Mapbox Cookbook

The Mapbox platform offers all the tools and APIs required to create and publish a totally customizable map. Starting with building your first map with the online Mapbox Editor, we will take you all the way to building advanced web and mobile applications with totally customizable map styles. In the initial few chapters we will dive deeper into the TileMill and Mapbox Studio components of Mapbox and use them to generate custom styled map tiles and vector maps. Furthermore, we will publish these custom maps using PHP, Node.js and third-party tools such as GeoServer. By the end of this book, you will learn about Mapbox GL and how to create a fully functional, location-aware mobile app.

What this book will do for you...

- Get accustomed to the Mapbox Editor to visually style your maps
- Learn everything about CartoCSS, and how it will help you fine tune your styled maps
- Use Mapbox Studio and TileMill to generate your own tiles and vector maps
- Publish your maps using a variety of technologies such as Node.js, PHP, and GeoServer
- Integrate with third-party APIs and services to populate your maps with public or private data

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions to solve problems efficiently
- Clear explanations of what you did
- Solutions that can be applied to solve real-world problems

Quick answers to common problems

Over 35 recipes to design and implement uniquely styled maps using the Mapbox platform

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'TileMill and Mapbox Studio'
- A synopsis of the book’s content
- More information on Mapbox Cookbook
About the Author

**Bill Kastanakis** has been a software engineer for over 15 years with experience in desktop, mobile, and web application development. His love for the Mapbox platform came from a project he worked on in 2010 to build a series of iOS tourist guide apps. With maps being simplistic in appearance and their customization options being nearly zero, he discovered the potential of a platform that allows the developer to totally customize the appearance of maps and the experience of using them.

Bill currently owns MindBomb, which specializes in mobile apps for the iOS platform, and he is a cofounder of nCoded+ Limited, which builds enterprise mobile and web applications for casino accounting. He also works as an iOS team lead and architect for one of the most popular apps in Ireland.

Bill often contributes to presentations as well as writing several mobile development blogs about new mobile technologies.
Maps are an essential element in today's location-aware applications, but they lack variation and customization. The Mapbox platform offers the tools and APIs required to totally customize, populate, and publish a map.

In this book, starting with the basics of Mapbox Editor for your first map styling steps, we will take you all the way to building advanced web and mobile applications with completely customizable map styles. The book focuses on the CartoCSS styling language as well as Mapbox tools and its JavaScript API, which inherits from Leaflet and is one of the most established, robust, and easy-to-use libraries.

We will then introduce two core Mapbox tools: TileMill and Mapbox Studio. Using them, we will generate custom-styled map tiles and vector maps. We will then move on to how to publish your custom maps using PHP, Node.js, and third-party tools such as GeoServer.

The next step is to start using the Mapbox JavaScript API and Leaflet to create different visualization map styles, such as a choropleth map and a heat map, and add user interactivity using UTFGrid.

We will continue with the advanced chapters and focus on integrating with third-party services such as Foursquare, Google Fusion Tables, CartoDB, and Torque to help us populate and even animate our maps.

Finally, we will end the book with a chapter dedicated to mobile devices. You will learn about Mapbox GL and how to create a fully functional, location-aware mobile app, which will use the map styles created in the earlier recipes.

This book is fast-paced, and the recipes are easy to follow. While it focuses on a recipe approach, it dives into the core concepts and theory of the technologies used to help you understand the theory required for GIS, web, and mobile development.
What this book covers

Chapter 1, Introduction to Mapbox, shows how to style your own base map using Mapbox Editor, add vector data, and publish your maps.

Chapter 2, Mapbox Services, shows how to use the lower-level Mapbox Web Services API to access data from the Mapbox servers.

Chapter 3, TileMill and Mapbox Studio, shows how to create stunning custom raster and vector maps using TileMill and Mapbox Studio.

Chapter 4, Mapbox.js, is an introduction to the Mapbox JavaScript API. This chapter will show how to create custom layers, add vector and raster data, and create interactivity and custom map styles such as choropleth and heat maps.

Chapter 5, Mapbox.js Advanced, shows how to use external data sources and integrate them with datasets from third-party services.

Chapter 6, Mapbox GL, shows how to create a fully functional mobile app using Mapbox GL for iOS.
In this chapter, we will cover the following recipes:

- Styling a map with TileMill
- Styling a map with Mapbox Studio
- Publishing your base map on your server with PHP
- Publishing your base map on your server with Node.js

**Introduction**

Up until now, we have used Mapbox Editor to create and customize our maps. We have seen how feature-rich and powerful it is and how, using just a web interface to customize your map, it provides you with so many possibilities. We were able to totally customize the color theme, add markers and features, and customize their appearance.

Although powerful, there are times when you need more control. What if we want to change the widths of the lines that represent roads? What if we want to totally customize this width over different zoom levels? What if we want a custom font, or even better, a different font (or font size) at each zoom level? What if we want to hide or show features when a user zooms in or out?

The possibilities are endless, and the power of unlocking this level of customization lies in a language called CartoCSS. Well, it's not exactly a language like C++ or PHP, with objects and complicated memory management, but it is a much more user-friendly, stylesheet-like language that shares a lot of similarities with the well-known CSS.
Let's look at an overview of what we have just learned. We know that CartoCSS is an easy-to-use language that allows us to fully customize our maps. Does this sound too good to be true? There's more!

There are a lot of ways to write CartoCSS, and Mapbox provides us with two awesome tools that help us create, edit, and share our maps.

The first one is called TileMill, which is a well-known solution that has been offered for a number of years now. The other candidate is the new and shiny Mapbox Studio.

Let's begin by explaining what TileMill is and what it is useful for.

### Understanding TileMill

TileMill is a standalone application that helps us add various layers of (map or vector) data, write CartoCSS, preview the results, and finally export the tiles:

To better understand how it works, we will explain the workflow to create, customize, and publish our own custom-styled map.
If you open TileMill and create a new project, you will notice that it only shows a global map with coastlines. If you try to zoom in, you will see that there are no roads, labels, or other details—just a solid white color for land and blue for water. In order to start working with TileMill, you need to import the data you need.

TileMill supports a variety of data from different sources; for example, it allows us to import shapefiles, which contain geographical points, lines, or polygons. Using shapefiles, you can import features such as roads, coastlines, and buildings. Shapefiles support a single geometry type.

It also supports GeoJSON, a flexible file format that supports different geometry types. Many agencies provide data in this format; for example, if you request data for earthquakes in the last $n$ days, you will receive a GeoJSON file that includes points indicating the recent earthquake locations.

You can even create images, such as GeoTIFF files, with data that you may want to overlay on your maps. This feature is extremely flexible; for example, you may want to overlay weather data, such as clouds, on top of your maps.

KML is another supported format popularized by Google. However, TileMill does not support many of the features of KML, such as images, flythroughs, and embedded styles, so keep this in mind when importing KML files.

TileMill also supports importing data from databases such as SQLite and PostGIS, which is an extension of PostgreSQL. You can connect directly from TileMill to a database and run queries.

Each separate file must be imported in a separate context, which TileMill calls a layer. Each layer can then be separately styled using CartoCSS. We can use the attributes found in the layer and style them differently. Once we have imported and styled all the layers, the last step is to export the map in order to publish it. We cannot simply export a huge image from it because, even for tiny areas with a zoom level of 15 or similar, it would export an immersive image in a size that would not be possible for a normal computer to read due to memory limitations.

For this reason, TileMill splits this huge image into separate tiny tiles. These tiles are then uploaded to Mapbox and displayed to the user.

If you have ever noticed a map loading on a slow connection, you must have seen tiles appearing as the map loads.

Well, at this point, I imagine that you have begun to understand how powerful the whole process must be to import any data we want in different formats, combine them all together using layers, and style them individually with absolute control!
Now that you understand how TileMill works and how awesome it is, it's time for us to discuss the bad side of it. First of all, there is no easy way to import data that you want in it. You cannot just say, "let's zoom to San Francisco and style the map". You will have to import San Francisco into TileMill with all the features required, such as roads, water, labels, and buildings. Well, TileMill understands specific formats, so finding the exact data needed or extracting it from a source, such as OpenStreetMap, in a format that TileMill understands may be a tedious task. Luckily for us, there are websites that do exactly this, at least for major cities, so you may find everything you need on a website.

The second major drawback of TileMill is exporting the tiles. Exporting tiles even for small areas at a reasonable zoom level may require thousands of images to be generated. These images take time to be exported, consume large amounts of hard drive space, require huge storage on the server that we plan to host them on (for example, Mapbox), and also require a lot of time to upload hundreds of megabytes of tiles to this web server, especially if we have a low upload speed. The overall process may take from minutes to hours and even days, depending on the area's size and zoom level.

Luckily, we are talking about TileMill, which the Mapbox folks consider to be "old technology". For this reason, they developed Mapbox Studio.

Let's now see what Mapbox Studio is all about and how it compares with TileMill.

### Understanding Mapbox Studio

Mapbox Studio was released a while ago to the public, and it is supposed to replace TileMill over time. Mapbox Studio uses vector data to represent maps, and, unlike TileMill, it's not based on bitmap tiles:
Let’s find out how Mapbox Studio works.

We learned in the previous section that we need to import our data into TileMill. Well, this is not the case with Mapbox Studio because we immediately have access to Mapbox Streets, Mapbox Terrain, and Mapbox Satellite data. We do not need to search for the region of interest, extract it, and use it, as we do in the case of TileMill. With this simple choice, we can have access to data from the entire world down to the finest detail.

We can use Mapbox Streets as a source; this provides us with the vector data required to build a map. We can also select one of the other sources of Mapbox, such as Mapbox Terrain or Mapbox Satellite. We can even combine all these sources and overlay them one on top of the other to use the features provided by these different sources.

You may ask, what if I want to import and use my custom data and then overlay it on top of these maps? Well, this is easy too. You can import your own data into Mapbox Studio by converting data from traditional formats, such as those supported by TileMill. Once it is imported, you can then upload it to your Mapbox account, and it will immediately become available for use as a source in Mapbox Studio. It can then be overlaid on top of other map data.

Mapbox Studio works with vector tiles, and because of this, we do not need to export the resulting styled map in separate tiles. With Mapbox Studio, we will simply upload our own custom-created map into our Mapbox account—just like that! We don't have the hassle of waiting for hours for the tiles to be exported, neither do we need tons of hard drive space, and we are not required to have a high upload speed in order for the upload to be completed in reasonable time. The whole process is nearly instantaneous.

In spite of the importing and exporting capabilities of Mapbox Studio and the differences from TileMill in the technologies that it uses, such as the vector tiles, we still find a CartoCSS editor, and we can style our map using CartoCSS just as with TileMill.

Another handy feature of Mapbox Studio is the ability to automatically adapt to high-resolution displays (retina displays). In the case of TileMill, we need to export a separate double-sized tileset to use with these displays, causing the exported time, hard drive space, and upload time to double.

Mapbox Studio provides us with all the typography that we should need. It comes bundled with 300 professional quality fonts to use on the maps.

Fonts are provided free of charge for exclusive use in maps with Mapbox Studio. It's actually not allowed to use them in any other case.
What about exporting high-resolution images to print? Mapbox Studio has the capability to export images up to 600 dpi with one click:

Now, we have come to a point where you may ask, "what's the catch?"

Using Mapbox Studio is not free; it's a paid service. There is a starter plan offered by Mapbox, which is free, but it's very limited in what you can do with it. Mapbox plans are based mainly on map views, storage, and the custom styles you can host.

The starter plan at the time of writing this book, apart from allowing 3,000 views per month and 100 MB storage, has just one custom style, which means that we cannot have more than one custom dataset overlaid on our vector maps. Luckily, Mapbox introduced a basic plan for just $5 per month, which offers three styles, 1 GB storage, and 10,000 views per month, so anyone can now start working with it without being limited. A while ago, Mapbox had a standard plan for $49 per month. While this was not a showstopper if you planned to use what Mapbox offers professionally, it was for people who just wanted to explore Mapbox Studio's capabilities.

By this time, you may be thinking that if Mapbox Studio and other Mapbox technologies are currently open source, what stops us from hosting the maps in our own servers? Well, nothing stops us, but deploying the whole Mapbox stack to our own server is extremely complicated and beyond most people's skills.
Introducing CartoCSS

The time has come to explain how we can fully customize our maps to create the style we want.

CartoCSS is a powerful stylesheet-like language powering TileMill and Mapbox. It's extremely powerful and only limited by our imagination. If you are familiar with CSS for the Web, you will feel right at home when you start working with CartoCSS. If you have never developed for the Web, don't worry at all. Learning CartoCSS is extremely easy and fun.

Let's begin by explaining how it works.

How CartoCSS works

CartoCSS, like CSS, is based on selectors. A selector, for example, can be an ID of a specific layer, a zoom range, or a layer class. An ID may be called #roads, which means that whatever attributes we change within the closures will only affect roads.

Once we have our selector defined, we have to specify which attributes we want to modify in this context. For example, we may want to change the street outlines to a different color, so the attribute in this case is line-color.

Let's consider an example selector in action:

```
#sf-lines[highway="motorway"] {
  line-width: 5;
}
```

Now, let's dissect this example.

#sf-lines is the name of the layer. We can name the layers at the time we import them into TileMill however we want.

In the square brackets, there is [highway="motorway"], which means that we want to find the highway feature in this layer and select all the motorways.

Within the curly brackets (which are called closures) is what we want to modify. In this example, it's all the motorways in the sf-lines layer.

The line-width: 5 part of the code means that we want to set the line width to 5 pixels.
Observe the \#sf-lines layer name in the Layers management dialog in the lower-left corner of the screen.

Certain attributes can only be applied to a specific type—which are called symbolizers—for example, line-width and line-color. These refer to the thicknesses and colors of lines and can obviously be applied to lines only, while others, such as fill-color, can only be applied to polygons.

The style types are called symbolizers in the Mapbox language (which originates from Mapnik, a technology used to render maps).

There are currently 10 symbolizers in TileMill, and each one can be applied to a certain type of geometry. The symbolizers are as follows:

- Line (lines and polygons)
- Polygon (polygons only)
- Point (points only)
- Text (points, lines, and polygons)
- Shield (points and lines)
- Line pattern (polygons)
- Raster (rasters)
- Marker (points, lines, and polygons)
- Buildings

We can repeat this process to specify as many selectors as we want, and in each one, we can change as many parameters as we want.

**Filters**

CartoCSS allows us to use filters in order to specify ranges; for example, we can use the \([\text{zoom} = 10]\) filter to apply styling only at zoom level 10.

Let's consider an example of how we can use zoom filters:

\[
[\text{zoom} < 16] \{ \\
\quad \#\text{sf-lines}[\text{highway}="primary"] \{ \\
\quad \quad \text{line-width: 3;} \\
\quad \} \\
\}
\]

In this example, the first selector is \(\text{zoom} < 16\), which means only the instances when the zoom is less than 16. From the \#sf-lines layer, we select all the primary roads. Then we set line-width to 3.

In the preceding example, we used zoom to adjust the line width of the roads. As you may have noticed in various mapping services, maps usually display less detail at zoom levels as you zoom out, which happens because they don't want to overwhelm the user by displaying information that does not make sense at these zoom levels.

Imagine a map of the United States zoomed out to a level where every state is visible on the screen. What would happen if we were to display, at this specific zoom level, every street name down to the smallest walk path?

**Comparisons**

As you may have noticed in the preceding section, we used comparison filters to specify the criteria. We have already used \(\text{highway}="\text{motorway}"\), which is a text filter, and \([\text{zoom} < 16]\), which is a zoom level filter.

Let's take a look at some more examples to understand the power of filtering our content.

For zoom level filters, we can limit the zoom levels to a specific range using the following code:

\[
\#\text{sf-lines}[\text{zoom}=>5] [\text{zoom}<=12]
\]
This simply means the instances when the zoom is between 5 and 12.

We can do the same using numeric filters. Here's an example:

```javascript
#earthquakes[magnitude > 4][magnitude < 7]
```

Here, from the `earthquakes` layer, we are selecting the points between magnitudes 5 and 6. Note that the equals sign is missing.

For text comparisons, we can use the following code:

```javascript
#sf-lines[highway != "primary"]
```

From the `#sf-lines` layer, select everything except primary highways. Alternatively, you can use the `=~` operator to specify a regular expression, as follows:

```javascript
#sf-lines[highway =~ ".* Highway"]
```

In the preceding example, both major and secondary highways will be selected.

**Working with colors**

We can define a color by directly setting the value next to the property, as follows:

```css
background-color: #b5e3ff
```

This will set the background color to blue. The color value is represented in hex.

To simplify our work and have an easier time when tweaking or fine-tuning our design, CartoCSS allows us to use variables to define colors (and many other values), as follows:

```css
@water: #b5e3ff;

Map {
    background-color: @water;
}

Water {
    fill-color: @water;
}
```

At the beginning, we defined a variable called `@water` with a hex value of `#b5e3ff`. We set this variable as the color value for both the background and the fill color of water. This will save us from replacing multiple values within the editor when we need to make a change.
In the TileMill editor, colors are also represented with a color swatch directly below the text. You can simply click on the swatch, a color palette will appear, and you can select a color you like.

**Styling lines**

Let's explore some common techniques of styling lines.

The attributes we already know for lines are `line-color` and `line-width`. Another useful attribute is `line-join`. It changes how lines appear at the points where they join other lines.

Let's consider an example of how it looks if we don't set this attribute at all, which means that it uses the default `line-join` parameter, `miter`:
Now, we will set `line-join: round`:

You can also try setting it to `bevel` and spot the differences as compared with the preceding screenshots.

Another useful parameter is `line-cap`, which sets how lines look at the ends. You can set it to `butts` (default), `round`, and `square`:

The `line-dasharray` property can take a comma-separated list of pixel widths as a value. Each value represents a dash and a space in this order. For example, `line-dasharray: 24, 12` will create a dash of 24 pixels and a space of 12 pixels.
You can set as many values as you like to create more complex styles; for example, line-dasharray: 24, 12, 15, 7, 12, 3 is perfectly valid.

Note that we combined line-dasharray with line-cap and line-join to get the rounded look.

**Styling polygons**

Polygons are filled areas. We usually style them with a solid color or pattern. Polygons can also be styled with the attributes that we saw before, such as line-width and line-color, but let’s explore some new ones that work specifically for polygons.

The most common one is polygon-fill. It simply fills the polygon with a solid color. Take a look at the following code:

```css
#countries {
    polygon-fill: #aaa;
}
```

We can specify the opacity of the polygon using the polygon-opacity attribute, as follows:

```css
#countries {
    polygon-fill: #aaa;
    polygon-opacity: 0.50;
}
```

If a layer is opaque, the layer behind it becomes visible. If there is no layer behind it, the background color becomes visible.
We can fill a polygon using a texture or seamless pattern. We can also create our own patterns using image processing software, such as Photoshop or Gimp. Another option is to download patterns from a website such as http://subtlepatterns.com or http://www.patterncooler.com/.

We can define the pattern we want to use using the `polygon-pattern-file` style. It takes the URL of an image file as a parameter, as follows:

```css
#countries {
  polygon-pattern-file:
    url("http://www.mysite.com/pattern01.png");
}
```

Have a look at the following screenshot:

This pattern is only applied to the polygon that we defined within our selector. To apply patterns in the background, we can use the `background-image` property, as follows:

```css
#countries {
  background-image: url("pattern01.png");
}
```
Styling labels

It's almost certain that at some point, we will need to style labels. By this, we mean how the text for road names, cities, or countries will look.

The most basic form of styling text can be found in the following example:

```javascript
#countries {
  text-name: [NAME];
  text-face-name: 'Arial Bold;
}
```

First of all, we need to fetch text from our layer data. We can do this using `text-name: [NAME]`, where `NAME` is the attribute name of our layer.

We also need to specify `text-face-name`. This is the font name that will be used by TileMill to display the text on the screen. This is not the font filename but how the font is named internally. You can use the fonts browser within TileMill to browse the available fonts in your system and see their internal names.

We can change the text color using the `text-fill` property and the text size with the `text-size` property.

There is a property called `text-halo-fill`, which in simple words is the outline color of the font; however, in order for it to be visible, you have to also specify the `text-halo-radius` value with the radius in pixels.

A complete styled text will look similar to the following example:

```javascript
#countries {
  text-name: [name];
  text-size: 16;
  text-fill: @text;
  text-face-name: "Arial Bold";
  text-halo-radius: 1;
  text-halo-fill: #AC8812;
}
```
Have a look at the following screenshot:

If we look at this screenshot, the street labels are not aligned with the streets. The map looks a bit weird because in many cases we can't tell which label belongs to which street. It would be much better if we could align the labels to the directions of the streets.

To do this, we can use the `text-placement` property. It takes four different parameters, but the most commonly used one is `line`, which aligns the label in the line's direction:
Chapter 3

The complete selector can be found in the following example:

```css
#countries {
    text-name: [name];
    text-size: 16;
    text-fill: @text;
    text-face-name: "Arial Bold";
    text-halo-radius: 1;
    text-halo-fill: #AC8812;
    text-placement: line;
}
```

**Attachments**

You may have noticed in the preceding screenshot that the selector contains another nested selector that starts with two double colons (::). This type of selector is called **attachments**.

It's a way to instruct TileMill to draw the same element again, but this time, we will style it with the properties modified within the selector closures.

If we start using two double colons, we must specify a name. This name can be whatever we want; we can name it so as to remember what this attachment does. Then, in this context, we can specify any properties that we want to modify.

Let's consider an example:

```css
#countries {
    line-color: #0000ff;
    line-width: 6;
    polygon-fill: #dd0000;

    ::second-outline {
        line-color: #ff0000;
        line-width: 3;
    }
}
```

In the preceding example, TileMill draws an outline of the #countries layer, with a width of 6 pixels and the color blue. Then, at ::second-outline, it draws the layer again, but this time with the color red and a width of 3 pixels.
As you can see in the following screenshot, it's a great way to generate glows and other types of effects; the best part of it all is that you can redraw the layer as many times as you want!

The power of CartoCSS lies in the concepts that we explained in the preceding section. By following these simple rules, you can transform your map to whatever you have in your imagination.

**Styling a map with TileMill**

If you haven't downloaded TileMill yet, head over to the Mapbox website and download it right now from https://www.mapbox.com/tilemill/.

The installation is pretty straightforward, and detailed instructions are provided in the page you download TileMill from, for all supported operating systems—Windows, OS X, and Linux.
Chapter 3

How it works...

Once TileMill is installed and you have opened it, you will be greeted with the templates screen, where you can select one of the predefined templates, one of your previous projects, or create a new one:

![TileMill template screen]

Go ahead and select any template you like. We will examine the user interface at this point.
After a template is selected, we will consider the main user interface of TileMill. It is separated into different sections, with the most dominant section being our styled map:

The map will be a live preview, showing the data that we imported into TileMill.

At this point, it is worth mentioning again that TileMill is *not connected to any mapping service* and does not fetch live map data. What we see in the map is the data that we (or the template) imported into TileMill.

In an ideal situation, we could open TileMill, zoom to a region of our choice, and be able to see and style all the features, such as the roads and the buildings. Sadly, however, this is not the case. What we import is what TileMill displays and allows us to style. Remember that it displays the changes after we save the project, not at the moment that the import is completed.

In the preceding screenshot, on the right-hand side is the CartoCSS editor, which displays a huge chunk of code. Don't be afraid; once we understand how CartoCSS works, parsing and understanding this code will be a piece of cake. CartoCSS is not as complex as a full object-oriented language. It's a matter of understanding some basic concepts and rules, that's all.
On the left-hand side of the screen, you will see a dark gray area with some buttons aligned vertically. Let’s check them out one by one:

- **Editor**: This is the first button; it brings you to the exact screen that you are currently looking at.
- **Projects**: This is the screen you saw when you first started TileMill. It is where you select a template or your own project.
- **Manual**: This is just a quick reference explaining the main functionality of TileMill.
- **Plugins**: This allows you to download plugins with extra functionality, which will be integrated directly into Mapbox.
- **Settings**: This is where we set up everything ranging from where your projects are saved, autoupdates, and other aspects of the application.

Let’s head back to the **Editor** tab. In the lower-left corner of the screen, we will find another group of buttons—the Editing Tools. Let’s start exploring them:

- **Templates** is the first button. Here, we can define various aspects of the map that we are styling.
  
The first option allows us to set up a legend, and we can use HTML and inline CSS to design our legend as we like. We can also set up a teaser that shows upon hovering or when tapping on a mobile device. Next, there is a full description tab, and in the end, we can set up the location to which we can also add a link:

```
<svg style="text-align:center;"

<text style="font:12pt/16pt Georgia,serif;">Geography Class</text>
<text style="font:italic 10pt/16pt Georgia,serif;">by MapBox</text>

<img src="data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAJAAAAD/CAYAAAABHNCVQJCAgtfAhkIAAAAAwF1AAAAAEuAABFMiAg/ugQAAAQ1iRQV0U29mdHcucUAa2zLmLu3m7JYXBIj4Z5vuPBoeACi5uSURBVHja7Z0JvIY98d1S1H5ICW8SB
pkvSmsSEy3ob9JAWKe91kOb8Ns880x16m2eYD17I59H53gjJ1f1
UwM7P/6Pp15/vvucu5/nPGf/Ade9en8/6uPc49wx7/5511/q45m
hIG/wWmuu15+qWd6x5x5Ykg2w/CLuACAGj5q/8HdW/2Ov7VYy3z
ohh1j2UPv6f19eruz4pFPRKv69FAigDnAgKX79gtt5apatjgw
Zvk926055wouko0uF8zD6xWhVC1coYn144b566dN8cIfL7db/
```
Fonts is the next button. TileMill opens a window and displays the full list of fonts installed in the system. Using this section, you can easily copy the font name and paste it into the editor. It's kind of hard to remember the full font name, so this section is very helpful:

After Fonts, you will notice an icon that looks like curly brackets. This is a reference to CartoCSS. This is the Carto button.

Do not underestimate this button! It will be your friend while learning CartoCSS! Especially at the beginning, remembering all CartoCSS parameters will be difficult. Do not be afraid to go into this section if you are looking for help:
The last button is the **Layers** button. The most important functionality of TileMill lies here as it gives you access to layers.

Each group of data that you import goes into a layer, and from this section, you can add, edit, or delete your layers. There is also a layer data inspector to view the layer data in a spreadsheet-like screen.
**TileMill and Mapbox Studio**

**How to do it...**

Perform the following steps:

1. Download, or generate using GIS software, the map data required for the map.
2. Create a new TileMill project by clicking on **New project**.
3. Go to the **Layers** palette, click on **Add Layer**, and import the data into TileMill.
4. Style the map using CartoCSS.

**Downloading the map data**

Before we begin with TileMill, let's get the data needed for this recipe. We will download data in an already extracted format. The data contains the shapefiles that we need to import into TileMill. Perform the following steps:

1. A well-known website to get extracted map data is [http://metro.teczno.com/](http://metro.teczno.com/). Head over to this website now and select a city or region:
2. From the region of your choice, you will need two shapefiles. Download the coastline shapefile and the osm2pgsql shapefiles. The first one is only the coastline, and osm2pgsql shapefiles include everything else that we will need, such as roads, building labels, and other features.

We have downloaded the San Francisco files for this example; if you want, you can follow our example, but it really doesn't matter. Any region will do just fine.

3. Extract the ZIP files in a folder of your choice.

4. Each ZIP file will extract several files with extensions such as .dbf, .prj, .shx, and .shp.

**Inspecting the downloaded files using QGIS**

You can totally skip this step and just import everything we downloaded directly into TileMill. However, I strongly suggest that you go through the process and learn how to inspect data.

TileMill shows the main attributes of the file in a spreadsheet-like form. We will use a free open source GIS software to explore the files in detail and the attributes that they contain. Perform the following steps:

1. **Quantum GIS (QGIS)** is an open source software that we can use if we want to take a look at what kind of data a shapefile contains. It's available for every major platform, and it's absolutely free, so go ahead and download it from http://www.qgis.org/en/site/.

The installation instructions can be found on the QGIS website, and they are pretty simple. Go ahead and install it now.

QGIS will install many applications, such as QGIS Desktop and QGIS Browser. We need to open QGIS Desktop.

2. Go to the **Layer** menu, click on Add Layer, and then on Add Vector File. A dialog will come up. Ensure that the **Source type** is File and then click on the Browse button to choose a .shp file. I picked san-francisco.osm-line.shp:
TileMill and Mapbox Studio

3. Click on Open to import the layer into QGIS:

I will not go into a detailed explanation of how QGIS works, as it could be a book of its own. For the purpose of this recipe, we just need to know how to navigate the map. Clicking and dragging on the map area scrolls the map. We can use the scroll wheel to zoom in or out.
4. Click on the Open Attribute Editor icon in the toolbar. This will open a spreadsheet-type editor, which displays all the attributes contained in this shapefile:

![Spreadsheet-type editor](image)

5. Observe the attributes, such as **highway**. We can see that it contains values such as **footway**, **path**, **residential**, and so on. These are the attributes and values that we will use in TileMill to style our map.
6. On the Attribute table, click on the Select Features Using an Expression button. The Select by expression dialog box will pop up:

7. In this dialog box, we can select the values we want in two ways. The first option is to type the expression directly in the Expression section at the bottom of the screen. The second option is to select the Fields and Values category in the Function list section on the left-hand side and then double-click on the attribute that you are interested in—for example, highway. Note that the attribute appears in the expression section at the bottom.

8. Next, select the equals (=) sign from Operators.

9. Click on the all unique button for Load values. All the values of the selected attribute column will appear in the Field values list.

10. Double-click on the value (for example, motorway). The expression editor will display "highway" = 'motorway'.

72
11. Finally, click on the **Select** button and then on **Close** to close the dialog.

12. If you scroll to the attribute editor, you will notice that some rows are now selected; these are all the rows where the **highway** attribute is equal to **motorway**. To view the motorways highlighted in the map, click on the **Zoom Map to the Selected Rows** button in the attribute editor toolbar. It's the one that looks like a magnifier. You can also press **Ctrl + J** in Windows and Linux or **Cmd + J** on Mac.

13. Close the attribute editor, and you will see all the motorways highlighted in the map.

In case you do not understand what an attribute or value represents, head over to [http://wiki.openstreetmap.org/](http://wiki.openstreetmap.org/) and type the attribute or value in the search box on the upper right-hand side. For example, you can see all the values of the **highway** attribute that we just explored at [http://wiki.openstreetmap.org/wiki/Key:highway](http://wiki.openstreetmap.org/wiki/Key:highway).

You have now learned a powerful way to inspect shapefiles and their attributes. If you are in the GIS business, you probably already have this skill. If you are just a developer with the ultimate purpose of creating the most beautiful maps in the world, then you may have never encountered GIS software before.

I strongly encourage you to learn how to use QGIS or any other GIS software in this case. There are many times that you may receive data from other sources in a non-standard format with unknown attributes.

Without this type of software, it is **extremely difficult** to find which attribute represents which feature on the map. Not having the capability to visualize shapefiles and find out what features they contain is a huge drawback.

### Importing downloaded data into TileMill

It's time to create a new project and import the data we downloaded earlier from the Metro Extracts website into TileMill:

1. Fire up TileMill. You will be greeted with the **Projects** screen, where you can choose one of the templates that come with TileMill, a previous project, or create a new project.

2. Click on the **New project** button, and we will start from scratch.

3. In the **New project** dialog box, type any filename you want. The **Name** and **Description** fields are optional. Ensure that the **Image Format** is PNG (24-bit).

4. Leave the **Default data** checkbox selected. This will import a world layer into the project automatically. We don't need it, but it will help us to navigate, and we can always get rid of it later.
5. To close the dialog box and create our new project, click on the **Add** button.

6. Great! We are now into TileMill. You will see the world layer that we imported earlier when creating the project. There is one style currently active—*style.mss*—which is the one that gives the current map the blue background color representing the sea. The countries are styled with *line-color* specified as a darker blue and *line-width* of 2, and they are filled with a pure white using *polygon-fill*:

![TileMill style example](image)

7. Open **Layers**. You will see the *#countries* layer that was imported by TileMill automatically. Leave it for now, and click on the **Add layer** button to add a new layer:

![Layers list example](image)
8. In the **Add layer** dialog box, select **Browse** and browse to the location of the extracted ZIP files. We will import the polygons first. My file is called `san-francisco.osm.polygon.shp`; yours may be different depending on the area that you download, but the filename will always end with `polygons.shp`.

9. Click on **Done** to browse the dialog box to return to **Add layer**. The next thing we will need to do is specify an ID. TileMill set mine as `sanfrancisco` automatically, but as I will import more layers, this ID won’t make sense, and it’s better to select something more appropriate. It will help us separate it from the lines and layers containing other datatypes. I will set my ID to **sf-polygons** for the San Francisco polygons. Finally, click on **Save**:

![Add layer dialog box](image)

In most cases, you won’t need to adjust the **SRS** field. However, in case it fails, we have to provide the correct value in this field.

**Spatial Referencing System (SRS)** is a quick way to identify a spatial referencing system using the PROJ.4 syntax. There are several ways to get the PROJ.4 string, but one of the easiest is by searching on [http://www.spatialreference.org/](http://www.spatialreference.org/).

You can find the PROJ.4 string for WGS 84 at [http://www.spatialreference.org/ref/epsg/4326/proj4/](http://www.spatialreference.org/ref/epsg/4326/proj4/), which is the following:

```
+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs
```

For more information regarding the PROJ.4 specification, visit [http://trac.osgeo.org/proj/](http://trac.osgeo.org/proj/).
10. You will return to TileMill. Note that \#sf-polygons is now in the Layers list. Click on the magnifier icon next to the \#sf-polygons layer. The map will zoom in to the region that the data is located on.

11. Repeat the procedure and import the other two layers san-francisco.osm-line.shp and san-francisco.shp from the coastlines ZIP file. You can set up the IDs as \#sf-lines and \#coast, respectively.

12. The layer order is very important in TileMill, and if you import the layers in the order described previously, you will have the \#sf-polygons layer at the bottom, \#sf-lines in the middle, and \#coast at the top. The layer order specifies the order that TileMill draws the layers in, and by this we mean that data may be covered by the layers above it. In this case, our lines, which include the roads, will be covered by the coastlines’ polygons.

Notice that the icon next to each layer shows what type of data it contains. It may show a line with two dots for line data and a filled polygon shape for polygon data.

You can drag this icon to change the layer order. Try it now and set the layer order with \#coast at the bottom, \#sf-lines in the middle, and \#sf-polygons at the top.

**Styling data using CartoCSS**

You need to perform the following steps:

1. First of all, let's set up our color palette. Clear the style.mss stylesheet of all code and copy the following variables:

   @water: #2f4145;
   @water-outline: #384C51;
   @land: #322f2e;
   @streets: #D7AE26;
   @streets-dark: #AC8812;
   @buildings: #6D6765;
   @text: #392C00;

   Well, you are looking at hex values, and it is kind of difficult to decipher the colors just by reading them.

   Luckily, TileMill displays the colors used in CartoCSS in the color swatches at the bottom of the screen. Clicking on a color swatch allows you to change this color using a handy color picker.
This is not the best way to build a color palette. Now is the perfect time to talk about the tools that can help us build a really beautiful color scheme. We want to design the most beautiful maps, after all!

One of the online tools we can use is Adobe Color (formerly, Adobe Kuler). You can access it at https://color.adobe.com/, and it will help you build your color scheme by selecting a color rule, such as monochromatic, complimentary, triadic, compound, or even building a custom color palette. We can save our color scheme and share it with the Adobe Color community, as well as explore the color schemes designed by other people.

Another popular option is Paletton (http://paletton.com). It offers almost the same features as Adobe Color.

There is also Color Sphere! from MadCube, which is a Chrome extension, and there are even native applications such as ColorSchemer Studio 2 (http://www.colorschemer.com/).
2. Now, let's style the background and `#coast`, which is the coastlines shapefiles layer imported earlier. It's a polygon layer, and we will use `polygon-fill` and `line-color` to create a thin outline:

```javascript
Map {
  background-color: @water;
}

#coast {
  polygon-fill: @land;
  line-color: @water-outline;
}
```

3. In order for TileMill to update the map, you will have to save, so ensure that you do this by clicking on the Save button on the upper right-hand side of TileMill:

![TileMill and Mapbox Studio](image)

The keyboard shortcut to save is Ctrl + S for Windows and Cmd + S for OS X.

At this point, we should be able to see the styled coastline. If nothing is visible, open Layers and click on the magnifier icon next to the coast. It will zoom to the region that the coastline is in.
Suppose you have an error in CartoCSS. In this case, TileMill will highlight the line number in yellow and display an error message at the bottom of the screen. Ensure that you fix all the errors and save again:

4. The next time we open the project, TileMill will not zoom automatically to the region we are working with. It's a good time to do it now as we have styled the basic coastline. Click on the Wrench icon in the upper right-hand side of the screen to go to Project Settings.

5. You will probably see a blank dark blue map. Our region is there, but we are currently at zoom level 1, and the whole world is visible. Using the zoom plus and minus buttons and dragging around the map, try to find the region that you are working with.

6. When you zoom in to the region, shift and drag to enclose it in a rectangle. This rectangle will define the area that we are working with and the one that TileMill will eventually export the tiles of.

7. Click on the center of the region, and a marker will appear. This is the center of our project.
8. When you have finished, click on **Save**: 

![Image of TileMill and Mapbox Studio interface](image)

9. Back in the editor, let's style the roads. We will set the colors of all roads to yellow and then use different line widths for different types of roads by executing the following code:

```javascript
// Set the colors of all roads to @street
#sf-lines[highway != ""]{
  line-color: @streets;
}

// for motorways, and trunk set line width to 5
#sf-lines[highway="motorway"],
#sf-lines[highway="trunk"] {
  line-width: 5;
}
```
// for primary roads, set line width to 3.
#sf-lines[highway="primary"] {
    line-width: 3;
}

// for secondary roads, set line width to 2
#sf-lines[highway="secondary"] {
    line-width: 2;
}

// for paths roads, set line width to 1, and use dashes
#sf-lines[highway="path"] {
    line-width: 1;
    line-dasharray: 5, 3;
    line-cap: round;
    line-join: round;
}

This is shown in the following screenshot:
10. To style the buildings, we will fill them with the color that we defined in the variable and set the opacity to 0.35. This will mix the color of the buildings with the layer behind—in this case, the coastline. Run the following code:

```css
#sf-polygons[building != ""] {
  polygon-fill: @buildings;
  polygon-opacity: 0.35;
}
```

In case you forget what features a layer contains, you can go to the **Layers** palette and click on the icon that looks like a spreadsheet next to the layer's name. TileMill displays a table with all the features of a layer and the values that they contain:

11. If you try to zoom out now, you will notice that the streets are very dense with a lot of detail at lower zoom levels. We will fix this next. Reduce the opacity of all roads to 0.45 when the zoom level is less than 16 through the following code:

```css
[zoom < 16] {
  #sf-lines[highway != ""] {
    line-opacity: 0.45;
  }
}
```
12. Reduce the opacity of all roads to 0.25 when the zoom level is less than 14 with the following script:

```javascript
[zoom < 14] {
  #sf-lines[highway != ""] {
    line-opacity: 0.25;
  }
}
```

13. Use this code to hide the roads when the zoom level is less than 12:

```javascript
[zoom < 12] {
  #sf-lines[highway != ""] {
    line-opacity: 0;
  }
}
```

14. When the zoom level is less than 16, set the line opacity to 1 by executing the following code:

```javascript
[zoom < 16] {
  #sf-lines[highway="motorway"],
  #sf-lines[highway="trunk"],
  #sf-lines[highway="primary"],
  #sf-lines[highway="secondary"],{
    line-opacity: 1;
  }
}
```

Remember that styles at the bottom overwrite the ones above them. So, even if we set the opacity to 0.45 for zoom < 16 for all roads, motorways, trunks, and primary and secondary roads, they will still have an opacity of 1. The rest will remain at the previous value.

15. When the zoom level is greater than 15, you can use the following code:

```javascript
[zoom > 15] {
  #sf-lines[highway="primary"]{
    line-width: 10;
  }

  #sf-lines[highway = "secondary"] {
    line-width: 8;
  }
```
Great! Try to zoom in and out now and you will notice the difference. Observe how the opacity of the roads reduces every time you zoom out and how certain features are hidden.

16. Our map already looks great, but it doesn’t have labels yet. A map without road names will not be very popular! At the beginning of our style, directly below the #coast styling, we have a section that sets all road colors to street colors. Execute the following code:

```markdown
#sf-lines[highway != ""]{
line-color: @streets;
}
```

17. Let’s add the label style in this selector. Replace CartoCSS in the preceding section with the following code. As we explained in the introduction, we forced TileMill to redraw the elements in the current selector with double colons :: (road-label is just a name we defined to remember what this redrawn part does). We will fetch road names from the [name] column. The rest of CartoCSS just consists of various typography attributes. We will also set text-placement to line, so the labels will be aligned with the lines of the roads:

```markdown
#sf-lines[highway != ""]{
line-color: @streets;

::road-label[zoom>15] {
text-name: [name];
text-size: 14;
text-fill: @text;
text-face-name: "Arial Bold";
text-halo-radius: 1;
text-halo-fill: @streets-dark;
text-placement: line;
}
```
This is how it will look:

**Styling a map with Mapbox Studio**

In this recipe, we will style a map using Mapbox Studio. The biggest difference from TileMill is the workflow, while other parts, such as CartoCSS, remain the same.

**How it works...**

If you haven't installed Mapbox Studio yet, head over to the Mapbox website and download it now from https://www.mapbox.com/mapbox-studio/.

It's available for all major platforms, and the installation is pretty straightforward. Once installed, open the application and prepare for an interface tour!
When you first start Mapbox Studio, you will be greeted with the Projects screen, just as in TileMill.

You can start either by selecting a starter style (a template), or by creating a new style by selecting a source from the ones provided by Mapbox. The sources provided by Mapbox are Mapbox Streets, Mapbox Terrain, and Mapbox Satellite.

In case you want to overlay or import external data from other sources, you can create a new source by clicking on the Blank Source button on the right-hand side.

Once you have selected a style, you will be transferred to the style editor. The interface is separated into two parts, with the map on the left-hand side and the CartoCSS editor on the right-hand side, just as in TileMill:
On the left-hand side, you will find the **Save as**, **Settings**, **Layers**, **Fonts**, and **Docs** buttons:

- **Save as**: This saves the project. You have to save each new project you create to your local hard drive.
- **Settings**: This allows you to configure the project settings, such as the available zoom levels and other parameters.
- **Layers**: This shows all the layers available at the source(s) that you select when you create the project.
- **Fonts**: This opens a window with a preview and the font names of the typefaces available for use in your maps.
- **Docs**: This opens a window with reference both to the user interface of Mapbox Studio and CartoCSS.

At the top of the window, there is a toolbar with some buttons. Let's check them out from left to right:

- **Search**: This allows you to search for a specific location on the map. Try a city or country to quickly jump to that location.
- **Full screen map**: This shows or hides the CartoCSS editor, showing the map in full screen.
- **+ and – buttons**: This zooms in or out of the map.
- **Inspector**: This allows you to visualize all layer data contained in the sources.

In the far upper-right corner of the CartoCSS window, you will find the Export Image button. It allows you to export images up to 600 ppi.

### How to do it...

Perform the following steps:

1. Create a new project in Mapbox Studio by clicking on **Projects** and then on **+ New project**.
2. In the **New project** window, select the sources to be used in the map. The source could be one of the Mapbox ones, such as Mapbox Streets, Mapbox Terrain, or Mapbox Satellite, or a combination of them.
3. Style the map using CartoCSS.
Styling a map
Perform the following steps:

1. Create a new project in Mapbox Studio. We will not use a template; we will start from scratch. At the bottom of the New projects screen, you will see a Create button; next to it, ensure that it's mapbox.mapbox-streets-v5, which means that Mapbox Streets will be used as source.

Remember, we do not import data into Mapbox Studio as we do with TileMill; instead, we will use the standard Mapbox sources provided to us. We can create our own sources if we want, and we can combine multiple sources together to create more complex maps.

For now, we will use just Mapbox Streets, so click on the Create button:

2. The editor will pop up with some basic imported CartoCSS. The first thing you will notice is that the Save as button in the upper-right corner is highlighted in blue. This means that the project is unsaved. We won't get far without saving, so let's do it right now.
Click on the **Save as** button, select a location on your hard drive, name the project as you want, and click on **Save**.

3. Zoom in to a region of your choice and clean up all the CartoCSS from the style. Hit **Save** again (Ctrl + S for Windows and Cmd + S for Mac). You will notice that the map is now blank. Don't worry, there is an awesome way to visualize data in Mapbox Studio.

4. Let's finish with the project setup first before we do this. Head over to **Settings** and increase **Maxzoom** to **22**.

5. Among the icons arranged on the topmost bar above the map is the Inspector icon. It looks like layers with a magnifier on top. Click on it now.

The map now shows the data that the layer contains. If your zoom range is below 15, the data may be dense, so ensure that you zoom in until you get a clear picture of what's going on.

If you click on various lines, you will see that a popup appears, showing you information such as the kind of layer, class, type, and name:
Let's define some colors. At the beginning of the style, add the following variables:

\[
\begin{align*}
@water: & \#046380; \\
@some: & \#002F2F; \\
@land: & \#E6E2AF; \\
@outlines: & \#EFECCA; \\
@text: & \#A7A37E; \\
@green: & \#002F2B;
\end{align*}
\]

Open Layers from the Mapbox Studio sidebar. You will see all the layer IDs available in Mapbox Streets as we are using this source. Click on a layer ID to see what classes it has; for example, click on \#road, and you will see that it can be one of motorway, motorway link, main, street, and so on.

We will begin by styling \#water. First of all, set the background-color value, which in this case actually represents the ground color, then style \#water. Run the following code:

```javascript
Map {
   background-color: @land;
}

#water {
   polygon-fill: @water;
   line-width: 1;
   line-color: @outlines;
}
```

Click on Save when you have finished and inspect the map:
10. Continue inspecting the layers and styling the IDs that you need one-by-one in the same way that we did in TileMill. Here is the code for the complete style:

```css
@water: #046380;
@some: #002F2F;
@land: #E6E2AF;
@outlines: #EFECCA;
@text: #A7A37E;
@green: #002F2B;

Map {
    background-color: @land;
}

#water {
    polygon-fill: @water;
    line-width: 1;
    line-color: @outlines;
}

\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\n```
TileMill and Mapbox Studio

[type='street'], [type='street_limited'], [type='driveway'], [type='residential'] {
  line-width: 4;
  [zoom < 14] { line-width: 1 }
}

[type='unclassified'], [type='secondary'] {
  line-width: 4;
  [zoom < 14] { line-opacity: 0 }
}

[type="path"] {
  line-width: 1;
  line-dasharray: 5, 3;
  line-cap: round;
  line-join: round;
  [zoom < 16] { line-opacity: 0 }
}

#road_label {
  text-name: [name];
  text-size: 18;
  text-fill: @text;
  text-face-name: "Meta Offc Pro Bold";
  text-halo-radius: 2;
  text-halo-fill: @land;
  text-placement: line;
}

////////////////////////////////////////////////////////////////////
/// BUILDINGS ///
////////////////////////////////////////////////////////////////////

#building {
  polygon-fill: @green;
  [zoom < 13] {
    polygon-opacity: 0.0;
  }
}


Here's the final result:

There's more...

In the previous section, you learned how you can use and style data provided by Mapbox.

Now, we will continue from there and take a look at how to import our own data into Mapbox Studio.
Perform the following steps:

1. Open Mapbox Studio. Click on Styles & Sources in the lower-left corner. We will need to create our own source. Click on + New Style or Source at the top of the sidebar that opens up. On the left-hand side of the New Style or Source window, there is a Source section. Click on the Blank source button:

2. Now, we need to add our layers. The data that a layer can contain can be a shapefile; can be in the .kml, .geojson, .gpx, .csv, .tif, and .vrt formats; or can even be PostGIS and SQLite databases. We will use a shapefile that contains the national parks in US. You can download the file from http://www.naturalearthdata.com/downloads/10m-cultural-vectors/parks-and-protected-lands/. It’s a ZIP file, so we will need to extract it somewhere on the hard drive.

3. Back in Mapbox Studio, click on the Browse button in the Add a New Datasource window, and select ne_10m_parks_and_protected_lands_area.shp from the files that we extracted in the previous step.
You will see that the shapefile has been imported correctly. Mapbox Studio now shows us a preview, and we can already zoom and pan. You can even click on a shape and a popup with the layer attributes will appear:

On the sidebar to the right, note that it correctly detected the projection and showed it in PROJ.4, which is as follows:

```
+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs
```

You can add as many layers as you want in a single source and even a mix of shapes, GeoJSON, and other file types.

4. You can rename the layer if you want. Click on the Rename button below the filename at the top of the sidebar to the right. I named mine national_parks for simplicity.

5. Before uploading our own data to Mapbox, we need to save the project locally. Press Ctrl + S on Windows or Cmd + S on OS X, choose a filename, and store it somewhere on your hard drive.

6. Now that the project is saved, click on Settings and then on Upload to Mapbox. Mapbox Studio will initiate the process to upload the data to your Mapbox account.
7. Once the upload is completed, you can find the Map ID above the **Upload to Mapbox** button. Copy the Map ID; we will need it in a while.

8. That's all that's needed to create a new data source to be used in Mapbox Studio. Now, we need to open the map from the previous recipe. If you didn't complete the previous recipe, any of the sample projects will do just fine.

9. Once the project is open, click on **Layers** and then on **Change Source**. In the **Sources** window, before the **Apply** button is the source that we will use for this map, which is `mapbox.mapbox-streets-v5`.

10. We will need to append our own source, which we just created. We have the Map ID already copied to the clipboard, so add a comma (,) and paste the Map ID. The complete string should look similar to the following:

    `mapbox.mapbox-streets-v5, mapboxrecipes.62340c1c`

11. Click on **Apply**. Our own custom datasource will be added to the project.

    You can make sure of this by opening the **Layers** palette. Note that there is a `#national_parks` layer here. This is our own data source—the layer that we added earlier.
12. In order to see the layer, we need to style it first. For this layer, paste the following CartoCSS code at the bottom of the style:

```css
#national_parks {
    polygon-fill: @green;
    opacity: 0.5;
    text-name: [name];
    text-size: 32;
    text-fill: @text;
    text-face-name: "Meta Offc Pro Bold";
    text-halo-radius: 2;
    text-avoid-edges: true;
    text-halo-fill: @green;
    [zoom < 7] {
        text-size: 18;
    }
}
```

13. Save the project and enjoy our custom data overlaid on the Mapbox Street map data:
Publishing your base map on your server with PHP

So far, we have successfully used the tools provided by Mapbox to style our maps. We have also seen how easy it is to publish a map to the Mapbox platform, especially if we use Mapbox Studio.

There are cases where we may need to host a map on our own server, and we will explore two different ways of doing this.

In the first case, we will use PHP. While this is not the best option for this task, PHP can be found installed on every server, and projects using PHP are easy to deploy.

How it works...

First of all, we will export our maps from TileMill. The format that we will export in will be .mbtiles, which is actually a SQLite file containing the data and the tiles.

MBTiles is just a SQLite database that contains metadata information along with PNG images. Its purpose is to have a single file that contains everything needed to serve a map. In cases such as mobile applications, storing thousands of tiles and metadata information in the app container can be difficult to manage and can be extremely poor in performance.

The following is a screenshot showing the tables of the SQLite database along with the table of PNG files:
Reading SQLite files directly in PHP is not performant at all, so we will use MBUtil to convert them into PNG files. Next, we will use a PHP tile server developed by Petr Pridal. Developing our own tile server is not a hard task, but it is beyond the scope of this book. Finally, we will set up a simple project and serve the tiles.

**How to do it...**

Perform the following steps:

1. Export MBTiles from TileMill by clicking on Export and choosing MBTiles as the export option.
2. In Export Options, ensure that the zoom is configured and the center is within the bounds of our map.
3. Click on Export, and once finished, click on Save to store MBTiles on the hard drive.
4. Convert the MBTiles (SQLite) files into regular PNG files using MBUtil.
5. Use one of the open source tile servers, such as the PHP tile server, to serve the newly generated PNG files.

**Exporting a map from TileMill**

The steps are as follows:

1. Fire up TileMill and open the project you want to export. In the upper-right corner, you will find an Export combo box. Select it and choose MBTiles.
2. Set up the region that you want to export by defending the bounding box in the map. Also, ensure that the center is a location within the region.
3. Configure the zoom range to export. You will notice that as you increase the zoom, the time and hard drive space required to export also dramatically increases:

4. Click on Export to export your tiles. You will then be transferred to the export queue, and you will be able to monitor the progress and the time remaining:

5. Once finished, click on the Save button. TileMill will show you the location on your hard disk to find the exported MBTiles.
Converting an MBTiles database into regular PNG files

We will convert the MBTiles SQLite database into regular PNG files. The files have to follow a specific folder structure and a naming schema for us to be able to use them in the tile servers.

For the task, we will need an external utility called MBUtil. It's a powerful tool that allows us to specify our own naming scheme and even change the output format. In this case, the default XYZ naming scheme will do just fine, and the default format is PNG, which is exactly what we need.

Perform the following steps:

1. Let's download MBUtil. Head over to the repo and follow the instructions to set up MBUtil globally at https://github.com/mapbox/mbutil.

2. Once installed, convert your files using the `mb-util <inputfilename> <outputfolder>` command. A new directory will be created in the location you specified containing the tiles in an XYZ format.

If you explore the folder exported by MBUtil, you will discover the file structure in the XYZ format:

It uses a folder for each zoom level, with each column being a subdirectory and each tile in the column a file. The other common format used in maps is TMS. You can read more about the XYZ format at http://wiki.openstreetmap.org/wiki/Slippy_map_tilenames.
Creating a tile server PHP project

We will download the PHP tile server developed by Petr Pridal, and we will set up a simple project that serves the tiles we exported in the previous recipe. Perform the following steps:

1. Create a new folder to host the project and get into it using the following command:
   
   ```bash
   mkdir MapBoxPHPServer && cd MapBoxPHPServer
   ```

2. We will use `tileserver-php`. It's extremely simple to use, and the whole project is just a single PHP file. Clone the project into the folder using the following command:
   
   ```bash
   git clone https://github.com/klokantech/tileserver-php
   ```

3. Now, copy the exported folder from MBUtil into the same directory as the `tileserver.php` file.

4. You can now access the `tileserver.php` file from a web browser at `http://localhost/tileserver.php`:

![TileServer](image)

That's it! We can now serve our maps from our own PHP server. Deploying the server is extremely easy, and the only thing needed is to copy the files into the server. The best part is that PHP is installed almost everywhere, and it is also very easy to find a cheap hosting service.
Publishing your base map on your server with Node.js

As we explained in the previous chapter, PHP is not the ideal technology to use to serve our tiles. This is not because PHP is a bad language or anything similar; it's because there are other technologies that excel in multithreading operations, which is exactly what's needed when serving tiles at an enormous speed.

In this recipe, we will use TileStream, which is a high-performance tile server powered by MBTiles. TileStream is maintained by Mapbox and uses Node.js and SQLite as the backend technologies.

In the preceding PHP recipe, we converted the tiles to the XYZ format to squeeze out any performance benefits while serving the tiles. Node.js doesn't have any performance issues while reading the tiles directly from the MBTiles container, which is just a SQLite database, so we don't need to convert the tiles in this case.

We can use the MBTiles we exported in the previous recipe. Then we will download TileStream from GitHub, which is the tile server that we will use in this case.

Getting ready

TileStream requires Node.js v0.10.x or 0.8.x. It will not work with the most recent versions of Node.js, so ensure that you have one of the preceding versions installed.

If you don't have Node.js installed yet, go ahead and install it. Ensure that you do not install the latest build, but one of the versions required by TileStream.

How to do it...

Perform the following steps:

1. Export the MBTiles file from TileMill by clicking on Export and selecting MBTiles as the export option.
2. In Export Options, ensure that the zoom is configured and the center is within the bounds of our map.
3. Click on Export, and once finished, click on Save to store the MBTiles file on the hard drive.
4. Use any one of the open source Node.js tile servers, such as TileStream, to directly serve MBTiles.
Creating a tile server Node.js project

Perform the following steps:

1. Ensure that you have the correct version of Node.js installed. Open a terminal and type `node --version`, and Node.js will print out the version.

2. Start by creating a new folder in your hard drive. I named mine `MapBoxRecipes-NodeServer`.

3. Get into the folder and clone the TileStream repo using the following command:
   ```
   git clone https://github.com/mapbox/tilestream.git .
   ```
   The dot (.) at the end clones it into the folder that we are in without creating a new subfolder.

4. Once installed, ensure that the server is working. Start the TileStream server by typing `./index.js`.

5. Let's copy the `.mbtiles` file into a location that TileStream expects them to be in. The default directory is `~/documents/MapBox/tiles`.

6. Head over to a browser and type `http://localhost:8888/` to access TileStream. You can access the map of your choice directly using `http://localhost:8888/#!/map/SanFrancisco`:

   ![MapStream image]

That's it! We now have our custom MBTiles file exported from TileMill and served by our own Node.js server. You can use a hosting service that supports Node.js, such as Heroku, to deploy your project.
Where to buy this book

You can buy Mapbox Cookbook from the [Packt Publishing website](https://www.packtpub.com). Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

[Click here](https://www.packtpub.com) for ordering and shipping details.