen used in the past is in relation to banners in a theme. Despite our love of alternate templates, there are occasions when creating a variety of templates simply to display different promotional banners would be very time consuming. Not only would you have to build the templates, but you'd also have to assign them to each product in turn. A more practical approach is to let Shopify do the heavy lifting for you.

Let's say we wanted to display different promotional banners on particular products. One way we could do this is to use product

handles and `case/when` . This code example will work in a `product.liquid` template.

```
{% assign handle = product.handle %}
{% case handle %}
  {% when 'coffee-cup' %}
    {% render 'promo-coffee-cup' %}
  {% when 'cup-saucer' %}
    {% render 'promo-cup-saucer' %}
  {% else %}
    {% render 'promo-default' %}
{% endcase %}
```

We start off by creating a variable called `handle` and assign it the current value of `product.handle` . Next we instantiate our `case` clause, followed by a series of `when` statements.

In our example, if our product handle is equal to `coffee-cup` , the snippet titled `promo-coffee-cup` will be rendered and Shopify will head right to `endcase` and carry on.

Alternatively, if the product handle is equal to `cup-saucer` then the snippet titled `promo-cup-saucer` will be rendered. If the product handle is neither `coffee-cup` or `cup-saucer` , then the `else` clause kicks in and the snippet titled `promo-default` will be rendered.

We have achieved quite a lot with a little. We're conditionally outputting different snippets depending on the product being viewed and outputting a default promotional banner if neither condition is met. We've also achieved this without having to create alternate templates. To extend the example, you could simply add in further specific product handles when needed. However, an alternative approach might be needed if you wanted to include tens of different banners.

The Shopify Partner Program

Building a profitable business with Shopify

At Shopify, we not only provide a product that merchants love, but offer a partnership you can build your business on.

Shopify is a commerce platform that lets entrepreneurs design, set up, and manage their store across multiple channels, including through an ecommerce website, social media, brick-and-mortar retail locations, and more. Shopify currently powers over 1 million businesses across 175 countries, and is trusted by brands like Unilever, Kylie Cosmetics, Allbirds, MVMT, and many more.

With thousands of integrations available in [the Shopify App Store](#), your client can access the unique functionality they need to get set up. And with one back office for all their sales channels, managing their business is easier than ever.

In addition to supporting merchants, we also understand the unique challenges that come with running your own web design and development business. As a freelancer, consultant, or agency working with commerce clients, you can join [the Shopify Partner Program](#).

This vast and collaborative network of international designers, developers, marketers, and consultants use Shopify as the commerce tool of choice for their clients.

Becoming a Shopify Partner is free, and gives you access to:

- Development stores to test the platform and build your client's site
- Revenue share opportunities
- Developer preview environments
- Priority partner support
- Educational resources
- And more

Together, these opportunities have enabled thousands of Shopify Partners to learn from one another, and build successful and profitable businesses.

We hope you'll join us.

Become a Shopify Partner