
First Graphics App Documentation

IRE/NICAR

Mar 07, 2020

Contents

1	What you will make	3
2	About the authors	5
3	Prelude: Prerequisites	7
3.1	Command-line interface	7
3.2	Text editor	7
3.3	Node.js	8
3.4	npm	8
3.5	Git and GitHub	8
4	Chapter 1: Hello Git	9
5	Chapter 2: Hello framework	13
6	Chapter 3: Hello template	17
7	Chapter 4: Hello data	25
8	Chapter 5: Hello cards	31
9	Chapter 6: Hello charts	41
10	Chapter 7: Hello map	65
11	Chapter 8: Hello Internet	75

A step-by-step guide to publishing a standalone story from a dataset.

This tutorial will show you how journalists at America's top news organizations escape rigid content-management systems to publish custom interactive graphics on deadline. You will get hands-on experience in every stage of the development process, writing JavaScript, HTML and CSS within a Node.js framework. You won't stop until you've deployed a working application on the World Wide Web.

CHAPTER 1

What you will make

By the end of this lesson, you will publish a standalone page with a series of graphics examining the high homicide rate in Harvard Park, a small neighborhood in South Los Angeles. You will do so by repurposing data from [a 2017 Los Angeles Times story](#) by Nicole Santa Cruz and Cindy Chang.

A working example of what you will make can be found at ireapps.github.io/first-graphics-app

CHAPTER 2

About the authors

This guide was prepared for training sessions of Investigative Reporters and Editors (IRE) and the National Institute for Computer-Assisted Reporting (NICAR) by Dana Amihire, Armand Emamdjomeh and Ben Welsh. It debuted in March 2018 at NICAR's conference in Chicago. It returned for a second run at the 2019 edition of the conference in Newport Beach, Calif. A third session is planned at the 2020 conference in New Orleans.

The course's development was inspired by the footloose spirit of funk music. We urge you to bust free of the computer systems that constrain your creativity. Hit play and get into the groove.

CHAPTER 3

Prelude: Prerequisites

Before you can begin, your computer needs the following tools installed and working:

1. A [command-line interface](#) to interact with your computer
2. A [text editor](#) to work with plain text files
3. Version 8.9.4 or greater of the [Node.js](#) JavaScript runtime
4. The [npm](#) package manager
5. [Git](#) version control software and an account at [GitHub.com](#)

Warning: Stop and make sure you have all these tools installed and working properly. Otherwise, [you're gonna have a bad time](#).

3.1 Command-line interface

Unless something is wrong with your computer, there should be a way to open a special window that lets you type in commands. Different operating systems give this tool slightly different names, but they all have some form of it. The generic term for it is the “command-line interface.”

On Windows you can find it by opening the “command prompt.” Here are [instructions](#). On Apple computers, you open the “Terminal” application. Ubuntu Linux comes with a program of the same name.

3.2 Text editor

A program like Microsoft Word, which can do all sorts of text formatting, like change the size and color of words, is not what you need. Do not try to use it.

You need a program that works with simple “plain text” files, and is therefore capable of editing documents containing Python code, HTML markup and other languages without dressing them up. Such programs are easy to find and some of the best ones are free, including those below.

Regardless of your operating system, we recommend installing [Visual Studio Code](#). [Atom](#) and [Sublime Text](#) are also excellent options.

3.3 Node.js

Node.js is an open-source programming framework built using JavaScript. Many programmers like it because it allows them to write JavaScript not just in their browser for “front-end” tasks, but also in the terminal or on a server for “back-end” tasks.

We recommend you use the latest “long-term support” version, which at the time of this writing was 12.16.1. The [Node.js site](#) has [installer packages](#) available for Windows and Mac OSX.

You can verify if you have Node installed, and if so what version, by typing the following into your terminal:

```
$ node --version
```

The number you get back is the version you have installed. If you get an error, you don’t have Node.js installed and you should start from scratch with an installer package. If you have a slightly older version, you are probably okay. But we make no guarantees. Consider upgrading.

3.4 npm

Installing Node will also install npm on your computer, which stands for “Node Package Manager.” During the class, we will use it to install open-source JavaScript packages that will help us draw charts and maps.

You can verify you have npm installed by running the following command on your terminal.

```
$ npm --version
```

3.5 Git and GitHub

[Git](#) is a version control program for saving the changes you make to files over time. This is useful when you’re working on your own, but quickly becomes essential with large software projects when you work with other developers.

[GitHub](#) is a website that hosts git code repositories, both public and private. It comes with many helpful tools for reviewing code and managing projects. It also has some [extra tricks](#) that make it easy to publish web pages, which we will use later. GitHub offers helpful guides for installing Git for [Windows](#), [Macs](#) and [Linux](#).

You can verify Git is installed from your command line like so:

```
$ git --version
```

Once that’s done, you should create an account at GitHub, if you don’t already have one. [The free plan](#) is all that’s required to complete this lesson. If you make a new account, make sure to confirm your email address with GitHub. We’ll need that for something later.

CHAPTER 4

Chapter 1: Hello Git

First things first. It always helps to store all your code in the same place, instead of haphazard folders around your computer. This way, you always know where to look if you need to find a project.

In this case, let's call that directory `Code`.

```
# You don't have to type the "$" It's just a generic symbol
# geeks use to show they're working on the command line.
$ mkdir Code
```

You can use the `cd` command to “change directory” into the directory we created.

```
# You don't have to type the "$" It's just a generic symbol
# geeks use to show they're working on the command line.
$ cd Code
```

Then, create a new directory where we can store the code for the project we're going to build today. Name it after our application.

```
$ mkdir first-graphics-app
```

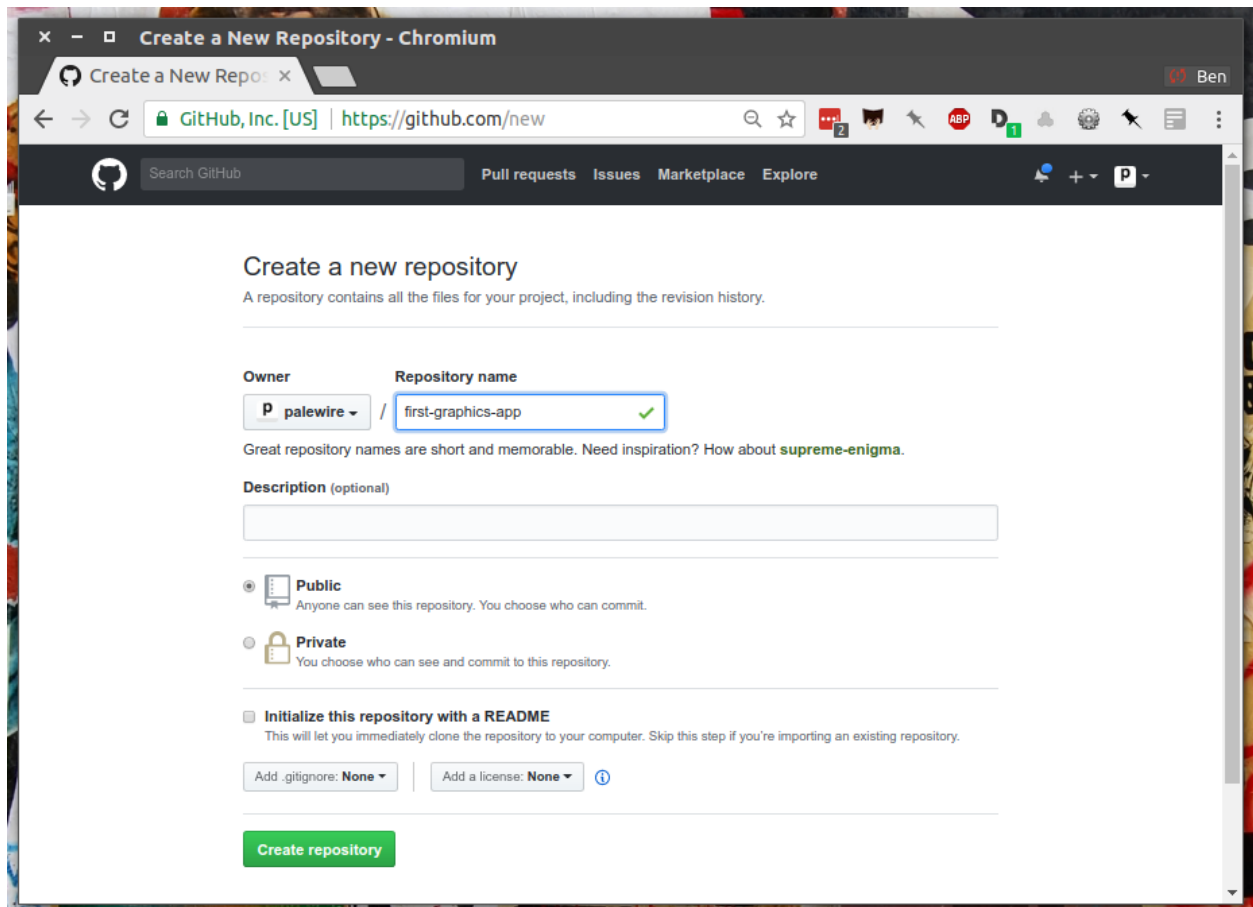
Now, use `cd` again to enter the the directory you just created.

```
$ cd first-graphics-app
```

Use the command `git init` to create a new Git repository in the current directory. This will be the root of our version-controlled project.

```
# "." is a common shortcut to refer to the current directory from the terminal
$ git init .
```

Visit [GitHub](#) and [create](#) a new public repository named `first-graphics-app`. Don't check “Initialize with README.” You'll want to start with a blank repository.



Then connect your local directory to GitHub with the following command. Replace `<yourusername>` with your GitHub user name.

```
$ git remote add origin https://github.com/<yourusername>/first-graphics-app.git
```

Create your first file, a blank README with a [Markdown](#) file extension since that's the preferred format of GitHub. The filename will be `README.md`. Markdown is a simple way of writing nicely formatted text, complete with headlines, links and images.

```
# Macs or Linux:
$ touch README.md

# If you're using Visual Studio Code, fire it up in your text editor right away:
$ code README.md
```

Open up the README in your text editor and type something in it. Maybe something like:

Make sure to save it. Then officially add the file to your repository for tracking with Git's `add` command.

```
$ git add README.md
```

Log its creation with Git's `commit` command. You can include a personalized message after the `-m` flag. If you're on a Windows machine, make sure you use double quotes around your commit message.

```
$ git commit -m "First commit"
```

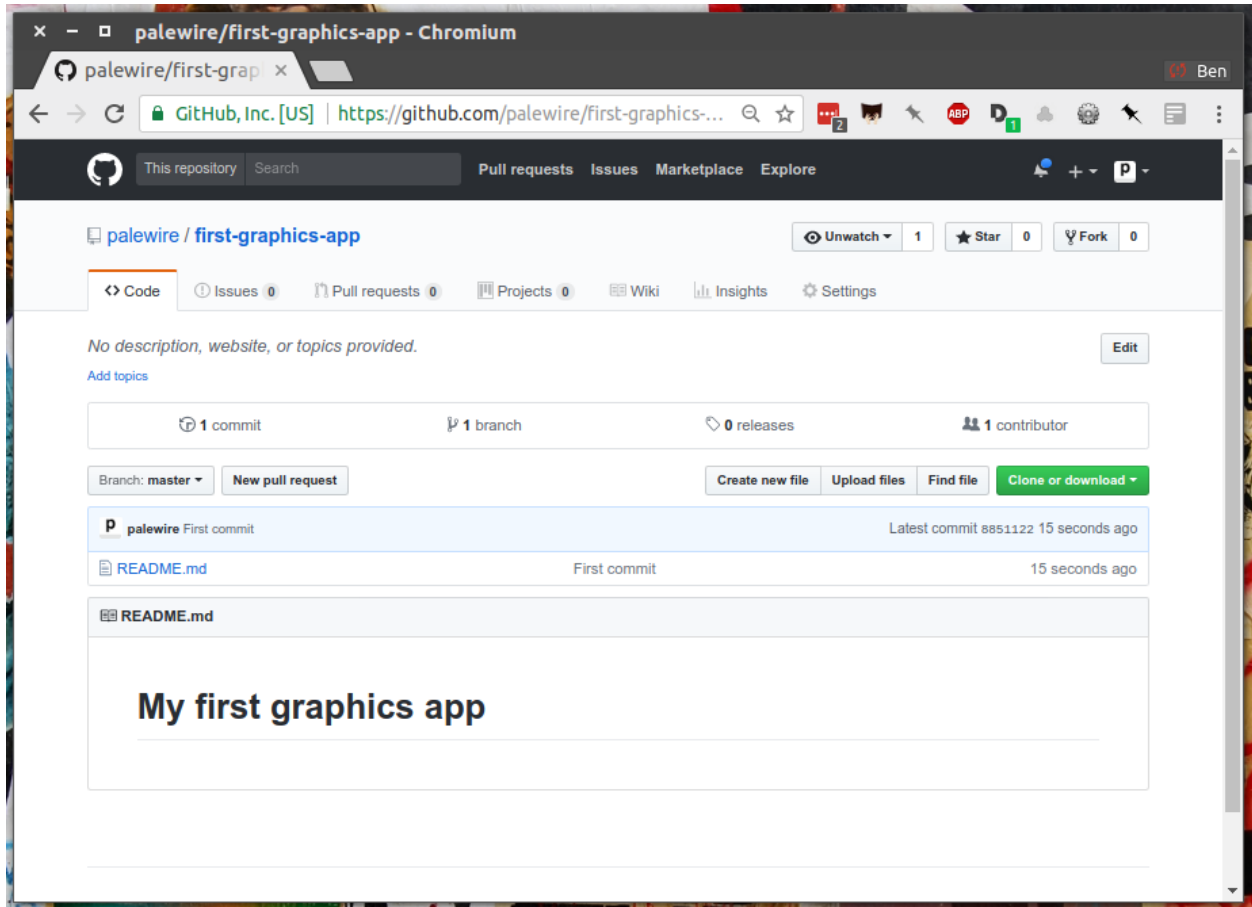
If this is your first time using Git, you may be prompted to configure your name and email. If so, take the time now. Then run the `commit` command above again.

```
$ git config --global user.email "your@email.com"
$ git config --global user.name "your name"
```

Now, finally, push your commit up to GitHub.

```
$ git push origin master
```

You just created your first code commit! Reload your repository on GitHub and see your handiwork.



Chapter 2: Hello framework

Now that we have our Git repository created, we're going to start installing the tools we need to do our job.

The first and more important is a [framework](#). What's that? Nothing more than fancy name for a set of software tools that, working together, can stand up a website. Believe it or not, it takes dozens of different things to pull a good site together. Frameworks aim to make the challenge easier by organizing a curated set of tools into a system that saves programmers time.

There are a lot of different frameworks out there. Maybe you've heard of some them, like [Django](#) for Python or [Rails](#) for Ruby.

Note: While some frameworks are more popular than others, each newsroom tends to go its own way with a custom system for publishing pages. The programming languages and the details vary, but the fundamentals are almost all the same. Some of them have even been released as open-source software. They include:

- The Los Angeles Times Data Desk's [bigbuild](#)
 - The Dallas Morning News' [generator-dmninteractives](#)
 - The Seattle Times' [newsapp-template](#)
 - The NPR Apps team's [dailygraphics](#)
 - Politico's [generator-politico-graphics](#)
-

Node.js is so fancy it has more than plain old frameworks. It even includes a framework for creating frameworks! It's called [Yeoman](#). Its "generator" system makes it easier for publishers to tailor a framework to their site without having to reinvent all the wheels themselves.

We'll start by installing Yeoman using the Node Package Manager ([npm](#)), which can visit the Internet to download and install any of the thousands of open-source Node.js packages listed in its directory.

```
$ npm install -g yo@3.1.1
```

The `-g` means that we're installing the packages globally. You'll be able to run these from any directory on your computer.

The `@` followed by numbers after the `yo` package means we're installing a specific version. Code libraries often change quickly. By specifying a version, we're protecting ourselves against future changes that could break the code of this lesson. If you don't care what version you're installing, you could just use the name of the package, i.e, `npm install -g yo`.

Next we'll install **Gulp**, a helpful Node.js utility for running a framework on your computer as you develop a site. Again, we turn to npm.

```
$ npm install -g gulp@4.0.2
```

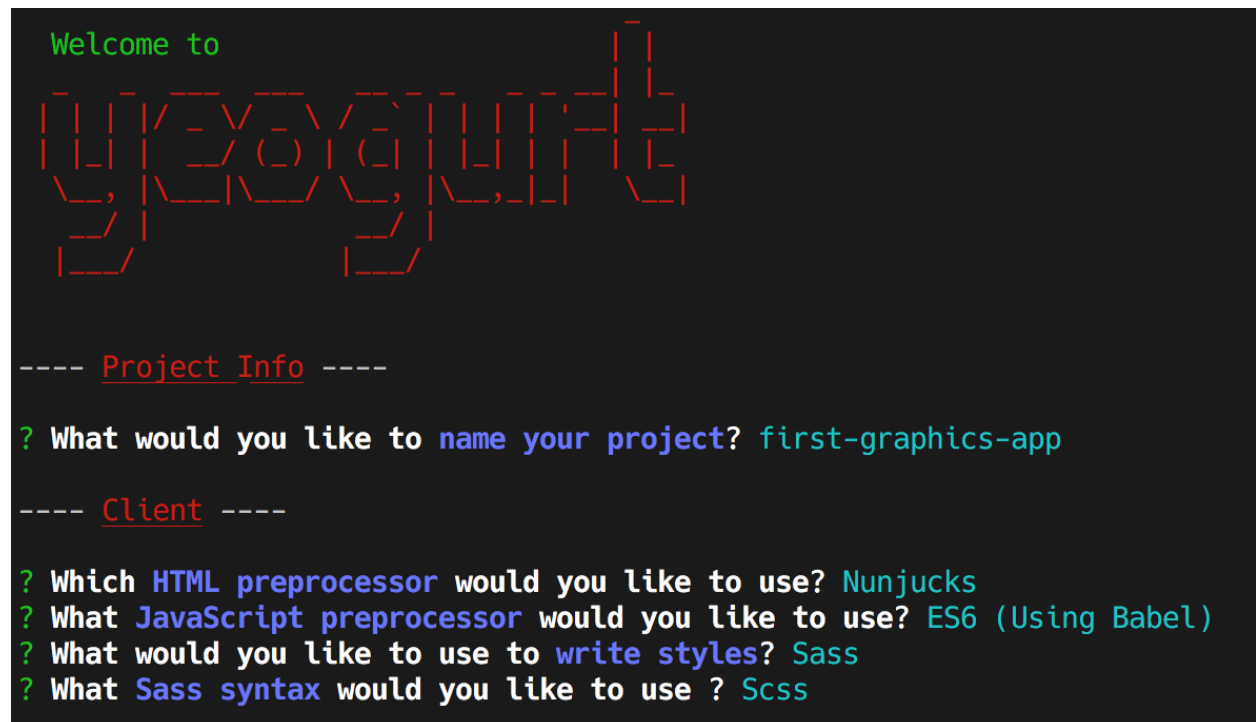
Finally, we use npm to install **yeogurt**, our project generator for Yeoman. It includes dozens of customizations created by its author to help us build websites. It can't do everything a full-featured newsroom framework might, but it can do enough for us to achieve our goals for this class.

```
$ npm install -g generator-yeogurt@3.1.2
```

Create a new project using our yeogurt generator as the guide.

```
$ yo yeogurt
```

After you run the command, you will be asked a series of questions. *Pay close attention* because you will need to choose the proper options to continue with our tutorial, and some of the correct selections are not the default choice.



```
Welcome to
yeogurt

---- Project Info ----
? What would you like to name your project? first-graphics-app

---- Client ----
? Which HTML preprocessor would you like to use? Nunjucks
? What JavaScript preprocessor would you like to use? ES6 (Using Babel)
? What would you like to use to write styles? Sass
? What Sass syntax would you like to use ? Scss
```

1. Your project name should be the slug “first-graphics-app”
2. The HTML preprocessor you choose must be “Nunjucks.”
3. Styles must be written with “Sass”
4. The Sass syntax must be “Scss”

Don't sweat the rest. But make sure you get the above right.

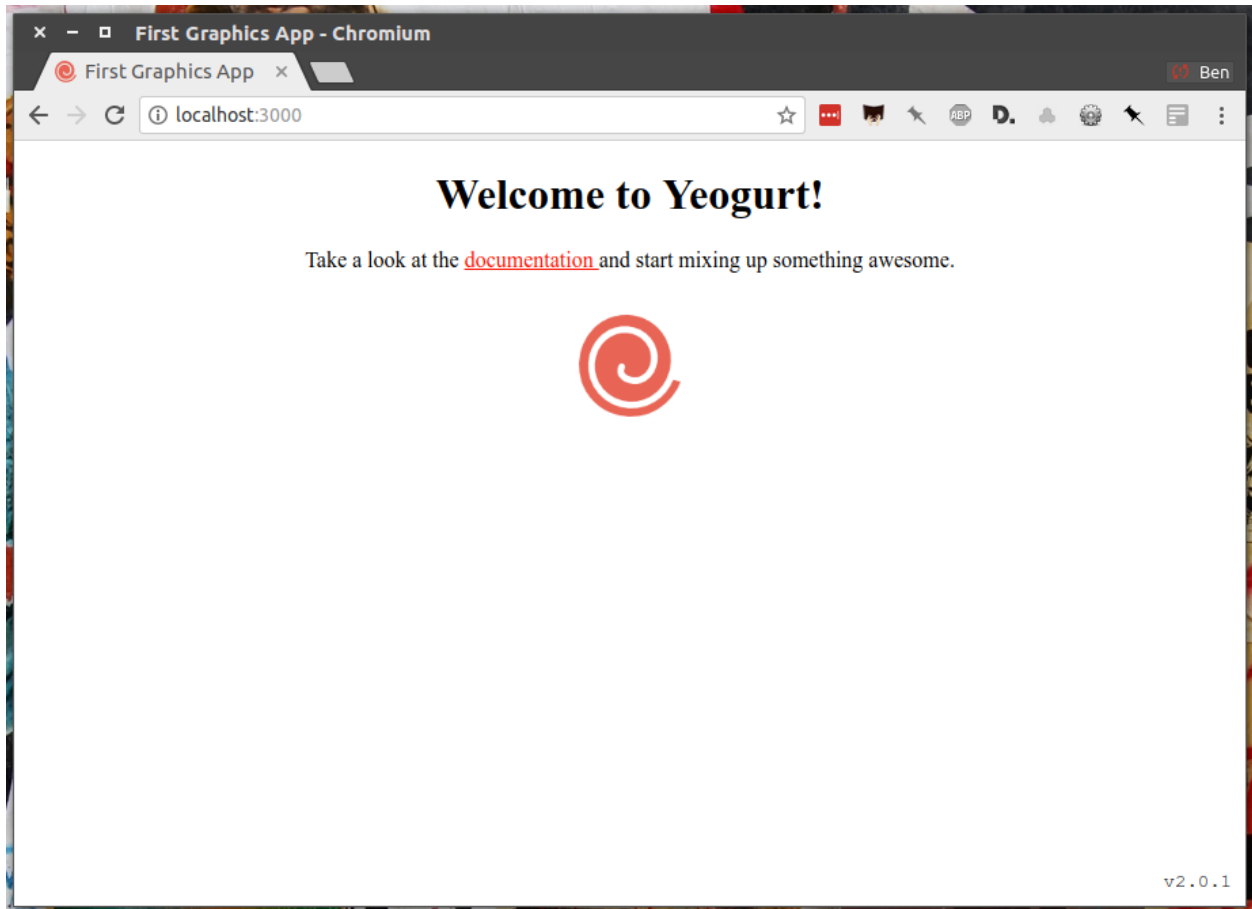
Yeoman will then use the generator to create a complete project that's ready for us to work in. Take a look at the folders its created in the `src` directory. That's the framework offering your a comfortable place to do your work. Let's

get in there set up shop.

First, fire up its test server to see what it has to offer out of the box.

```
$ npm run-script serve
```

Visit `localhost:3000` in your browser. There you can see the generic website offered as a starting point by our Yeoman generator.



Congratulations, you've got your framework up and running. Let's save our work and then we'll be ready to start developing our own content.

Note: You'll notice that all of the sub folders in the `src/` directory of your project have underscores `_` in front of their name. This convention is used to note that these files are **private**, and won't be deployed to your live site.

Instead, Gulp processes the contents of these folders when it builds the project and serves the files from a `tmp/` folder, where you'll see unprefixed `images/`, `scripts/` and `styles/` directories.

Open a second terminal (this way you can keep your server running) and navigate to your code folder.

```
$ cd Code
$ cd first-graphics-app
```

Commit our work.

```
$ git add .  
# A fun trick to add *all* of the pages you've changed with one command.  
$ git commit -m "Installed framework"
```

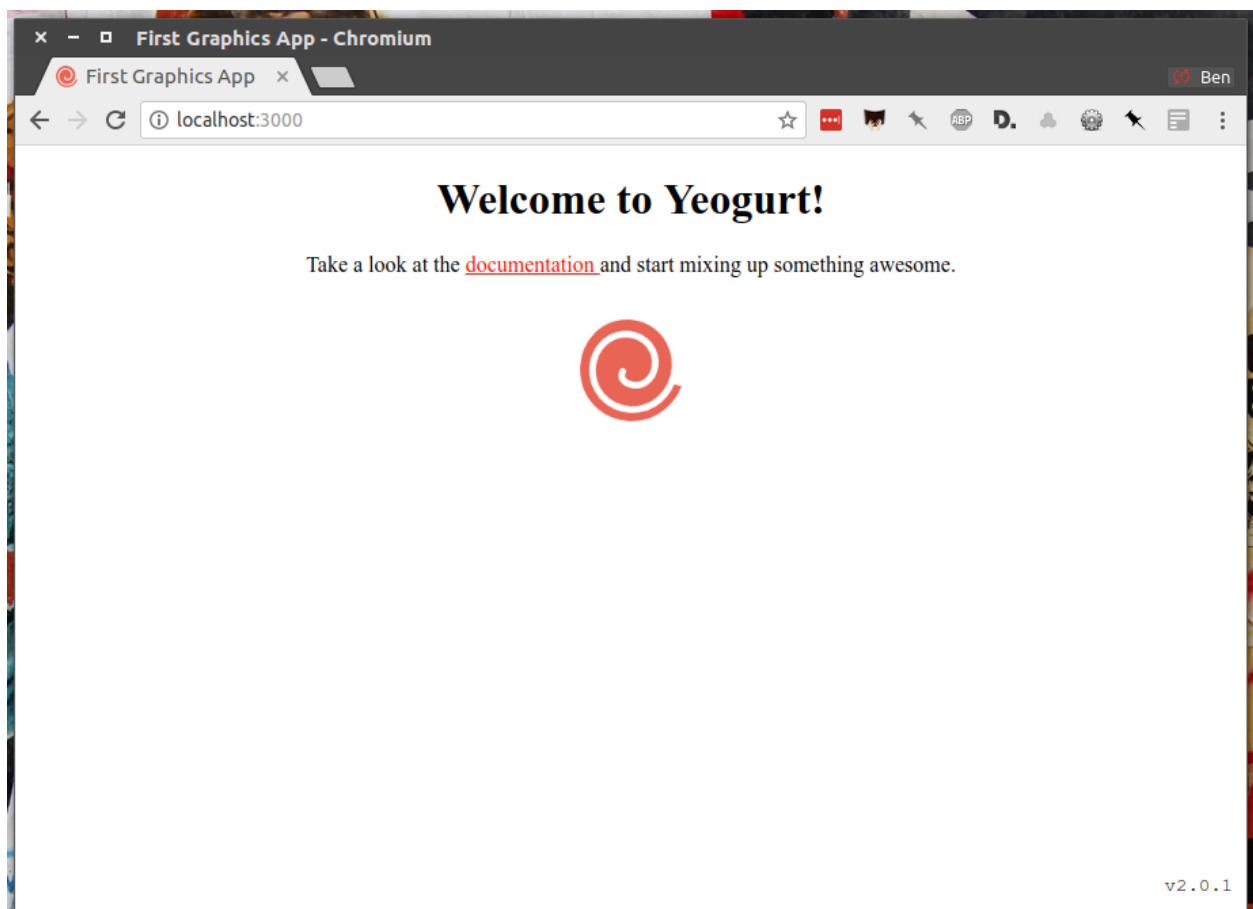
Push it to GitHub.

```
$ git push origin master
```


CHAPTER 6

Chapter 3: Hello template

Navigate back to `localhost:3000` in your browser. You should see the same default homepage as before.



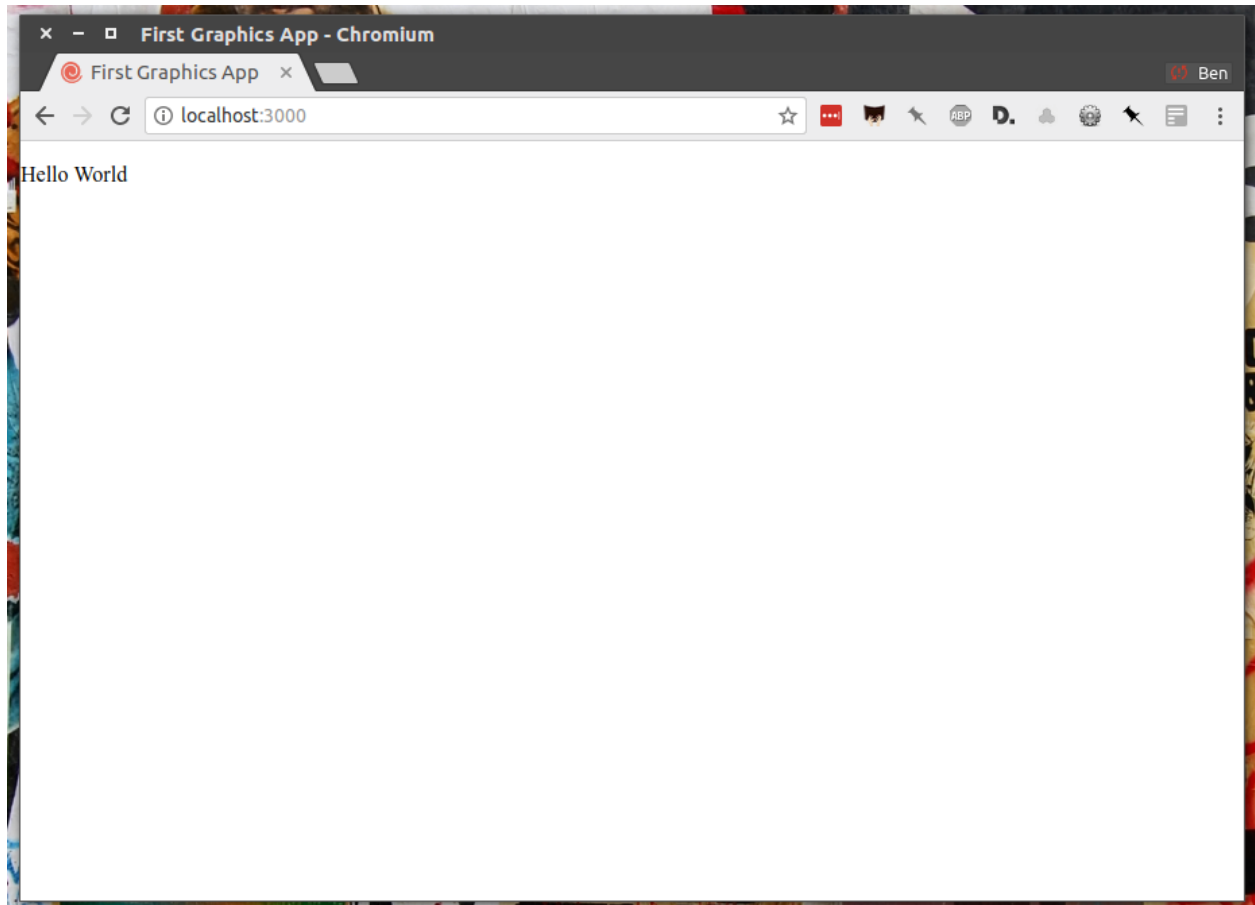
Its contents is configured in the `index.nunjucks` file found in the directory Yeoman created. It uses a templating

language for JavaScript invented at Mozilla called [Nunjucks](#).

You can edit the page by changing what’s found inside of the `content` block. Make a change and save the file.

```
{% block content %}  
<p>Hello World</p>  
{% endblock %}
```

You should see it immediately show up thanks to a [BrowserSync](#), a popular feature of Gulp that automatically updates your test site after you make a change.



Now, look closely at the `index.nunjucks` file. You will notice that it doesn’t include code for much of what you expect from an HTML page. For instance, you won’t see the `<html>` or `<body>` tags. Nor will you find the stylesheets that dictate how a page looks.

That’s because that boilerplate has been moved back into a parent template “extended” by the index file with a line of Nunjucks code at the top of the page.

```
{% extends '_layouts/base.nunjucks' %}
```

That “base” file, sometimes called the “layout,” can be inherited by other pages on your site to avoid duplication and share common code. One change to a parent file instantly ripples out to all pages the extend it.

This approach to “template inheritance” is not just found in Nunjucks. It can be found in other templating systems, including Python ones like [Django](#) and [Jinja](#). It’s probably even used at some level in your organization’s CMS.

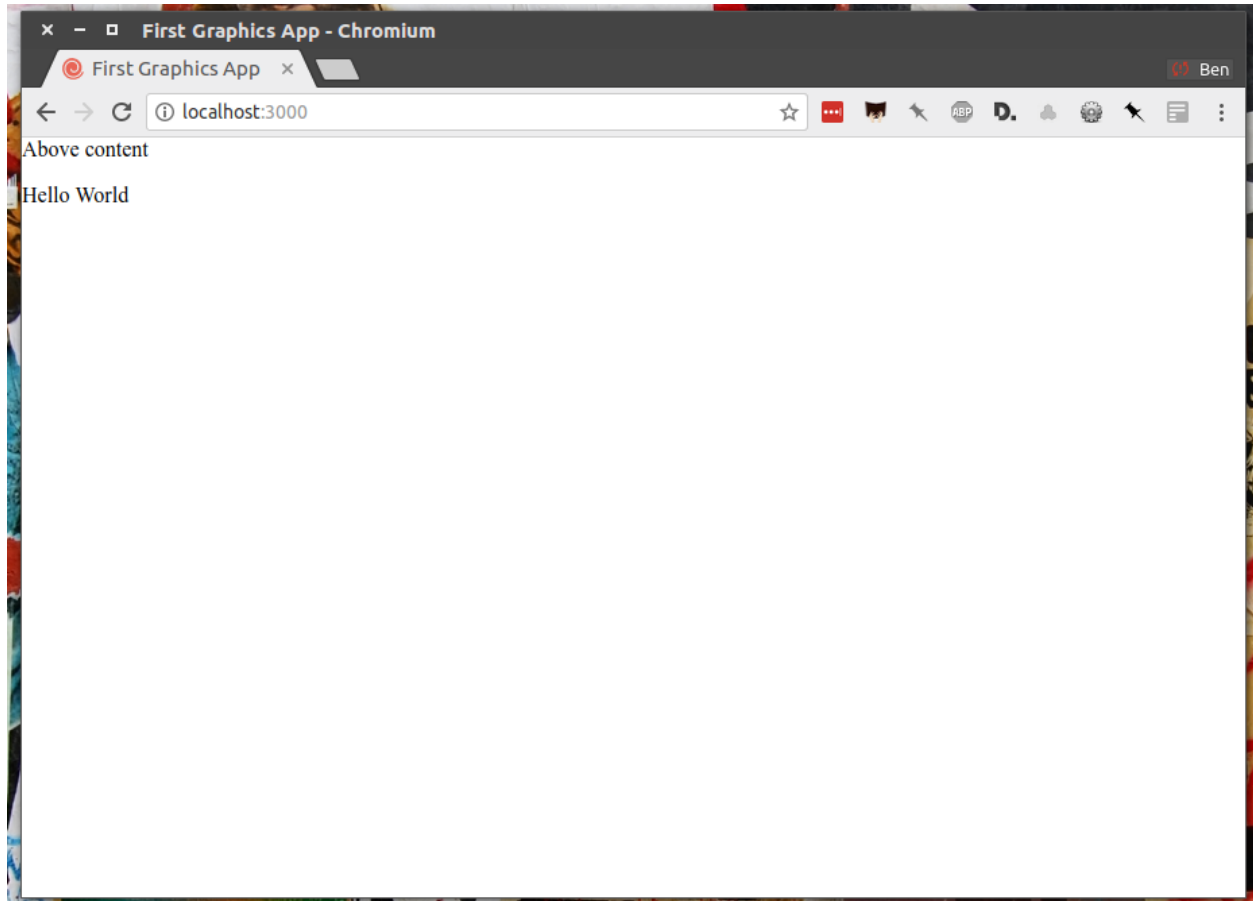
You can find the base layout packaged with our framework by following the path and opening the `_layouts/base.nunjucks` file. You’ll see it includes a set of block tags, like `content`, that act as placeholders for use in templates

that extend it.

Make a small change to `_layouts/base.nunjucks` above the `content` block and save the file.

```
Above content
{% block content %}{% endblock %}
```

You should see the change on our site, with the new line appearing above the paragraph we added earlier to the index file.



Most newsrooms that use a similar system have their own base template for their custom pages. Graphic artists and designers install and extend it as the first step in their work. They develop their custom page within its confines and largely accept the furniture it provides, like the site's header and footer, fonts and common color schemes. This allows them to work more quickly because they do not have to bother with reinventing their site's most common elements.

Note: While most newsrooms keep their base templates to themselves, a few have published them as open-source software. You can find them online, if you know where to look. They include:

- The Los Angeles Times's [HTML Cookbook](#)
- The Texas Tribune's [style guide](#)
- Politico's [style guide](#)

For this class, we have developed a simplified base template that will act as a stand-in for a real newsroom's base template. It is not as sophisticated or complete as a real-world example, but it will provide all of the basic elements we will need.

You can find it in the code block below. Copy all of its contents and paste them into `_layouts/base.nunjucks`, replacing everything.

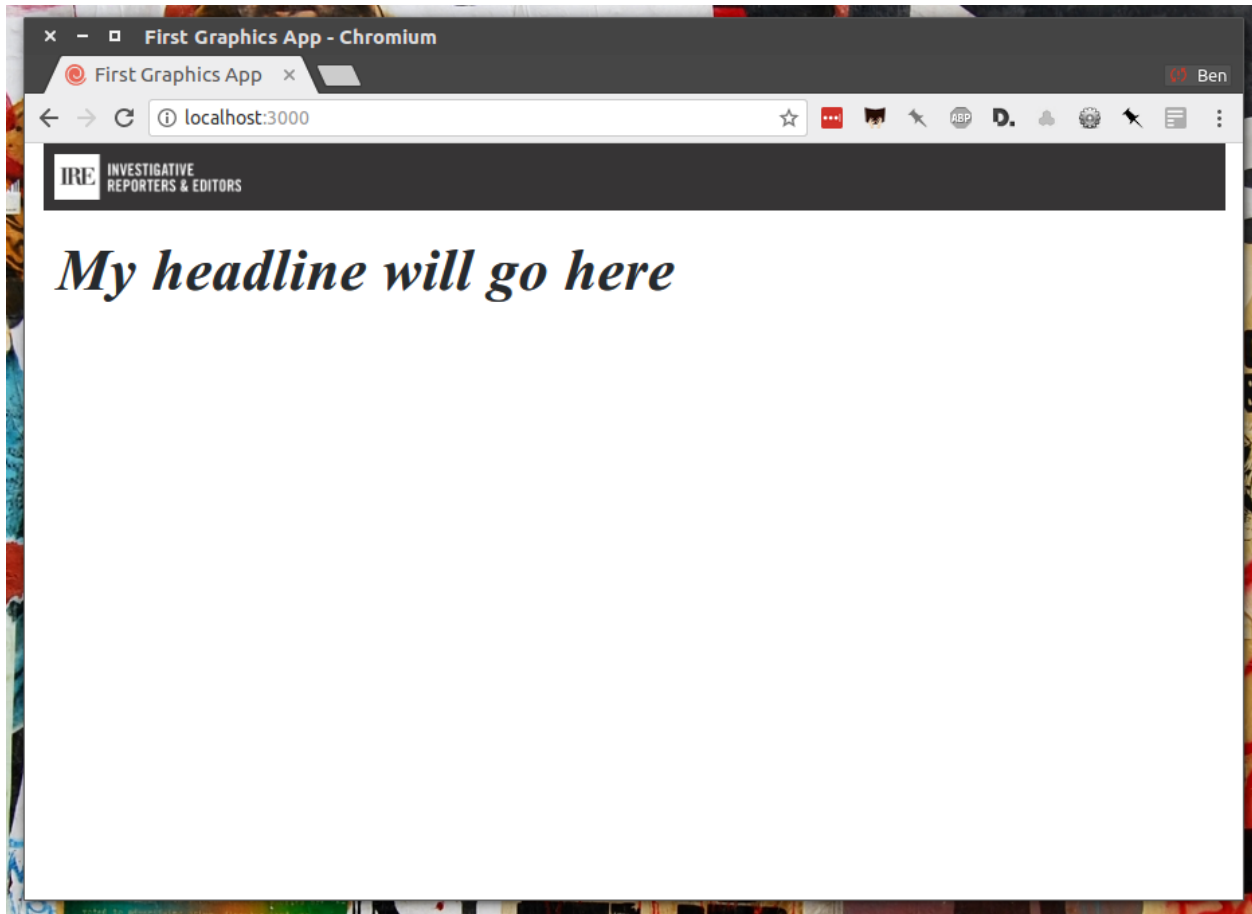
```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>First Graphics App</title>
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/
↪bootstrap.min.css">
  <link rel="stylesheet" href="styles/main.css">
  <link rel="stylesheet" href="https://bl.ocks.org/palewire/raw/
↪1035cd306a2f85b362b1a20ce315b8eb/base.css?rev=8">
  {% block stylesheets %}{% endblock %}
</head>
<body>
  <nav>
    <a href="http://first-graphics-app.readthedocs.org/">
      
    </a>
  </nav>
  <header>
    <h1>{% block headline %}{% endblock %}</h1>
    <div class="byline">
      {% block byline %}{% endblock %}
    </div>
    <div class="pubdate">
      {% block pubdate %}{% endblock %}
    </div>
  </header>
  {% block content %}{% endblock %}
  {% block scripts %}{% endblock %}
  <script src="scripts/main.js"></script>
</body>
</html>
```

As you can see, it includes all of the standard HTML tags, with our custom stylesheets and content blocks mixed in.

To see the effects, return to `index.nunjucks` and fill in a headline using the headline block introduced by our base template. Save the page and you should quickly see it appear on the page.

```
{% extends '_layouts/base.nunjucks' %}

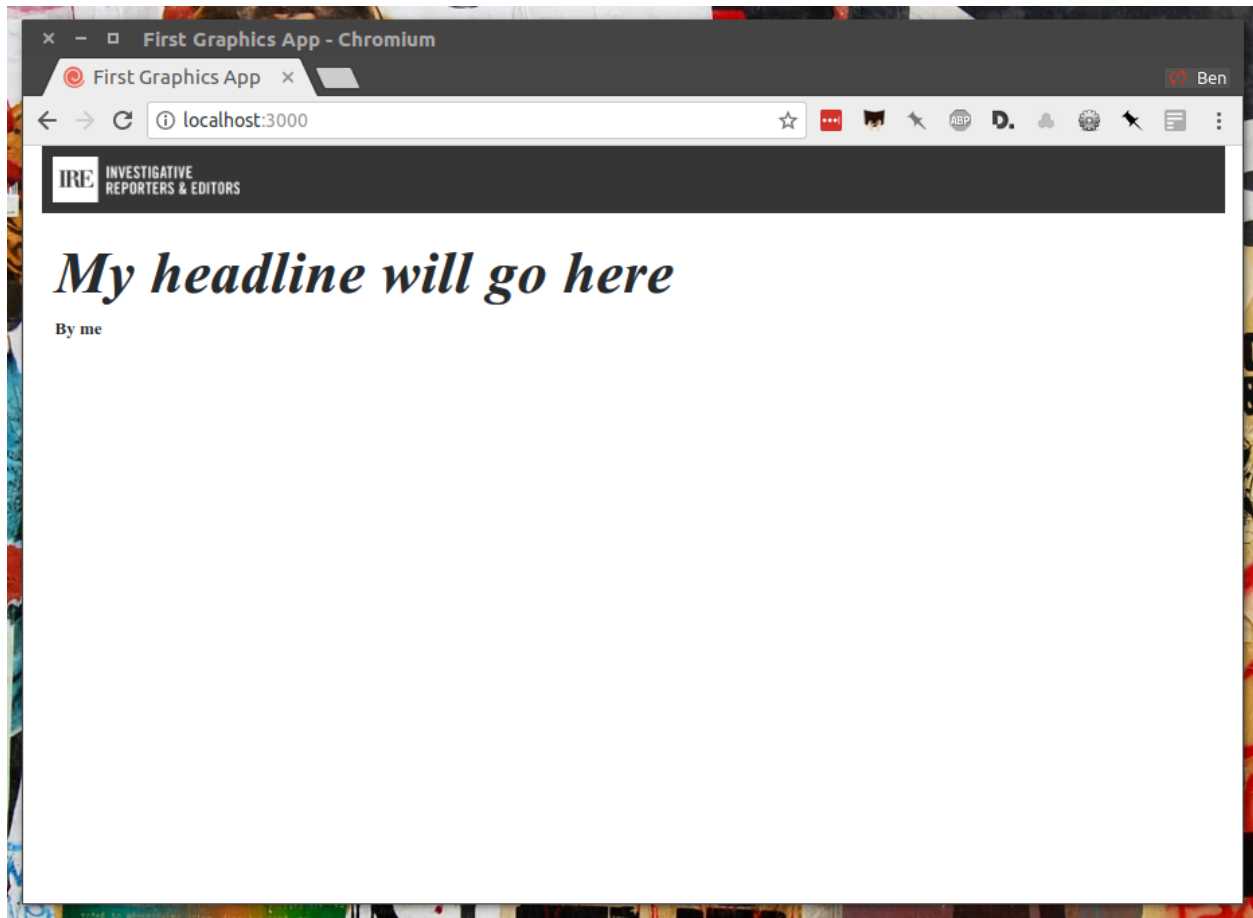
{% block headline %}My headline will go here{% endblock %}
```



Now fill in a byline.

```
{% extends '_layouts/base.nunjucks' %}

{% block headline %}My headline will go here{% endblock %}
{% block byline %}By me{% endblock %}
```



And let's do the publication date too while we are at it.

```
{% extends '_layouts/base.nunjucks' %}

{% block headline %}My headline will go here{% endblock %}
{% block byline %}By me{% endblock %}
{% block pubdate %}
  <time datetime="2020-03-07" pubdate>Mar. 7, 2020</time>
{% endblock %}
```



My headline will go here

By me

MAR. 7, 2020

Hello World

Congratulations, you've installed a base template and started in on creating your first custom page. Now is another good time to pause and commit your work.

```
$ git add .  
$ git commit -m "Started editing templates"
```

And, again, push it to GitHub.

```
$ git push origin master
```


CHAPTER 7

Chapter 4: Hello data

We've got our system set up. Now it's time to start telling our story. To do that, we need our data.

If we were writing this application entirely in the browser with traditional JavaScript we'd have to pull it in with dynamic "AJAX" calls that retrieve data over the web as the page is constructed. But since we're working with a Node.js system, running code here on the backend, we can import data directly into the template instead and lay it out before the page is rendered in the browser. This results in a faster experience for our users, and opens up new ways for us to be creative with our data.

Every newsroom's system will handle this differently. Our Yeoman generator is preconfigured to open all JSON data files in the `_data` folder and import them into our Nunjucks templates.

Let's give it a try. Grab the [list of Harvard Park homicides](#) published by the Los Angeles Times and save it to `_data/harvard_park_homicides.json`. It includes every homicide victim in the neighborhood since 2000 in the [JSON data format](#) favored by JavaScript.

```
[
  {
    "case_number": "2017-04514",
    "slug": "eddie-rosendo-lino",
    "first_name": "Eddie",
    "middle_name": "Rosendo",
    "last_name": "Lino",
    "death_date": "2017-06-18T00:00:00.000Z",
    "death_year": 2017,
    "age": 23.0,
    "race": "black",
    "gender": "male",
    "image": null,
    "longitude": -118.304107484,
    "latitude": 33.9904336958
  },
  {
    "case_number": "2017-03454",
    "slug": "alex-david-lomeli",
    "first_name": "Alex",
```

(continues on next page)

(continued from previous page)

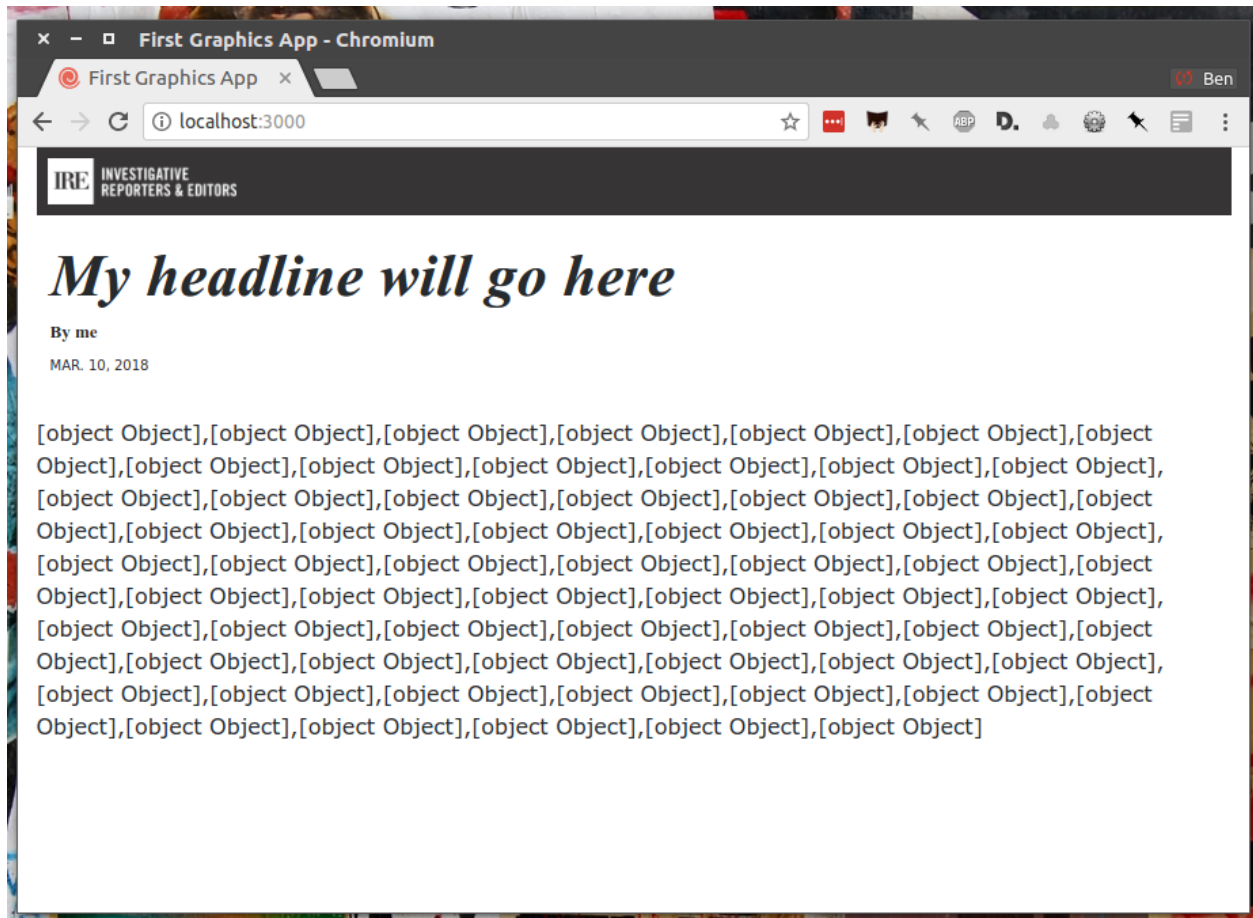
```
"middle_name": "David",
"last_name": "Lomeli",
"death_date": "2017-05-07T00:00:00.000Z",
"death_year": 2017,
"age": 18.0,
"race": "latino",
"gender": "male",
"image": null,
"longitude": -118.300290584,
"latitude": 33.9793646958
},
...
```

Return to `index.nunjucks` and add the following to the bottom to print the data out on the page. We can do that using the `{{ }}` print tags provided by Nunjucks.

```
{% block content %}
  {{ site.data.harvard_park_homicides }}
{% endblock %}
```

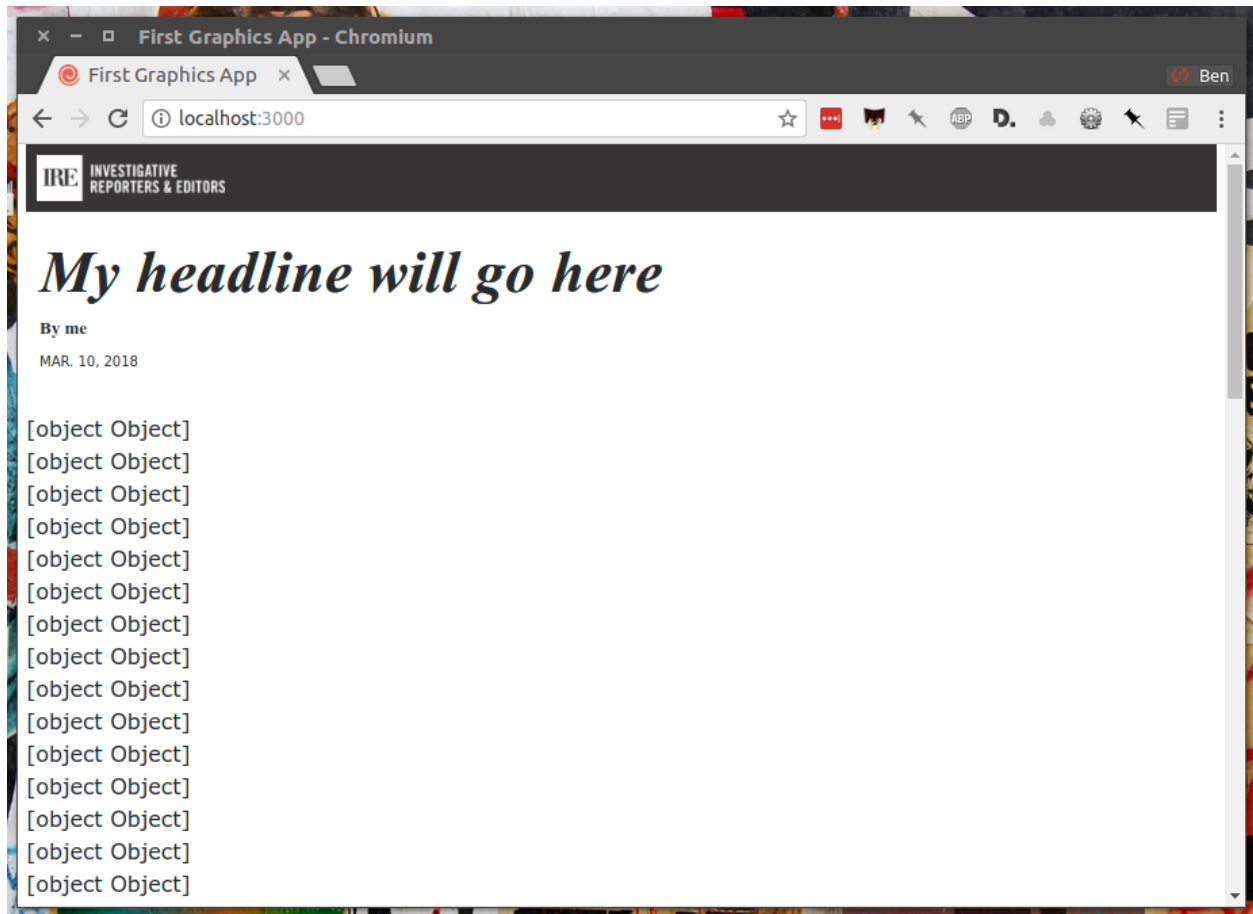
Instead of just printing the data in one big block, let's loop through the records and print them one by one. We'll use the `{% %}` template tags provided by Nunjucks, which allow you to use common computer programming logic when you're laying out a page.

```
{% block content %}
{% for obj in site.data.harvard_park_homicides %}
  {{ obj }}
{% endfor %}
{% endblock %}
```



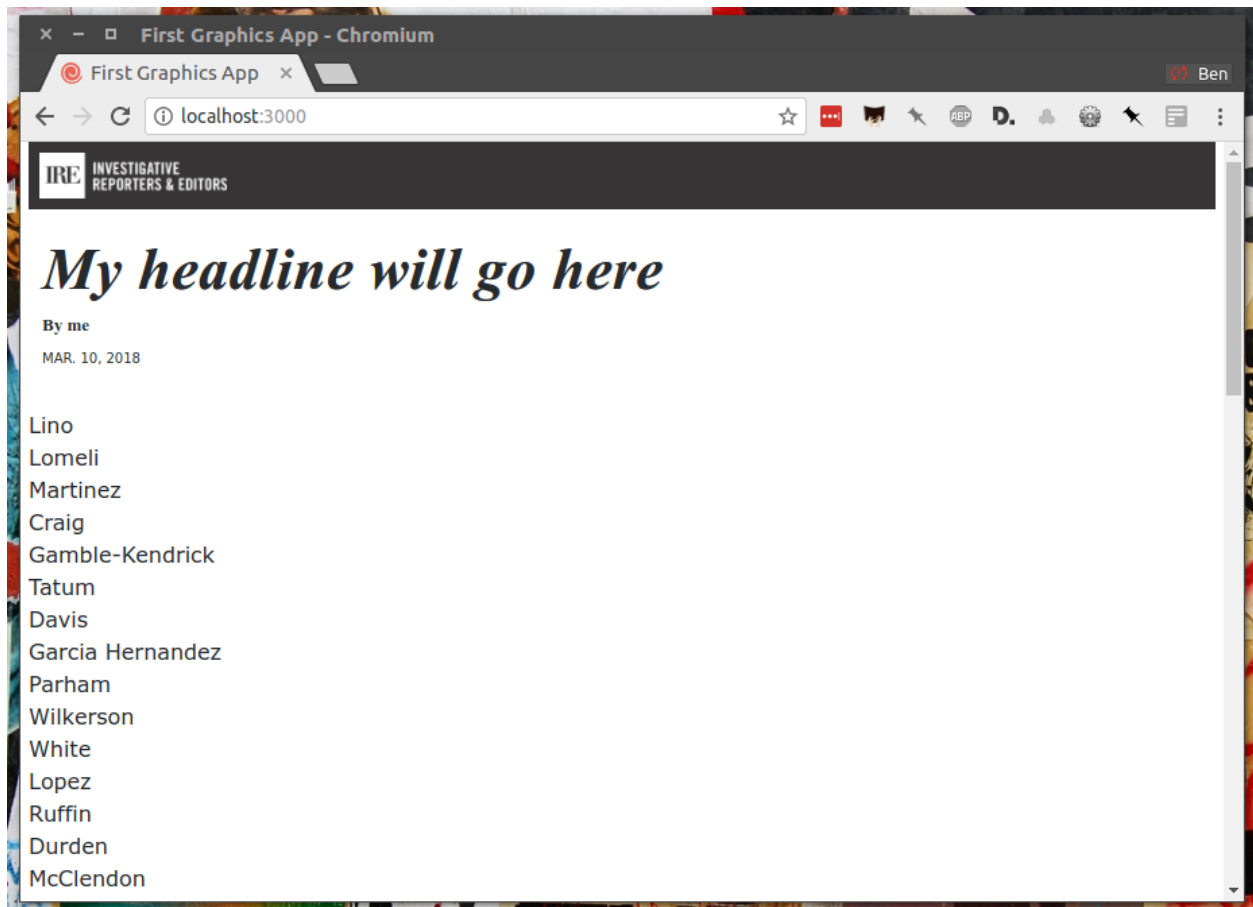
To put each one on its own line, add a line break with a `
` tag. That's just boring old HTML. Writing pages with a templating language like Nunjucks is typically nothing more than mixing traditional HTML with the template tags that negotiate your data files and other variables.

```
{% block content %}
{% for obj in site.data.harvard_park_homicides %}
  {{ obj }}<br>
{% endfor %}
{% endblock %}
```



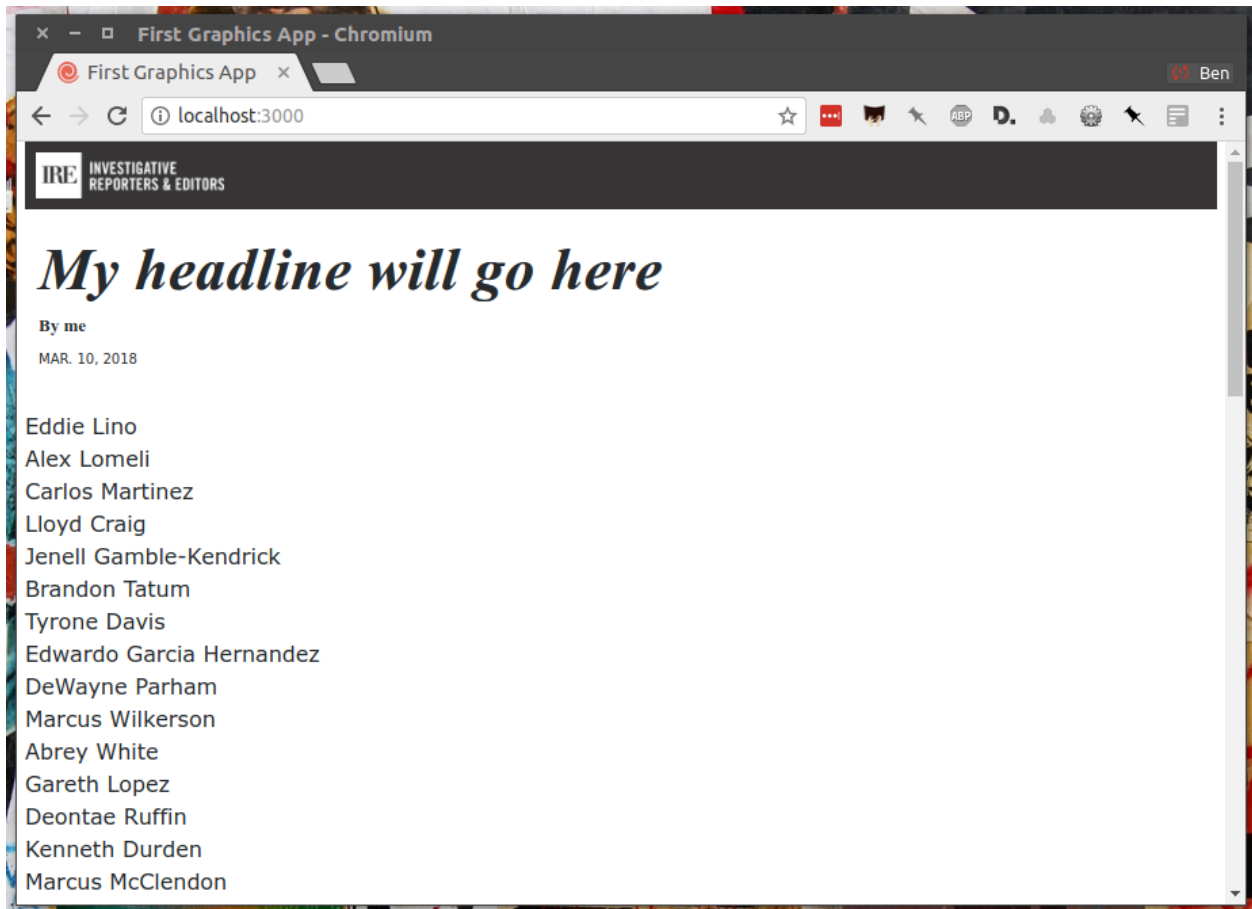
That's good, but hardly informative. How do we start printing out the contents of the data? The fields in the JSON dictionary for each homicide are available by adding a `.` after the object. For instance, here's how to print the contents of the `last_name` field.

```
{% block content %}
{% for obj in site.data.harvard_park_homicides %}
    {{ obj.last_name }}<br>
{% endfor %}
{% endblock %}
```



Now the first name.

```
{% block content %}
{% for obj in site.data.harvard_park_homicides %}
  {{ obj.first_name }} {{ obj.last_name }}<br>
{% endfor %}
{% endblock %}
```



Not bad. We've actually got some data on the page. Seems like a good moment to stop, take a break and commit our work.

```
$ git add .  
$ git commit -m "Printed a list of names from data"
```

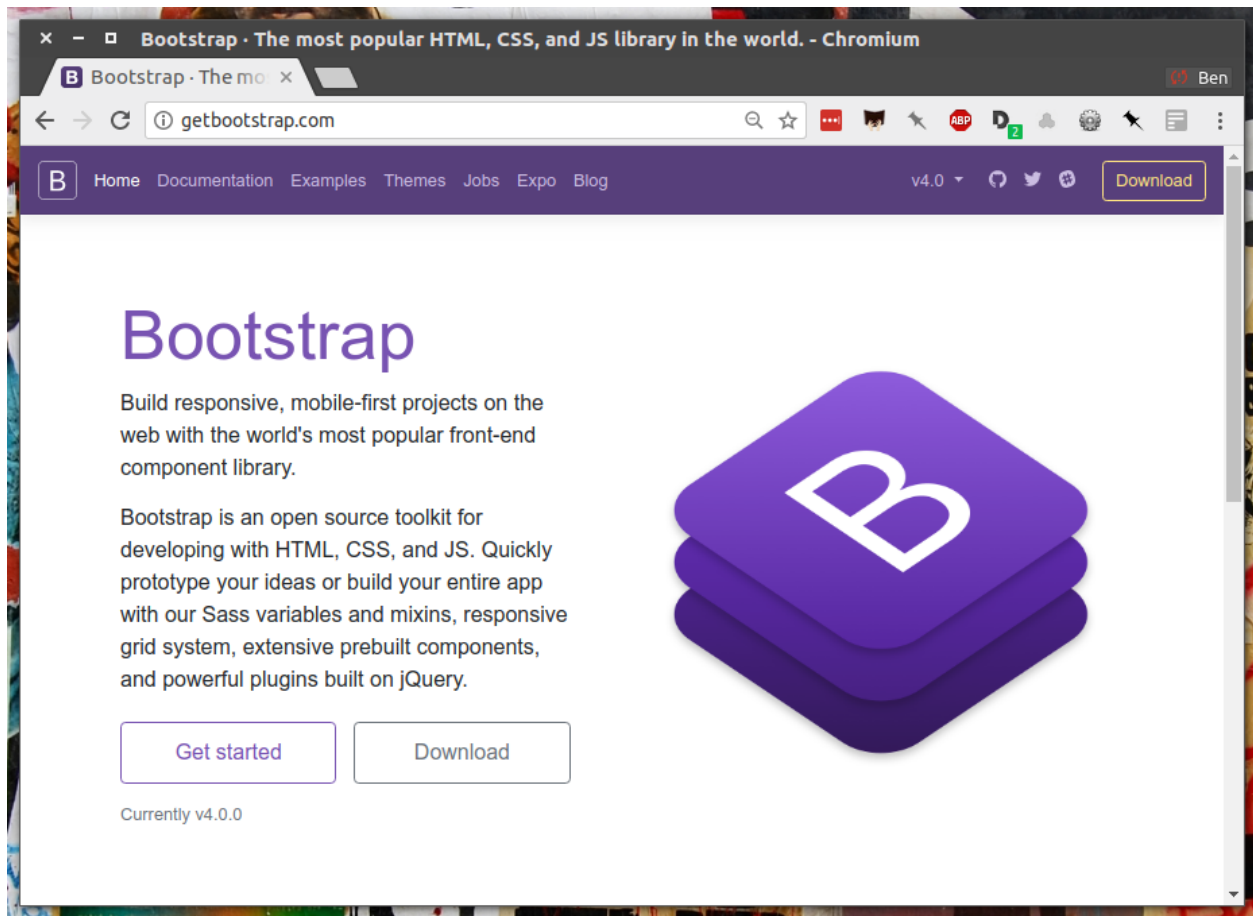
Push it to GitHub.

```
$ git push origin master
```

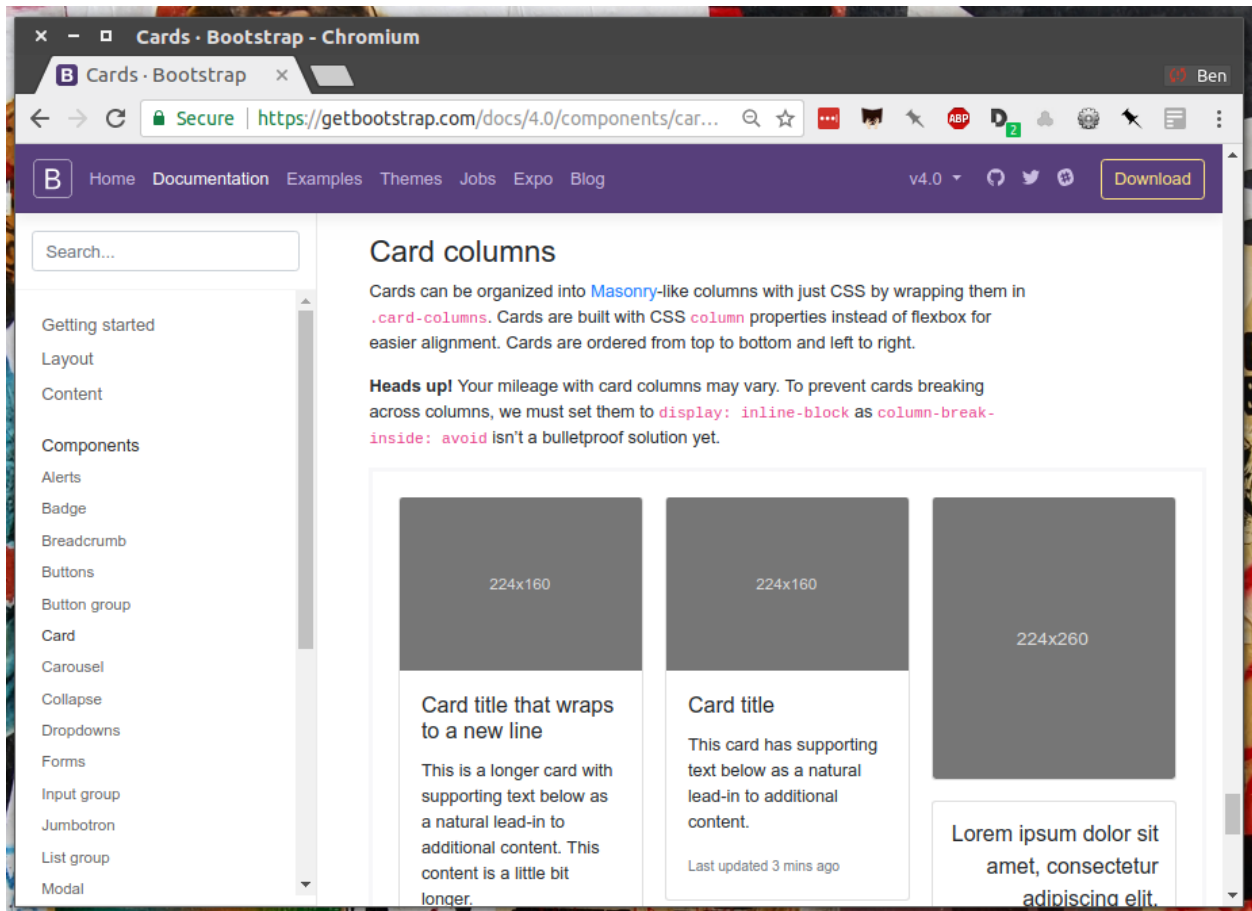
Chapter 5: Hello cards

Bootstrap is an HTML, CSS and JavaScript toolkit that you can use to create the cosmetic “front-end” of web applications. It is a collection of ready-to-use pieces called components, which are building blocks you can mix and match to help jumpstart a project. Its components can be used as-is or as a base to be customized by the developer.

The components library includes things that you might include in a project, like buttons, modals and dropdowns.



We're going to create a photo grid of pictures of Harvard Park homicide victims. Each grid block will have a picture and some basic information. We're going to use the "cards" component included in Bootstrap's version 4 to accomplish this. Cards are self-contained boxes of information which can be arranged and grouped on a page any way you want.



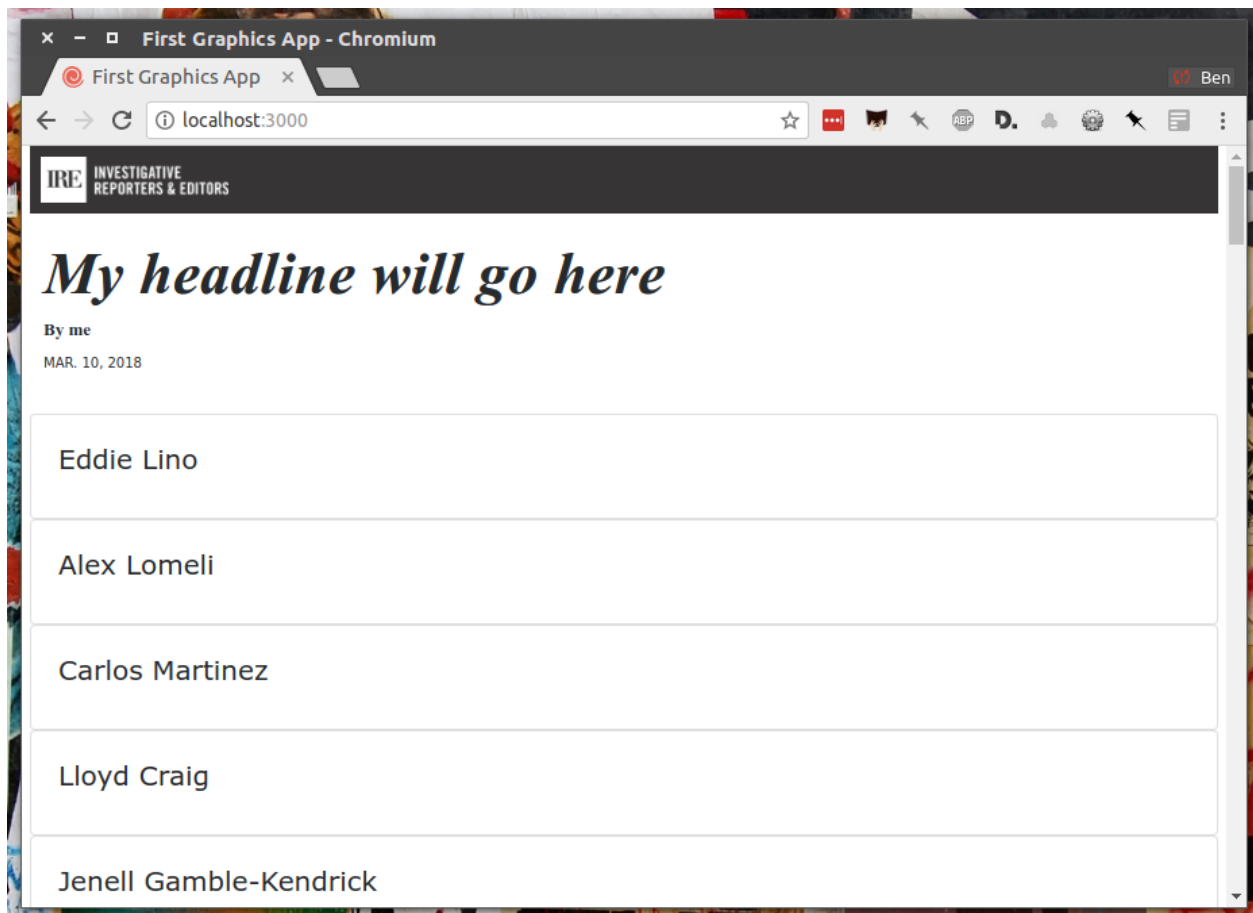
First, we need to set up our grid. To do that, we need to talk about [HTML's division tag](#), also known as a `<div>`. The simplest way to think of a div is as container. Like any container, divs hold things. Divs can be nested inside of each other, like putting a box inside a box.

This is how Bootstrap cards work. Each card is a container which has additional containers inside it to hold, in this case, a picture, the victim's name, age, race and when he or she was killed.

Like other HTML tags, divs can have `class` attributes that help identify their function and link them to cosmetic styles. Bootstrap provides us with a standard layout of divs that, if structured and labeled properly, will instantly snap together to look like cards.

Let's give it a try. We will start by creating a container div for each victim. We'll add just the name of the first and last name of each victim first.

```
{% block content %}
{% for obj in site.data.harvard_park_homicides %}
  <div class="card">
    <div class="card-body">
      <h5 class="card-title">{{ obj.first_name }} {{ obj.last_name }}</h5>
    </div>
  </div>
{% endfor %}
{% endblock %}
```

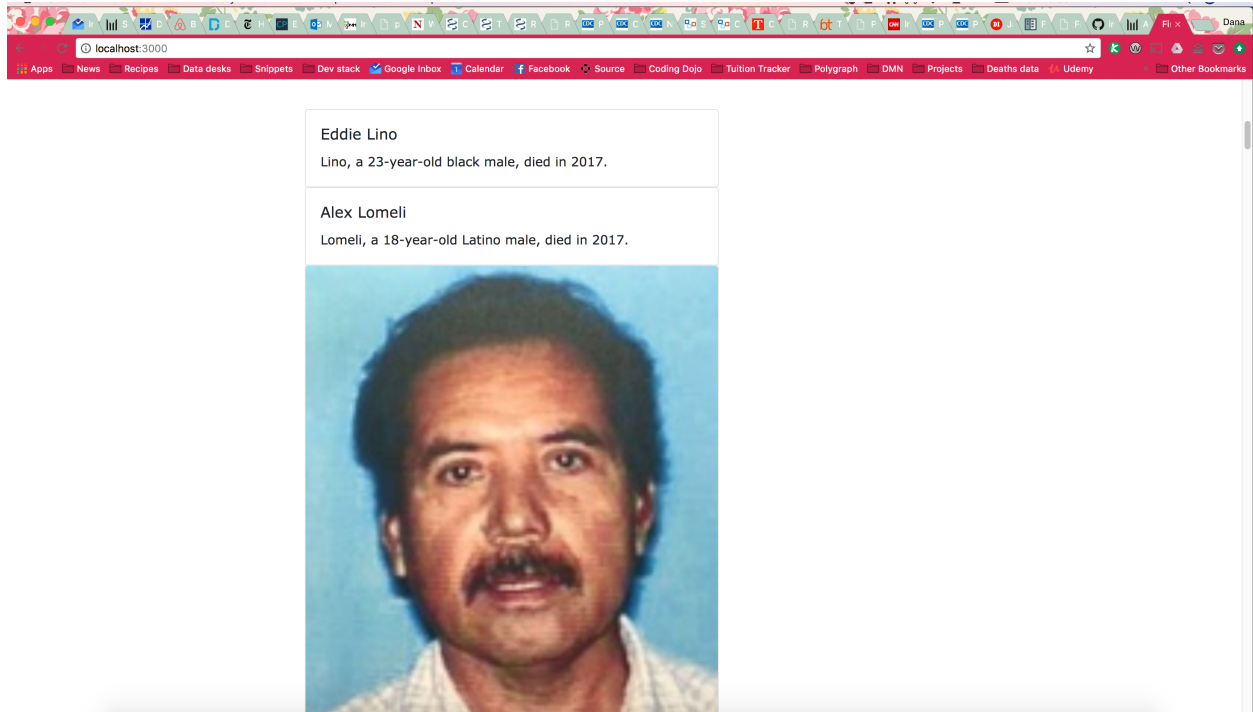


Let's add a sentence below the name that summarizes who each victim was and when they died.

```
{% for obj in site.data.harvard_park_homicides %}
  <div class="card">
    <div class="card-body">
      <h5 class="card-title">{{ obj.first_name }} {{ obj.last_name }}</h5>
      <p class="card-text">{{ obj.last_name }}, a {{ obj.age }}-year-old {{ obj.race_
      ↳}} {{ obj.gender }}, died in {{ obj.death_year }}.</p>
    </div>
  </div>
{% endfor %}
```

Now let's add each victim's image to their card.

```
{% for obj in site.data.harvard_park_homicides %}
  <div class="card">
    
    <div class="card-body">
      <h5 class="card-title">{{ obj.first_name }} {{ obj.last_name }}</h5>
      <p class="card-text">{{ obj.last_name }}, a {{ obj.age }}-year-old {{ obj.race_
      ↳}} {{ obj.gender }}, died in {{ obj.death_year }}.</p>
    </div>
  </div>
{% endfor %}
```



Ugh. That's a lot of broken images. To fix it, let's add an `if` clause around the image tag to check for an image in the data. This way, our code will loop through the list of victims and `_if_` there is an image it will add it to the right card. If not, the code will move on to the next row in the data.

```
{% for obj in site.data.harvard_park_homicides %}
  <div class="card">
    {% if obj.image %}{% endif %}
    <div class="card-body">
      <h5 class="card-title">{{ obj.first_name }} {{ obj.last_name }}</h5>
      <p class="card-text">{{ obj.last_name }}, a {{ obj.age }}-year-old {{ obj.race_
}} {{ obj.gender }}, died in {{ obj.death_year }}.</p>
    </div>
  </div>
{% endfor %}
```

Phew. Better. But what we've got so far is a grid that doesn't look much like a grid. In fact it's not a grid at all. It's just a big stack.

To arrange our cards using Bootstrap's system, we need to play by Bootstrap's rules. Look at its documentation and you'll see that Bootstrap asks us to wrap our cards in a div with the class `card-columns` if we want a grid. Like this:

```
<div class="card-columns">
  {% for obj in site.data.harvard_park_homicides %}
    <div class="card">
      {% if obj.image %}{% endif %}
      <div class="card-body">
        <h5 class="card-title">{{ obj.first_name }} {{ obj.last_name }}</h5>
        <p class="card-text">{{ obj.last_name }}, a {{ obj.age }}-year-old {{ obj.race_
}} {{ obj.gender }}, died in {{ obj.death_year }}.</p>
      </div>
    </div>
  {% endfor %}
</div>
```

(continues on next page)

(continued from previous page)

```

</div>
{% endfor %}
</div>

```

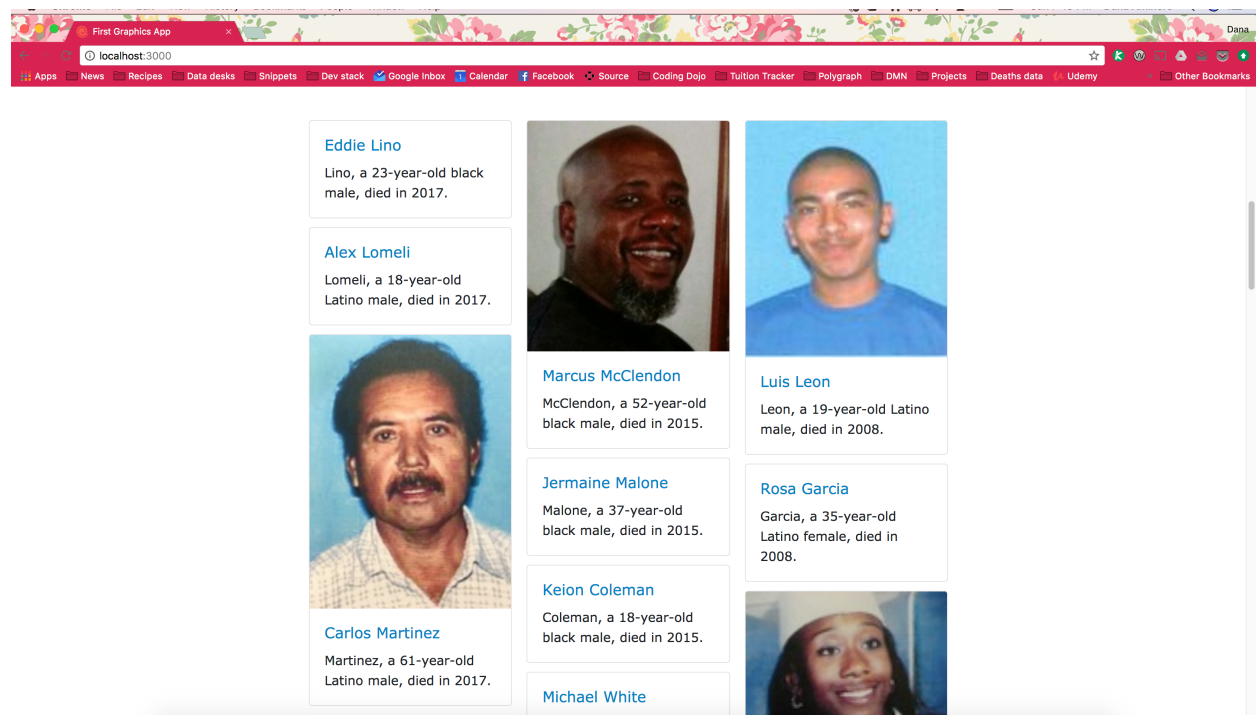
Note that the new div is outside of our `for` loop, meaning it only appears on the page once with all of the card divs inside of it.

We're not done yet. We want to be able to click on each card and be redirected to the victim's page on the Los Angeles Times Homicide Report site. While we're at it, let's add some `` tags to the victims' names to make them stand out from the sentence about them.

```

<div class="card-columns">
  {% for obj in site.data.harvard_park_homicides %}
  <div class="card">
    {% if obj.image %}{% endif %}
    <div class="card-body">
      <a href="http://homicide.latimes.com/post/{{ obj.slug }}" target="_blank">
        <strong>
          <h5 class="card-title">{{ obj.first_name }} {{ obj.last_name }}</h5>
        </strong>
      </a>
      <p class="card-text">{{ obj.last_name }}, a {{ obj.age }}-year-old {{ obj.race_
→ }} {{ obj.gender }}, died in {{ obj.death_year }}.</p>
    </div>
  </div>
  {% endfor %}
</div>

```



Now let's start to wrap things up by writing a headline for our cards section.

```

<h3>Lives lost in Harvard Park</h3>
<div class="card-columns">

```

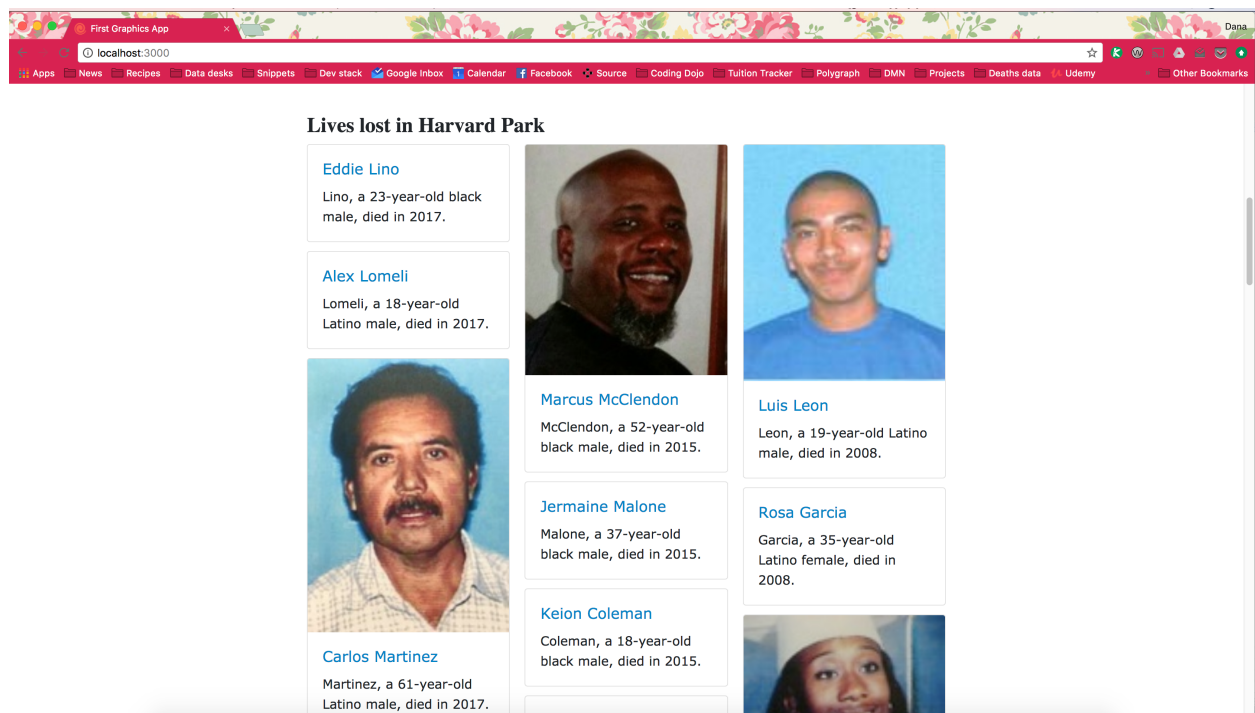
(continues on next page)

(continued from previous page)

```

{% for obj in site.data.harvard_park_homicides %}
<div class="card">
  {% if obj.image %}{% endif %}
  <div class="card-body">
    <a href="http://homicide.latimes.com/post/{{ obj.slug }}" target="_blank">
      <strong>
        <h5 class="card-title">{{ obj.first_name }} {{ obj.last_name }}</h5>
      </strong>
    </a>
    <p class="card-text">{{ obj.last_name }}, a {{ obj.age }}-year-old {{ obj.race_
→}} {{ obj.gender }}, died in {{ obj.death_year }}.</p>
  </div>
</div>
{% endfor %}
</div>

```



And now, some introductory text. We can use a new templating trick, the length filter, to insert some automatically generated statistics into the text.

```

<h3>Lives lost in Harvard Park</h3>
<p>The {{ site.data.harvard_park_homicides|length }} homicides in Harvard Park since
→2000 were primarily black and Latino males, but the list includes husbands, wives,
→fathers, mothers of all ages, and even some small children.</p>
<div class="card-columns">
  {% for obj in site.data.harvard_park_homicides %}
  <div class="card">
    {% if obj.image %}{% endif %}
    <div class="card-body">
      <a href="http://homicide.latimes.com/post/{{ obj.slug }}" target="_blank">
        <strong>
          <h5 class="card-title">{{ obj.first_name }} {{ obj.last_name }}</h5>
        </strong>
      </a>
    </div>
  </div>
  {% endfor %}
</div>

```

(continues on next page)

(continued from previous page)

```

        </a>
        <p class="card-text">{{ obj.last_name }}, a {{ obj.age }}-year-old {{ obj.race_
↪}} {{ obj.gender }}, died in {{ obj.death_year }}.</p>
    </div>
</div>
    {% endfor %}
</div>

```

Let's set up our card grid as it's own section by adding `<section>` tags. This is simple example of adding some hidden stucture to your page so its easier for search engines and other spiders to parse.

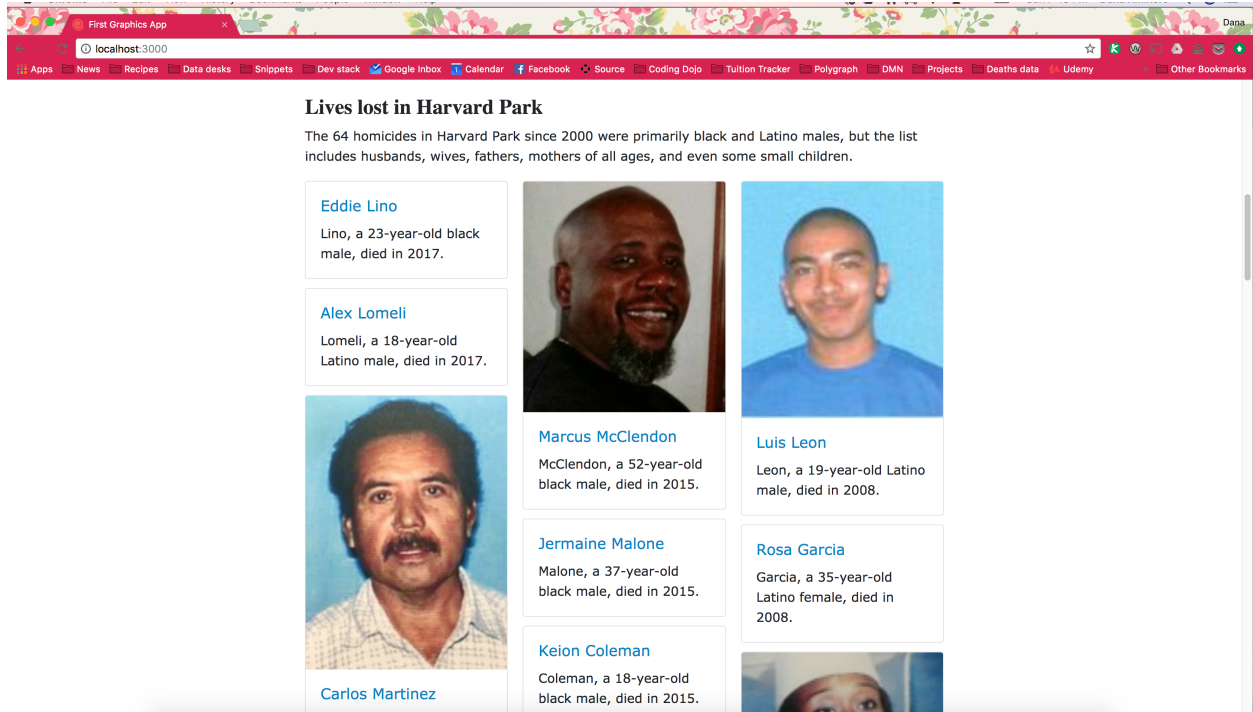
```

<section>
    <h3>Lives lost in Harvard Park</h3>
    <p>The {{ site.data.harvard_park_homicides|length }} homicides in Harvard Park_
↪since 2000 were primarily black and Latino males, but the list includes husbands,
↪wives, fathers, mothers of all ages, and even some small children.</p>
    <div class="card-columns">
        {% for obj in site.data.harvard_park_homicides %}
        <div class="card">
            {% if obj.image %}{% endif
↪%}

            <div class="card-body">
                <a href="http://homicide.latimes.com/post/{{ obj.slug }}" target="_blank">
                    <strong>
                        <h5 class="card-title">{{ obj.first_name }} {{ obj.last_name }}</
↪h5>

                        </strong>
                    </a>
                    <p class="card-text">{{ obj.last_name }}, a {{ obj.age }}-year-old {{ obj.
↪race }} {{ obj.gender }}, died in {{ obj.death_year }}.</p>
                </div>
            </div>
            {% endfor %}
        </div>
</section>

```



We're all done here. So let's commit our work for this chapter.

```
$ git add .  
$ git commit -m "Created a victims card grid"
```

Push it to GitHub.

```
$ git push origin master
```


CHAPTER 9

Chapter 6: Hello charts

To visualize our data, we're going to use the [D3.js](#) library, which has become the industry standard for creating custom data visualizations. Because it is so flexible and allows for so many different data-driven expressions, D3 powers many of the news graphics made with JavaScript you see online.

Note: We're going to dive straight into the deep end by creating a D3 chart from scratch. It involves a lot of code and will probably seem like overkill.

In fact, it is. D3 gives you a very high level of control over your graphics. It takes a long period of study to master. No one can do in a day.

Our goal in this class is to instead quickly introduce you to D3's basic outlines so you can see how it is being used in the field.

First, return to your terminal and use npm to install D3.

```
$ npm install d3@5
```

From here, we'll be working in our `_scripts` folder, where our framework expects us to write JavaScript.

It includes a file called `main.js` which is run every time the page loads. Our framework starts us out with some boilerplate there.

```
// Main javascript entry point
// Should handle bootstrapping/starting application

'use strict';

import 'core-js';
import 'regenerator-runtime/runtime';
import $ from 'jquery';
import { Link } from '../_modules/link/link';

$(() => {
```

(continues on next page)

(continued from previous page)

```

new Link(); // Activate Link modules logic
console.log('Welcome to Yeogurt!');
});

```

Rather than write our code directly in there, we are going to create a separate file for our charts. We will call it `_charts.js` and save it inside of `_scripts/`.

Note: Remember the underscore coding convention we talked about before? Here it is again. We named our new file `_charts.js` with an underscore (`_`) in front because its code will be compiled into `main.js` when the site is baked. Unlike `_charts.js`, `main.js` doesn't have an underscore, because it is the master file that pulls in all the other scripts.

Structuring our code this way helps keep things organized, as each file controls one specific part of the page. Need to make an adjustment to your chart? Go to `_charts.js`.

You can include the libraries we installed (or any JavaScript file!) by using `import`. While with modern versions of D3 you can import only the specific parts of the library needed by your app, we're just going to import the whole library for simplicity.

```

import * as d3 from "d3";

// At the end of the _charts.js file
console.log('hello, this is my charts file!');

```

Now we can use the same `import` method to pull our charts code into `main.js`.

You don't have to use this convention, but it's handy as a visual marker of what files are dependent on others.

```

// Main javascript entry point
// Should handle bootstrapping/starting application

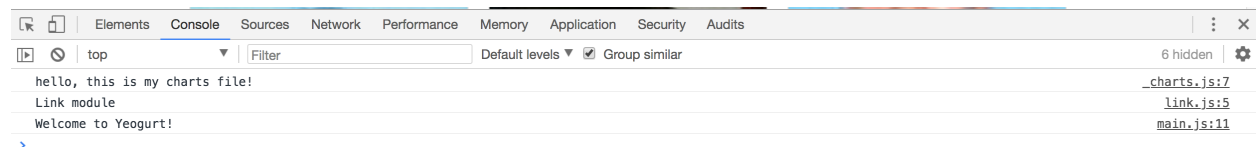
'use strict';

import 'core-js';
import 'regenerator-runtime/runtime';
import $ from 'jquery';
import { Link } from '../_modules/link/link';
import './_charts.js';

$(() => {
  new Link(); // Activate Link modules logic
  console.log('Welcome to Yeogurt!');
});

```

Now if you reload your page and go to your inspector by right clicking on the page and selecting “inspect”, you should see `hello, this is my charts file!` in the console.



Now that we have our tools, it's time for the real work. What chart should we make? The story points out that Harvard Park experienced an increase in homicides at the same time there was a decrease across the rest of the county. Let's try to visualize that.

First, we need somewhere for our charts to live on the page. In our `index.nunjucks` file, inside of `{% block content %}` where you want the chart to appear, create a `div` element with an `id` of `county-homicides`, and another with an `id` of `harvard-park-homicides`.

```
{% extends '_layouts/base.nunjucks' %}

{% block headline %}My headline will go here{% endblock %}
{% block byline %}By me{% endblock %}
{% block pubdate %}
    <time datetime="2018-03-10" pubdate>Mar. 10, 2018</time>
{% endblock %}

{% block content %}

<div class="charts">
    <div class="inline-chart" id="county-homicides"></div>
    <div class="inline-chart" id="harvard-park-homicides"></div>
</div>

<section>
    <h3>Lives lost</h3>
    <p>The {{ site.data.harvard_park_homicides|length }} homicides in Harvard Park,
    ↪since 2000 were primarily black and Latino males, but the list includes husbands,
    ↪wives, fathers, mothers of all ages, and even some small children.</p>
    <div class="card-columns">
        {% for obj in site.data.harvard_park_homicides %}
            <div class="card">
                {% if obj.image %}{% endif %}
                <div class="card-body">
                    <a href="http://homicide.latimes.com/post/{{ obj.slug }}" target="_blank">
                        <strong>
                            <h5 class="card-title">{{ obj.first_name }} {{ obj.last_name }}
                            ↪</h5>
                        </strong>
                    </a>
                    <p class="card-text">{{ obj.last_name }}, a {{ obj.age }}-year-old {{ obj.
                    ↪race }} {{ obj.gender }}, died in {{ obj.death_year }}.</p>
                </div>
            </div>
        {% endfor %}
    </div>
</section>
{% endblock %}
```

That's nice, but we can't make a chart without data. Copy the [annual totals data](#) on GitHub to a new file at `_data/annual_totals.json`. This file contains annual homicide counts for Harvard Park and all of Los Angeles County.

That's nice, but we can't make a chart without data. Copy the [annual totals data](#) on GitHub to a new file at `_data/annual_totals.json`. This file contains annual homicide counts for Harvard Park and all of Los Angeles County.

We can use the same import syntax to include the data in our `_charts.js` file. Since it's a JSON file it will work right away.

```
import * as d3 from "d3";
import annualTotals from "../_data/annual_totals";
```

We want to make two charts - one of county homicides and one of killings in Harvard Park. Let's start with county homicides. D3 requires us to do a bit of house work before we get started. The first thing we need is a container for

our chart to go in. We'll be making these charts in an `<svg>` element, which stands for Scalable Vector Graphic.

The first thing we'll want to do is select the HTML container of the chart with D3, and "append" an `svg` element to it. So, back in `_charts.js`, let's add the following:

```
import * as d3 from "d3";
import annualTotals from "../_data/annual_totals";

// Make sure you use the # here!
var container = d3.select('#county-homicides');
var svg = container.append('svg')
```

Now if you look in your inspector, you'll see that we've appended an `<svg>` to the element with an ID of `county-homicides`. However, we also need to specify a height and width for the SVG, otherwise it will always just have default dimensions of 300x150, no matter how large our screen or device is.



Let's use `.node()` to access the HTML element and save the width and height of the container to variables. I like to specify the height as a percentage of the width, to get an aspect ratio.

```
import * as d3 from "d3";
import annualTotals from "../_data/annual_totals";

// Make sure you use the # here!
var container = d3.select('#county-homicides');
```

(continues on next page)

(continued from previous page)

```
var containerWidth = container.node().offsetWidth;
var containerHeight = containerWidth * 0.66;

var svg = container.append('svg')
```

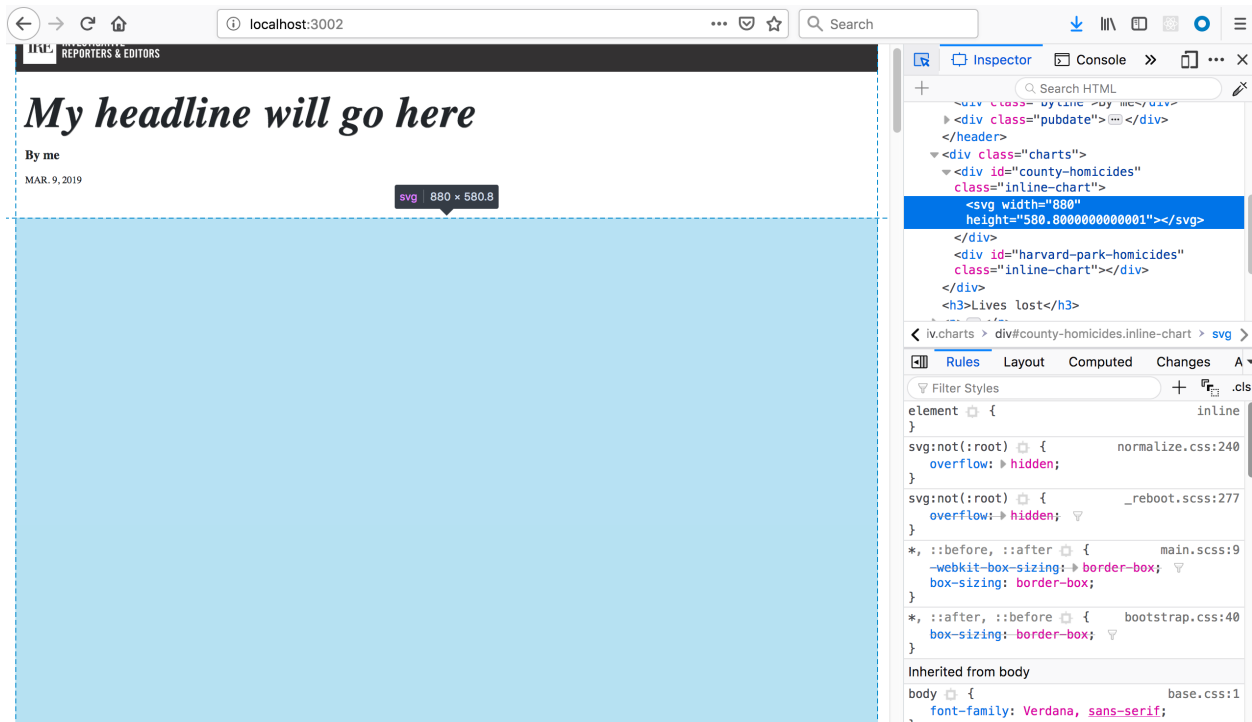
Now we can use them to set the properties, or “attributes” of the SVG using D3’s `.attr()` method. Notice that we can “chain” methods on a selection in D3, which allows our code to be a little more concise.

```
import * as d3 from "d3";
import annualTotals from "../_data/annual_totals";

// Make sure you use the # here!
var container = d3.select('#county-homicides');
var containerWidth = container.node().offsetWidth;
var containerHeight = containerWidth * 0.66;

var svg = container.append('svg')
    .attr('width', containerWidth)
    .attr('height', containerHeight)
```

Now if you look, your SVG should be rendered at the appropriate height and width, filling the available space.



Two more setup steps before we actually start making our charts. First, if we simply start drawing data onto the SVG, we’ll likely see areas where the data clips off the chart. We can avoid this by defining a pre-set margin we’ll use throughout the process.

We also create two variables, `chartWidth` and `chartHeight` that refer to the dimensions of the chart with the margins included.

```
import * as d3 from "d3";
import annualTotals from "../_data/annual_totals";
```

(continues on next page)

(continued from previous page)

```

var margin = {top: 20, right:20, bottom:20, left:40};
// Make sure you use the # here!
var container = d3.select('#county-homicides');
var containerWidth = container.node().offsetWidth;
var containerHeight = containerWidth * 0.66;
var chartWidth = containerWidth - margin.right - margin.left;
var chartHeight = containerHeight - margin.top - margin.bottom;

var svg = container.append('svg')
    .attr('width', containerWidth)
    .attr('height', containerHeight)

```

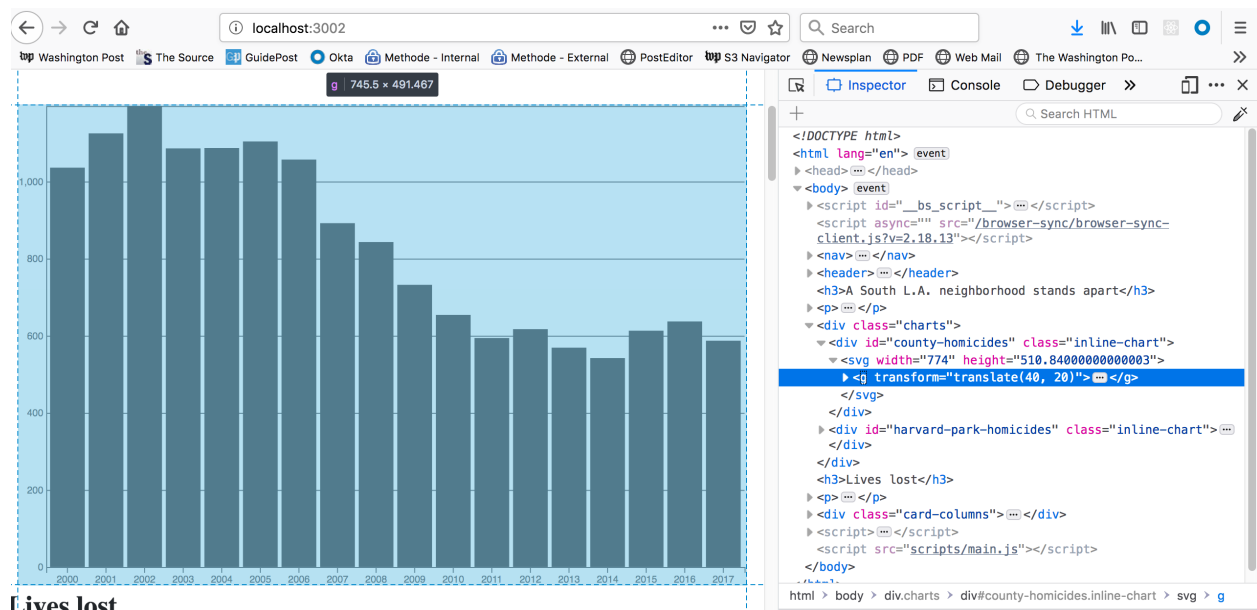
Second, we should add a `<g>`, or “group” tag, where everything else in our chart will go. Add this to the end of our `svg` declaration. We’ll also want to give it a `transform` attribute that shifts it slightly according to our margins.

```

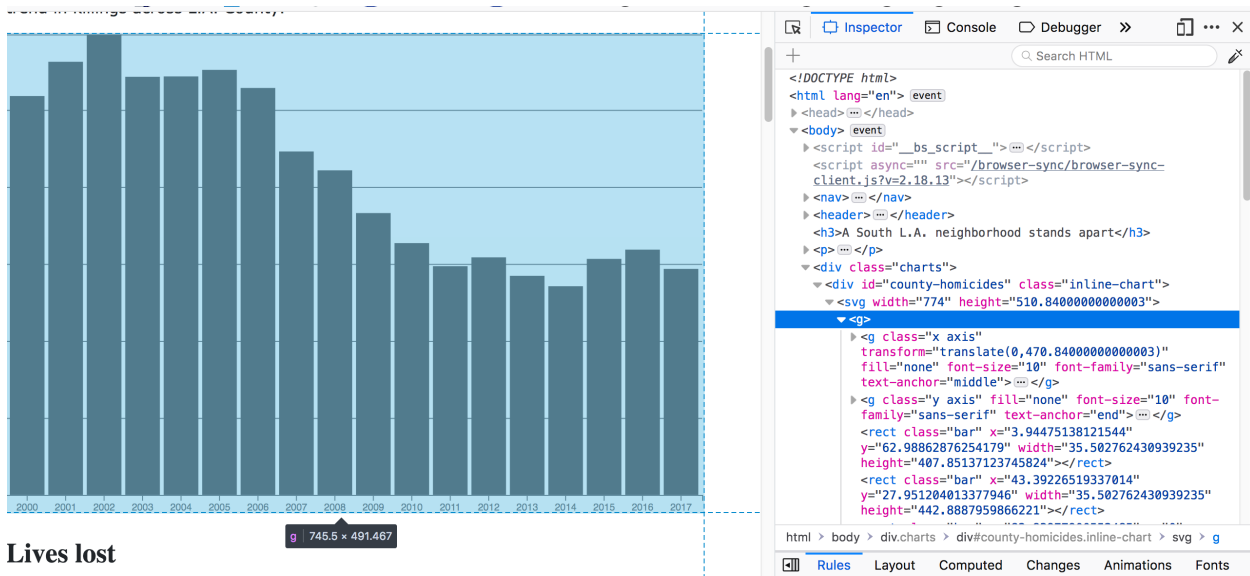
// ... more code is up here
var svg = container.append('svg')
    .attr('width', chartWidth)
    .attr('height', chartHeight)
    .append('g')
    .attr('transform', `translate(${margin.left}, ${margin.top})`)

```

Adding the `g` tag and shifting it may seem like a weird step, but it’s an important step to take to make sure the value labels aren’t going to clip off the edges of our charts. To show what this does, this example skips a few steps ahead so you can see elements inside the `g` tag shifted by the margins of the chart.



And here’s what it looks like without the margins, see how the labels are clipped?



At this point, we're ready to start drawing our chart. Let's start by creating the "scales" for our data. D3 manages its data by mapping input values from the data, also known as the domain, into output values on the screen, or the range. This creates a scale that transforms the input into the output.

D3 has many different types of scales, for linear, categorical and time-based data. In this case, we'll want a linear scale for the Y axis, and a "band" scale, which is a type of categorical scale useful for bar charts, for the X axis.

I like to calculate the input, or domain, before creating the axes. The domain takes the form of an array with the minimum and maximum value that you want to map: e.g., `[0, 100]` if you're looking at a 100-point grade scale. We can use D3's `min` and `max` helper functions to find this.

If you look at the data in `src/_data/annual_totals.json`, you'll see that each year's data is organized like this:

```
{
  "year":2000,
  "homicides_total":1036,
  "homicides_harvard_park":3
}
```

Since we're charting homicides for the entire county we want the `homicides_total` attribute in our data for the Y axis, and the X axis will be the year. The arrow `=>` is a shorthand method of accessing the `homicides_total` attribute of each object in the `annualTotals` array.

For the X axis, all we want is an array of the years, e.g.: `[2000, 2001, ...]` so we can call `.map()` on our data to return the year value. `.map()` iterates over every value in an array and returns a value for every item.

```
// The rest of your code is up here

var xDomain = annualTotals.map(d => d.year);
```

For the Y-axis, we want the domain to start at 0, so we can set that manually. We can use D3's `max` method to get the largest value in the dataset.

```
// The rest of your code is up here

var yDomain = [0, d3.max(annualTotals.map(d => d.homicides_total))];
```

If you know the min and max values, you can also set these manually, which can be useful if you want your chart max to be a nice even number.

At the bottom of your file, let's create an `xScale` and `yScale` now. Note that at this point we're also setting the range, or output values, to the range between 0 and the height and width of our SVG.

```
// The rest of your code is up here

var xScale = d3.scaleBand()
  .domain(xDomain)
  .range([0, chartWidth])
  .padding(0.1);

var yScale = d3.scaleLinear()
  .domain(yDomain)
  .range([chartHeight, 0]);
```

Note that the X scale has an additional method, `.padding()`, which specifies how far apart our bars are from one another.

Now that we have scales, we can create our axes. D3 has helper functions for each side of the chart we want our axes on, in this case the left for the Y-axis and bottom for the X-axis. We also assign one of the scales we just created to each axis.

For the Y-axis, we also want to add grid lines and limit the number of ticks that are shown.

```
// The rest of your code is up here

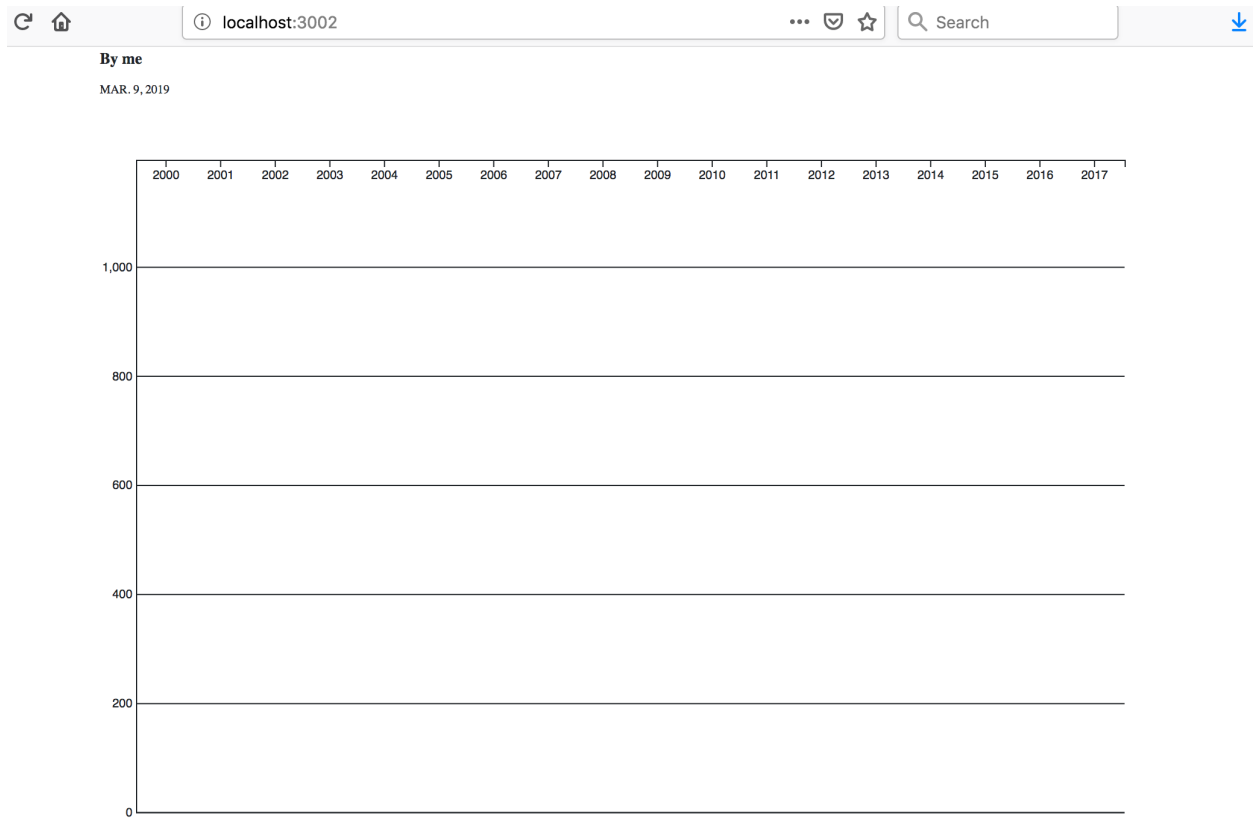
var xAxis = d3.axisBottom(xScale);
var yAxis = d3.axisLeft(yScale)
  .tickSize(-chartWidth)
  .ticks(4);
```

Finally, we append those to the chart by appending a `<g>` tag and “calling” the axis function we just created. I like to give each axis element a class of “axis” and “x” or “y”, depending on which axis we’re creating.

```
// The rest of your code is up here

svg.append("g")
  .attr("class", "x axis")
  .call(xAxis);

svg.append("g")
  .attr("class", "y axis")
  .call(yAxis);
```

Well that doesn't look quite right. The reason the X axis is displaying at the top of the chart is that in SVGs, the coordinate 0,0 is at the top left. So we need to shift, or `translate` the X axis down by the height of the chart. The Y axis is fine where it is.

Replace the code for the X axis with the below.

```
// The rest of your code is up here

svg.append("g")
  .attr("class", "x axis")
  .attr("transform", `translate(0, ${chartHeight})`)
  .call(xAxis);
```



Now that the axes are there, we're finally ready to draw our bars. D3 handles its data by binding the records to the SVG elements - hence the name: "Data Driven Documents."

The format seems a little strange at first, because you're selecting elements, then binding data to the selection, then creating elements that are bound to the data. You do this by chaining two methods, `.data()`, which determines the data set that you're binding, and `.enter()`, which iterates over the data set.

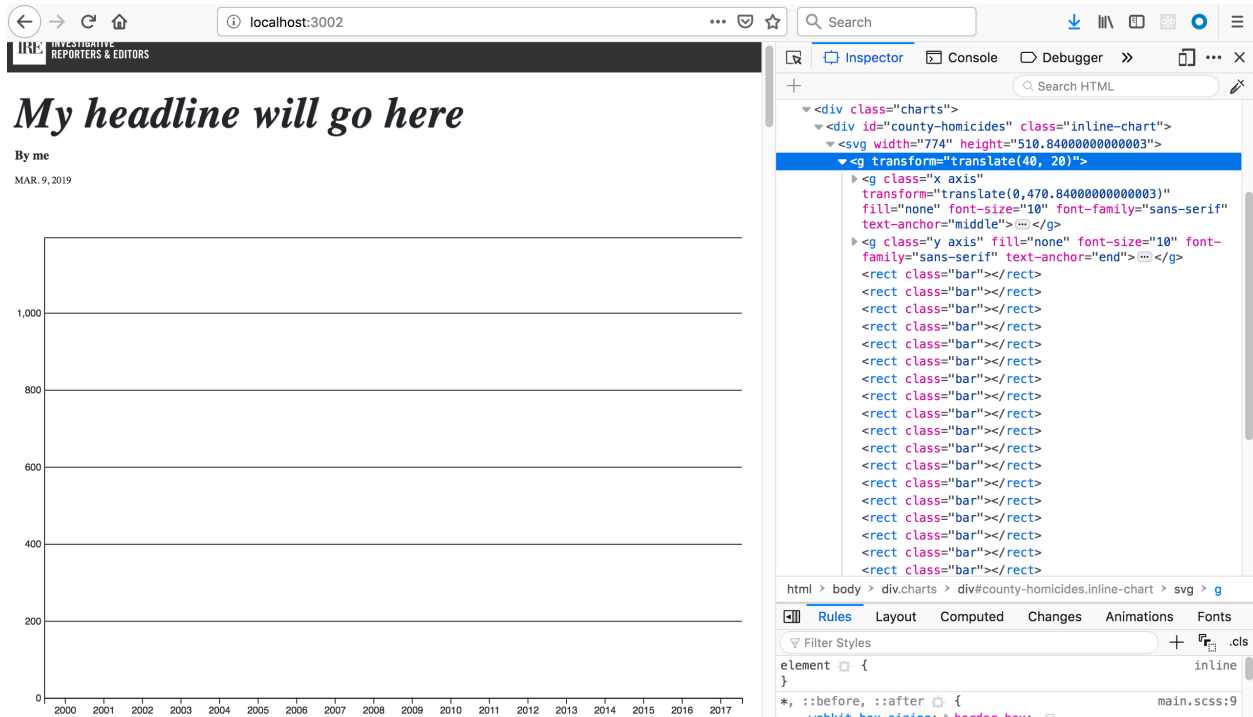
Since we're making a bar chart, we're going to create a `<rect>` element, and give it a class of `bar`.

Note: If you'd like to know more about how D3 data binding works, Scott Murray has an [excellent explanation and tutorial](#) on his website.

Let's give it a try, by binding our `annualTotals` data to the bars on the chart. Start below the code for your axes. First, let's simply append the `<rect>` elements to the chart

```
// The rest of your code is up here
```

```
svg.append('g')
  .attr('class', 'bars')
  .selectAll('.bar')
  .data(annualTotals)
  .enter()
  .append('rect')
  .attr('class', 'bar')
```

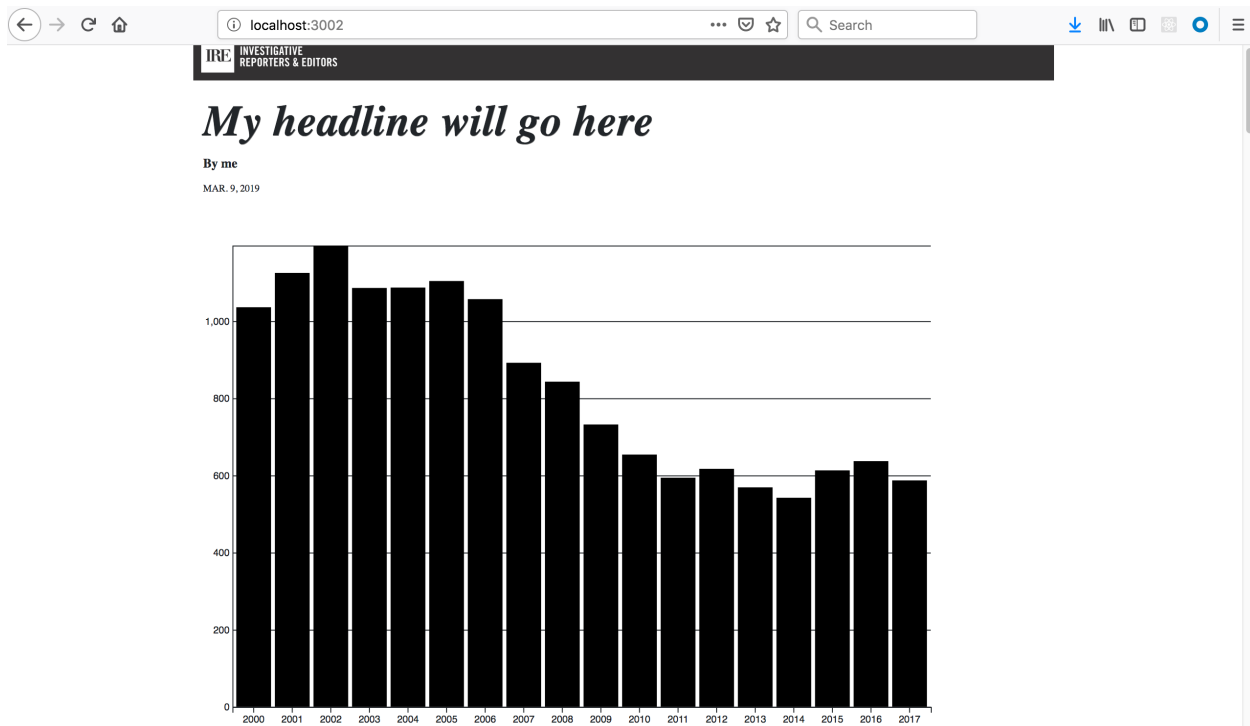


Now if you look at your chart... nothing has changed! But open your inspector and look at your SVG - you'll see lots of `<rect>` elements, you just can't see them because they don't have any values for height and width, or x and y position values. Let's do this next.

```
// The rest of your code is up here

svg.append('g')
  .attr('class', 'bars')
  .selectAll('.bar')
  .data(annualTotals)
  .enter()
  .append('rect')
  .attr('class', 'bar')
  .attr('x', d => xScale(d.year))
  .attr('y', d => yScale(d.homicides_total))
  .attr('width', xScale.bandwidth())
  .attr('height', d => chartHeight - yScale(d.homicides_total))
```

The X value will be determined by the year, and the Y by the `homicides_total` value of each object. The width of each bar is set by a method called `.bandwidth()` on our scale, and the height will be scaled corresponding to the number of homicides.



You have a bar chart! At this point we can step back and style out the chart, and leave room for a second chart that shows Harvard Park homicides.

At this point, create a new file in the `_styles` folder, and call it `_charts.scss`.

The first thing we need to do is make the chart smaller - right now it's huge! Add the following CSS rule to `_charts.scss` which will allow the chart to display at roughly half width and leave room for a second chart.

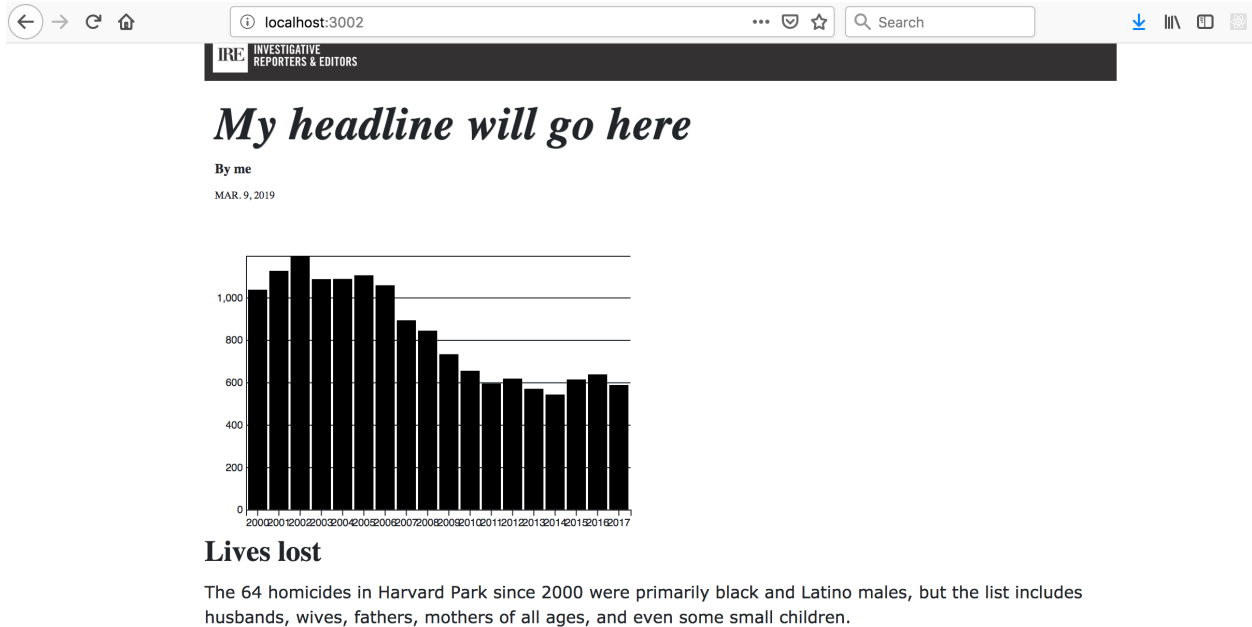
```
.inline-chart {
  width: 49%;
  display: inline-block;
}
```

If you look at the page now, you'll see that nothing has changed. That's because, like the JavaScript, we need to import the styles that we just created into our `main.scss` file.

You can do that by adding the following line to `main.scss`.

```
// Normalize Styles
@import 'node_modules/normalize.css/normalize';

// Import Modules
@import '../_modules/link/link';
@import '../_charts.scss';
```



Let's also color the bars and clean up some of the lines. If you remember, the bars were `<rect>` elements, and if you use the inspector, you can find the x axis lines we want to remove. Back in `_charts.scss`:

```
rect {
  fill: #86C7DF;
}

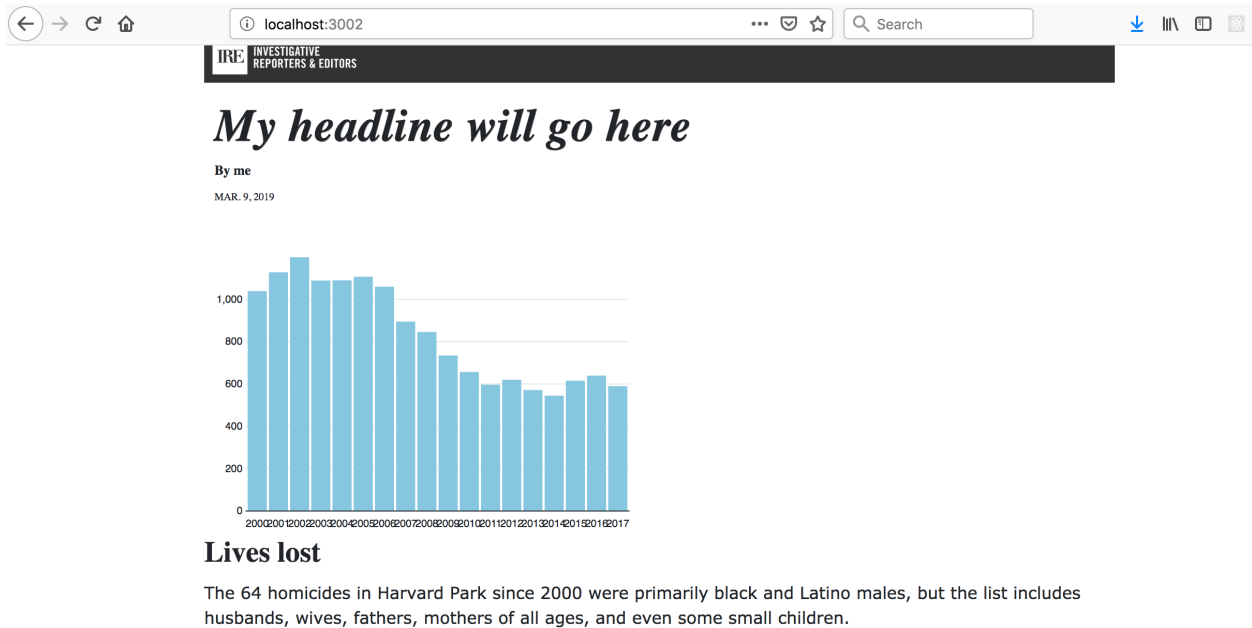
.y .domain {
  display: none;
}

.x .domain {
  display: none;
}

.x .tick line {
  display: none;
}
```

The last thing we want to style is the grid lines - they're too heavy and should fade into the background more. Note that we want to keep the baseline black to indicate that we're starting at 0, so we'll use a fancy CSS selector that says to style every tick line that's not the baseline.

```
// The rest of your styles are up here
.y .tick:not(:first-of-type) line {
  stroke: #e7e7e7;
}
```



Now we have a nicely styled chart, and we're ready to start on our second one. Do we want to copy everything all over again? No! Instead, we can pull the JavaScript we just wrote into a function that will take our data and an element, and create the chart for us!

Open up `_charts.js` again, and create a function, `createChart`. We'll need to think about this for a second - what are the values that are going to change between the two charts?

- The container element
- The data field used for the homicide counts
- The y-axis values

If we calculate the domain values correctly the Y-axis values should automatically update so we shouldn't have to worry about that too much. So our function should have two arguments - the ID of the container element, and the data field we're using.

```
// the rest of your code is up here
function createChart(el, fieldname) {

}
```

Now, you can copy everything we wrote in `_charts.js` under the `import` line into this function. Your file should look like this now.

```
import * as d3 from "d3";
import annualTotals from "../_data/annual_totals";

var createChart = (el, fieldname) => {
  var margin = {top: 20, right:20, bottom:20, left:40} ;
  var container = d3.select('#county-homicides');
  var containerWidth = container.node().offsetWidth;
  var containerHeight = containerWidth * 0.66;
  var chartWidth = containerWidth - margin.right - margin.left;
  var chartHeight = containerHeight - margin.top - margin.bottom;

  var svg = container.append('svg')
```

(continues on next page)

(continued from previous page)

```

    .attr('width', containerWidth)
    .attr('height', containerHeight)
    .append('g')
      .attr('transform', `translate(${margin.left}, ${margin.top})`)

    var xDomain = annualTotals.map(d => d.year);

    var yDomain = [0, d3.max(annualTotals.map(d => d.homicides_total))];

    var xScale = d3.scaleBand()
      .domain(xDomain)
      .range([0, chartWidth])
      .padding(0.1);

    var yScale = d3.scaleLinear()
      .domain(yDomain)
      .range([chartHeight, 0]);

    var xAxis = d3.axisBottom(xScale);
    var yAxis = d3.axisLeft(yScale)
      .tickSize(-chartWidth)
      .ticks(4);

    svg.append("g")
      .attr("class", "x axis")
      .attr("transform", `translate(0, ${chartHeight})`)
      .call(xAxis);

    svg.append("g")
      .attr("class", "y axis")
      .call(yAxis);

    svg.append('g')
      .attr('class', 'bars')
      .selectAll('.bar')
      .data(annualTotals)
      .enter()
      .append('rect')
      .attr('class', 'bar')
      .attr('x', d => xScale(d.year))
      .attr('y', d => yScale(d.homicides_total))
      .attr('width', xScale.bandwidth())
      .attr('height', d => chartHeight - yScale(d.homicides_total));
  }

```

Now, if you reload your page, your chart will have disappeared! That's because our code is packed away inside a function, which won't run until we call on it.

At the end of the file, let's return to `_charts.js` and call the function with the arguments necessary for the countywide homicides chart. Remember the element id is `county-homicides`, and the field we're using is `homicides_total`.

```

// the rest of your code is up here
createChart("#county-homicides", "homicides_total")

```

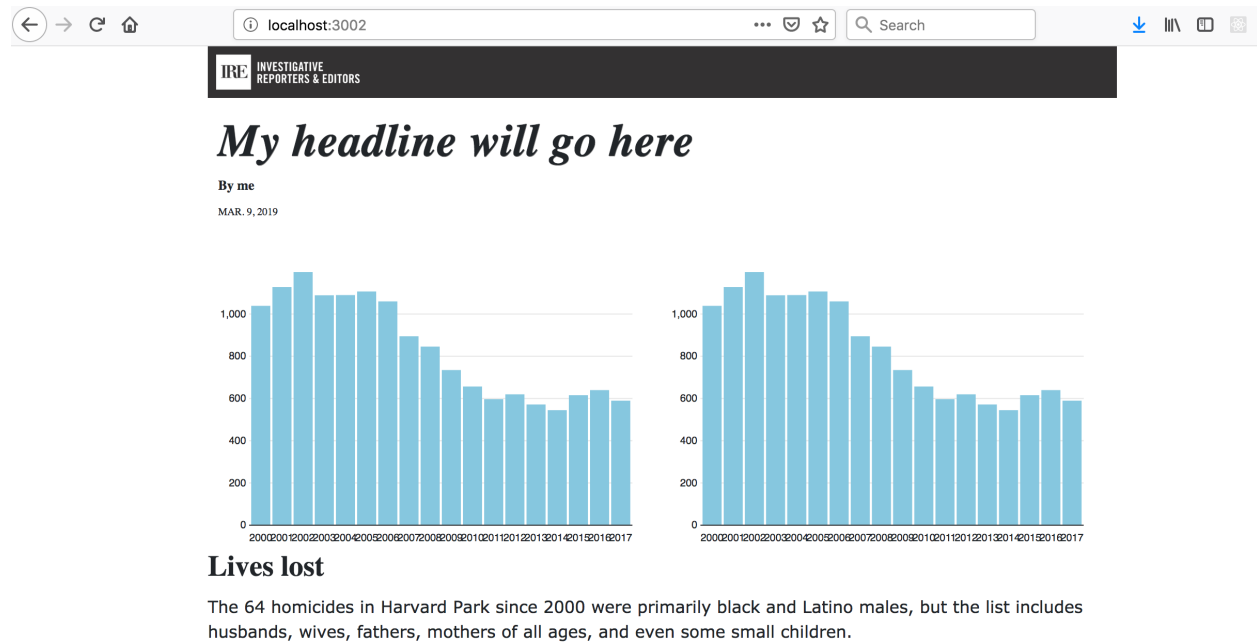
You'll see that your chart is back! But the only reason this is actually working is because we've already hard-coded our variables into the script. Let's abstract it out to use the arguments that we're providing.

First, let's change the container variable to use the ID we're providing.

```
var createChart = (el, fieldname) => {  
  var margin = {top: 20, right:20, bottom:20, left:40} ;  
  var container = d3.select(el);  
  
  //... the function continues down here  
}
```

Now try calling it on the second element we created, with the `homicides_harvard_park` variable as the second argument.

```
createChart("#county-homicides", "homicides_total")  
createChart("#harvard-park-homicides", "homicides_harvard_park")
```



This gives us the same chart twice, which is expected since we still have the data values hard-coded.

To change this, we'll have to find every instance where we reference the `homicides_total` field directly in the function, and change it to reference the argument we are passing in for the data field.

Note that in many cases we'll have to change the syntax from `d.homicides_total` to `d[fieldname]` - this is because we're referencing a variable and not a specific field.

Luckily, we only have to do this a few times, once where we're calculating the domain, and then where we're setting the y position and heights of the bars.

```
var createChart = (el, fieldname) => {  
  var margin = {top: 20, right:20, bottom:20, left:40};  
  var container = d3.select(el);  
  var containerWidth = container.node().offsetWidth;  
  var containerHeight = containerWidth * 0.66;  
  var chartWidth = containerWidth - margin.right - margin.left;  
  var chartHeight = containerHeight - margin.top - margin.bottom;  
  
  var svg = container.append('svg')  
    .attr('width', containerWidth)
```

(continues on next page)

(continued from previous page)

```
.attr('height', containerHeight)
.append('g')
  .attr('transform', `translate(${margin.left}, ${margin.top})`)

var xDomain = annualTotals.map(d => d.year);

var yDomain = [0, d3.max(annualTotals.map(d => d[fieldname]))];

var xScale = d3.scaleBand()
  .domain(xDomain)
  .range([0, chartWidth])
  .padding(0.1);

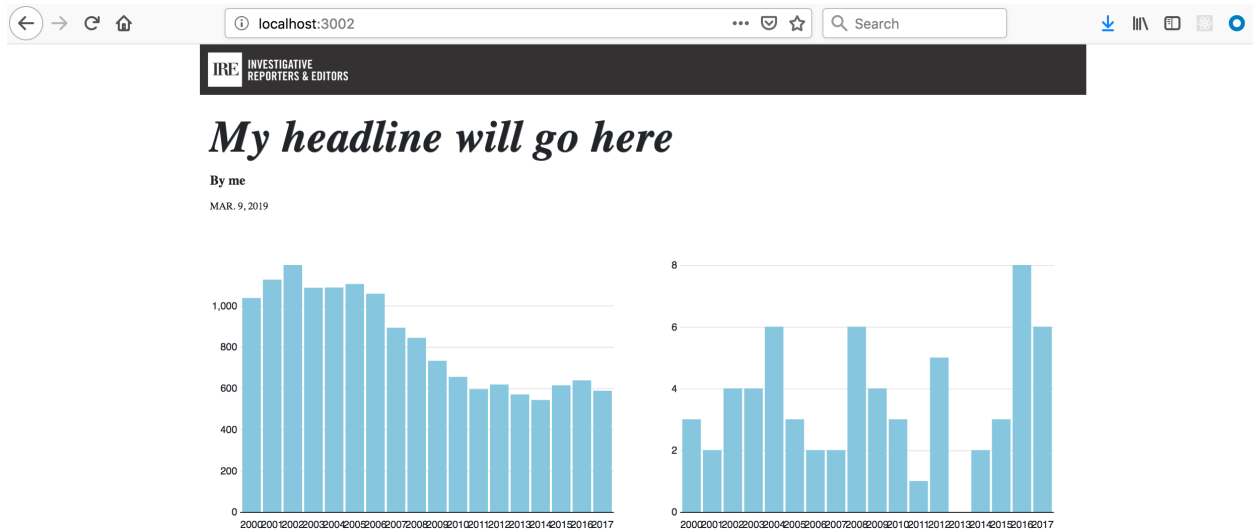
var yScale = d3.scaleLinear()
  .domain(yDomain)
  .range([chartHeight, 0]);

var xAxis = d3.axisBottom(xScale);
var yAxis = d3.axisLeft(yScale)
  .tickSize(-chartWidth)
  .ticks(4);

svg.append("g")
  .attr("class", "x axis")
  .attr("transform", `translate(0, ${chartHeight})`)
  .call(xAxis);

svg.append("g")
  .attr("class", "y axis")
  .call(yAxis);

svg.append('g')
  .attr('class', 'bars')
  .selectAll('.bar')
  .data(annualTotals)
  .enter()
  .append('rect')
  .attr('class', 'bar')
  .attr('x', d => xScale(d.year))
  .attr('y', d => yScale(d[fieldname]))
  .attr('width', xScale.bandwidth())
  .attr('height', d => chartHeight - yScale(d[fieldname]));
}
```



Lives lost

The 64 homicides in Harvard Park since 2000 were primarily black and Latino males, but the list includes husbands, wives, fathers, mothers of all ages, and even some small children.

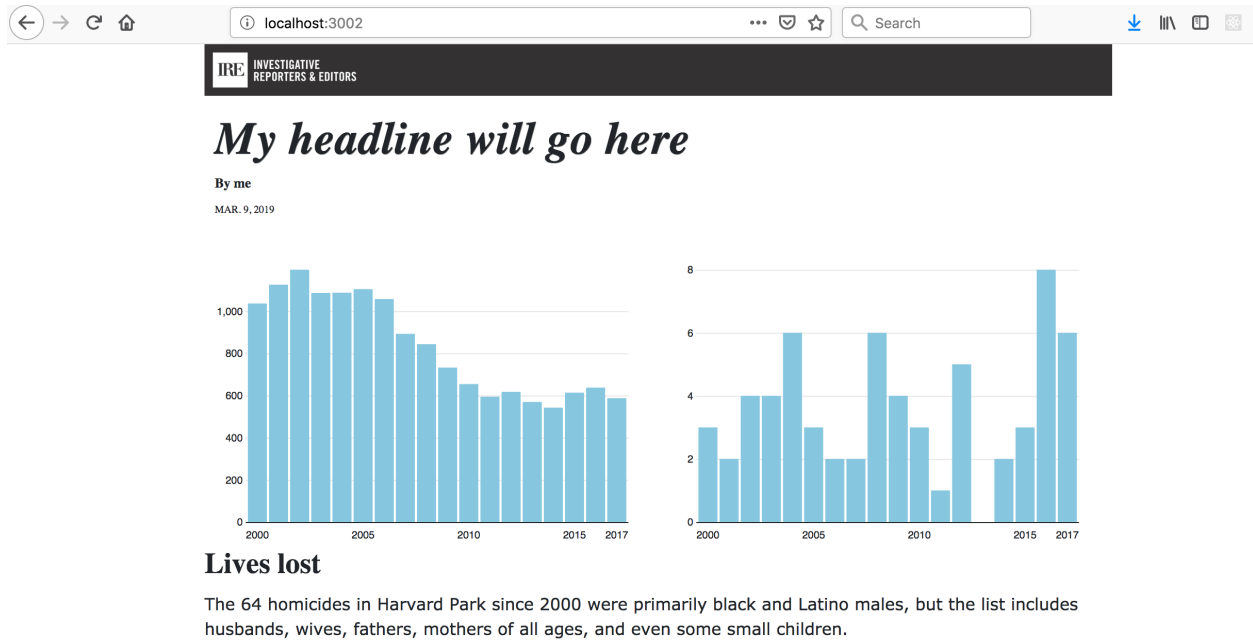
Now that our charts are smaller and they're right next to each other, we need to clean up those year labels. Since our years are the same in both charts, we can set this manually when we're creating the X axis.

Let's update the `xAxis` variable in `createCharts` to label the first and last bars on the chart, and the 5-year intervals.

```
// ... more code is up here
var xAxis = d3.axisBottom(xScale)
    .tickValues([2000, 2005, 2010, 2015, 2017]);

var yAxis = d3.axisLeft(yScale)
    .tickSize(-chartWidth)
    .ticks(4);

// ... more code is down here
```

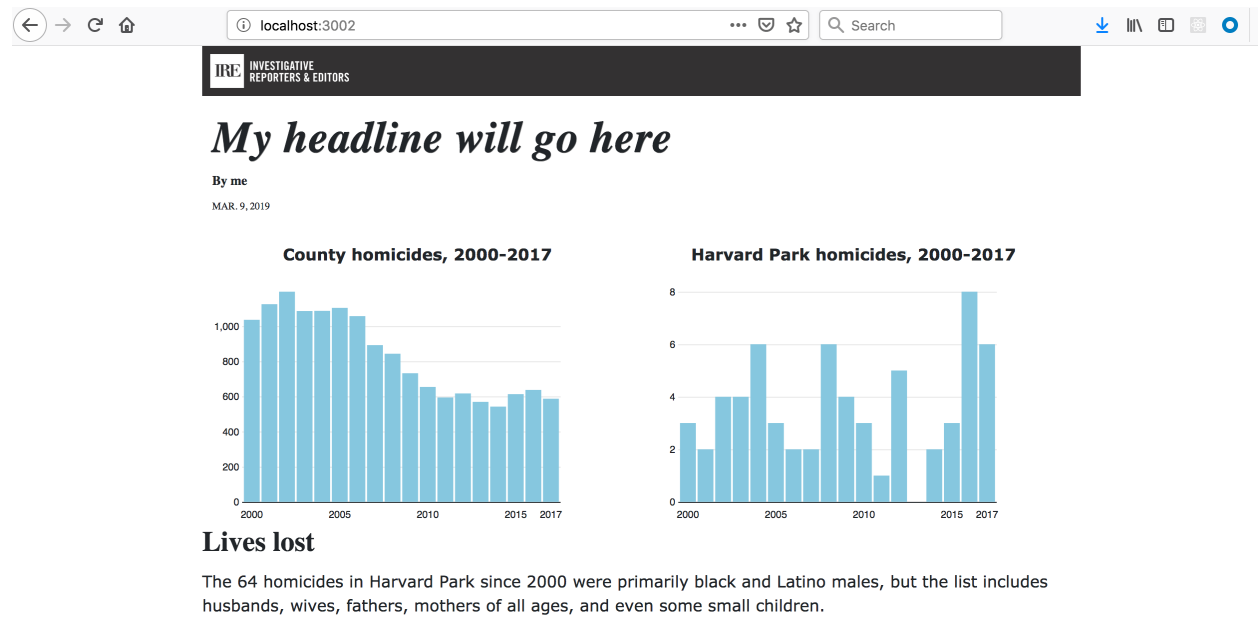


This cleans things up a lot! We have some pretty good-looking charts. Our charts need titles, which we can add directly to the HTML. Going back to the `index.nunjucks` file, add the titles in `<h4>` tags inside your chart containers

```
<div class="charts">
  <div class="inline-chart" id="county-homicides">
    <h4 class="chart-title">County homicides, 2000-2017</h4>
  </div>
  <div class="inline-chart" id="harvard-park-homicides">
    <h4 class="chart-title">Harvard Park homicides, 2000-2017</h4>
  </div>
</div>
```

Let's style those a bit too, add these rules to the bottom of `_styles/_charts.scss`.

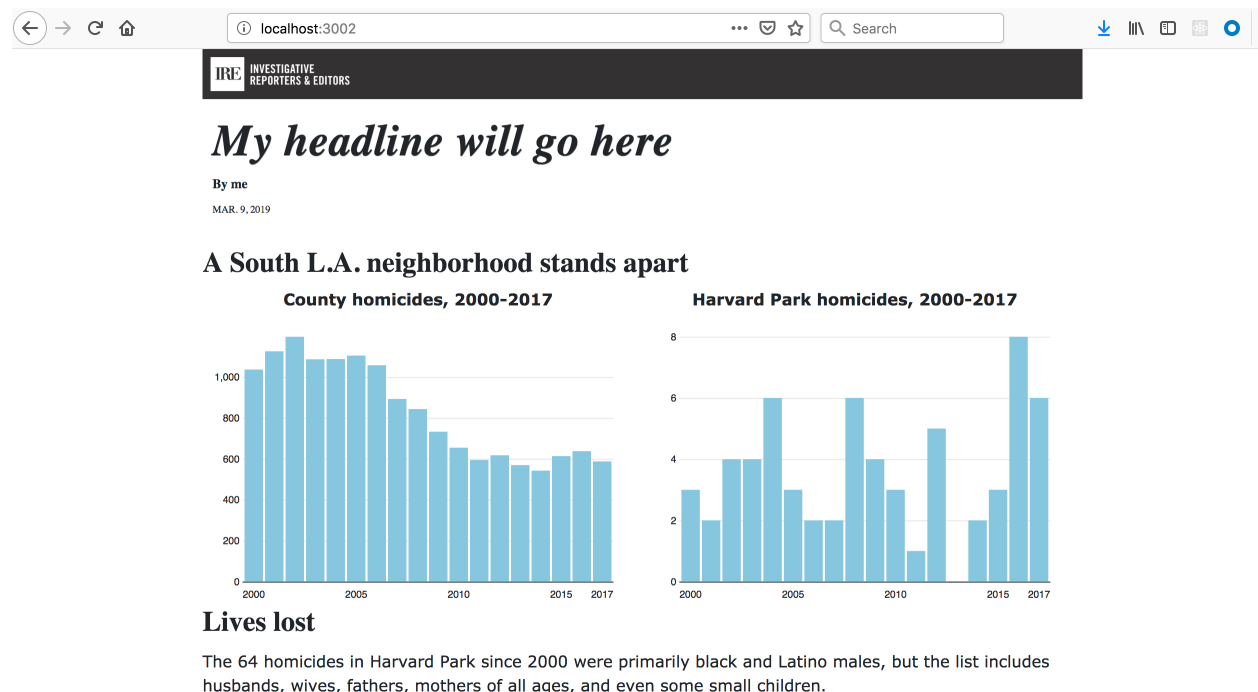
```
.chart-title {
  font-weight: bold;
  font-size: 16px;
  text-align: center;
}
```



Last, let's add a headline to introduce our charts section.

```
<h3>A South L.A. neighborhood stands apart</h3>

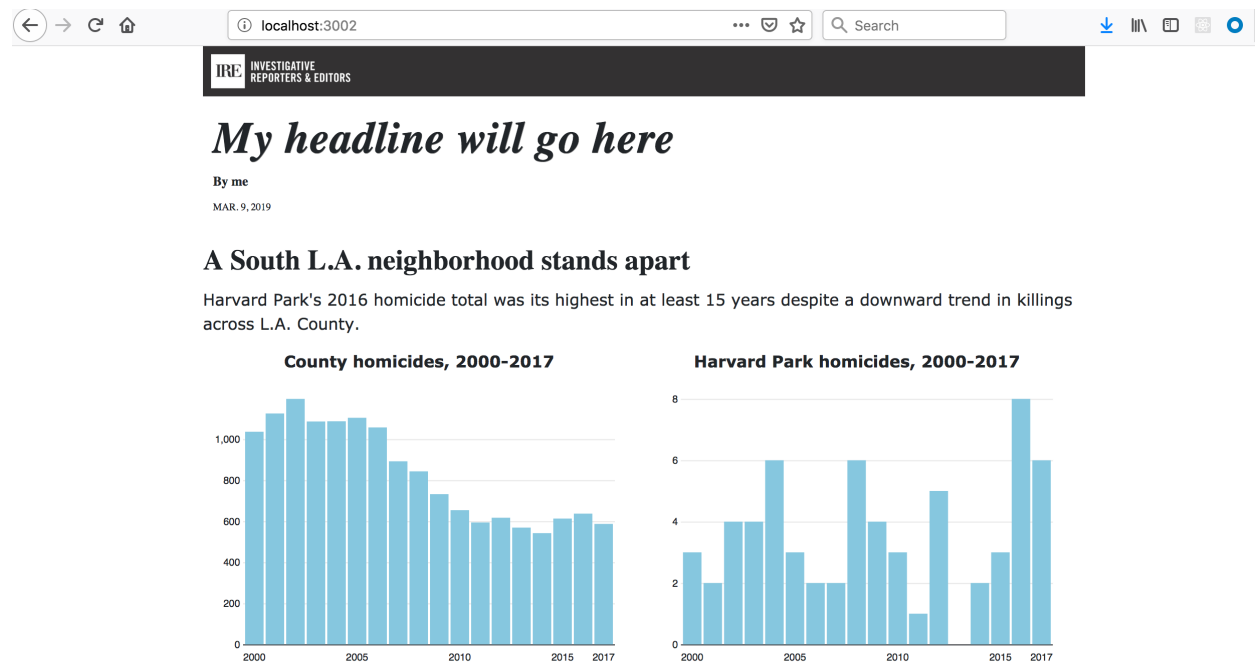
<div class="charts">
  <div class="inline-chart" id="county-homicides">
    <h4 class="chart-title">County homicides, 2000-2017</h4>
  </div>
  <div class="inline-chart" id="harvard-park-homicides">
    <h4 class="chart-title">Harvard Park homicides, 2000-2017</h4>
  </div>
</div>
```



And an introductory paragraph to say a little bit about what we're looking at.

```
<h3>A South L.A. neighborhood stands apart</h3>
<p>Harvard Park's 2016 homicide total was its highest in at least 15 years despite a
↳downward trend in killings across L.A. County.</p>

<div class="charts-holder clearfix">
  <div class="inline-chart" id="county-homicides"></div>
  <div class="inline-chart" id="harvard-park-homicides"></div>
</div>
```



Lives lost

The 64 homicides in Harvard Park since 2000 were primarily black and Latino males, but the list includes

Last, let's wrap our charts HTML in `<section>` tags to keep things orderly.

```
<section>
  <h3>A South L.A. neighborhood stands apart</h3>
  <p>Harvard Park's 2016 homicide total was its highest in at least 15 years,
↳despite a downward trend in killings across L.A. County.</p>

  <div class="charts-holder clearfix">
    <div class="inline-chart" id="county-homicides"></div>
    <div class="inline-chart" id="harvard-park-homicides"></div>
  </div>
</section>
```

Congratulations, you've made your charts! Let's commit our changes and move on to our next challenge.

```
$ git commit -m "Made my first charts."
$ git push origin master
```

Note: We used D3.js in this class, but there are many other JavaScript charting libraries, each one slightly different. If you want to explore this on your own, here are some other options that generally abstract away the process we used in this class.

- [Vega-lite](#)
- [Charts.js](#) Looks really awesome and abstracts a lot of the pain points of D3 away, but as it only draws to `<canvas>` and we wanted to be able to individually inspect SVG elements, we didn't use it for this class.
- [C3.js](#) Important to note that this does not seem to support the latest versions of D3.

There are also tools that allow you to use a visual editor, creating charts and other visualizations that you can download and/or embed in your project.

- [Observable](#) is a relatively new site that allows you to take a more exploratory approach to building your visualizations. Charts and maps update automatically as you update data or settings.
 - [Chartbuilder](#) from [Quartz](#), is very good for basic, fast charts with light customization.
 - [DataWrapper](#) allows a range of visualizations beyond basic charts, including scatter plots and maps.
-

Extra credit - interactivity!

Now let's try and make these charts interactive. We want to highlight a bar and display its value whenever a user hovers over it. To do this, we're going to use D3's "event binding."

In our `createChart()` function, we'll want to add a new method, `.on()` to the code snippet where we create our bars.

For now, let's log the value to our console.

```
svg.append('g')
  .attr('class', 'bars')
  .selectAll('.bar')
  .data(annualTotals)
  .enter()
  .append('rect')
  .attr('class', 'bar')
  .attr('x', d => xScale(d.year))
  .attr('y', d => yScale(d[fieldname]))
  .attr('width', d => xScale.bandwidth())
  .attr('height', d => chartHeight - yScale(d[fieldname]))
  .on('mouseenter', d => {
    console.log(d[fieldname])
  });
```

Now if you look in your console, you should see the values for each bar being logged when you mouse over.

Now let's use this change each bar's color, and the `mouseleave` event to remove that highlight when the mouse exit.

```
svg.append('g')
  .attr('class', 'bars')
  .selectAll('.bar')
  .data(annualTotals)
  .enter()
  .append('rect')
  .attr('class', 'bar')
  .attr('x', d => xScale(d.year))
  .attr('y', d => yScale(d[fieldname]))
  .attr('width', d => xScale.bandwidth())
  .attr('height', d => chartHeight - yScale(d[fieldname]))
  .on('mouseenter', function(d) {
    d3.select(this).classed('highlight', true);
  })
```

(continues on next page)

(continued from previous page)

```
.on('mouseleave', function(d) {
  d3.select(this).classed('highlight', false);
});
```

And add the rule for `.highlight` to the CSS.

```
.highlight {
  fill: #2AB2E4;
}
```

We have interactivity!

Now let's add a tooltip. First, in the `createCharts` function, add a line that appends a `<text>` element to the SVG. Place this under the lines where you append your axes to the SVG, but before you add the bars.

```
svg.append("g")
  .attr("class", "y axis")
  .call(yAxis);

var tooltip = svg.append('text')
  .attr('class', 'chart-tooltip');
```

Now in `_charts.js`, let's go back to our `.on()` statement and try filling out the text element with the proper value, and positioning it. Let's also clear the div when the mouse leaves.

```
svg.append('g')
  .attr('class', 'bars')
  .selectAll('.bar')
  .data(annualTotals)
  .enter()
  .append('rect')
  .attr('class', 'bar')
  .attr('x', d => xScale(d.year))
  .attr('y', d => yScale(d[fieldname]))
  .attr('width', d => xScale.bandwidth())
  .attr('height', d => chartHeight - yScale(d[fieldname]))
  .on('mouseenter', function(d) {
    d3.select(this).classed('highlight', true);

    // centers the text above each bar
    var x = xScale(d.year) + xScale.bandwidth() / 2;
    // the - 5 bumps up the text a bit so it's not directly over the bar
    var y = yScale(d[fieldname]) - 5;

    tooltip.text(d[fieldname])
      .attr('transform', `translate(${x}, ${y})`)
  })
  .on('mouseleave', function(d) {
    d3.select(this).classed('highlight', false);
    tooltip.text('');
  });
```

Now back in our CSS, we can style this out a bit.

```
.chart-tooltip {  
  font-family: Helvetica;  
  font-size: 12px;  
  text-anchor: middle;  
}
```

You have an interactive chart!

CHAPTER 10

Chapter 7: Hello map

Next we'll move on to creating a map focused on West 62nd Street and Harvard Boulevard, an intersection in South Los Angeles where four men died in less than a year and a half.

To draw the map we will rely on [Leaflet](#), a JavaScript library for creating interactive maps. We will install it just as before by using `npm` from our terminal.

```
$ npm install leaflet@1.6.0
```

Next we import Leaflet's stylesheets in `_styles/main.scss` so that they are also included on our site.

```
// Normalize Styles
@import 'node_modules/normalize.css/normalize';

// Import Modules
@import '../_modules/link/link';
@import '_charts.scss';
@import 'node_modules/leaflet/dist/leaflet';
```

Now, back in the `index.nunjucks` template, we should create a placeholder in the page template where the map will live. Let's set it right above the charts section we've just finished.

```
<div id="map"></div>

<section>
  <h3>A South L.A. neighborhood stands apart</h3>
  <p>Harvard Park's 2016 homicide total was its highest in at least 15 years,
  ↳ despite a downward trend in killings across L.A. County.</p>

  <div class="charts-holder clearfix">
    <div class="inline-chart" id="county-homicides"></div>
    <div class="inline-chart" id="harvard-park-homicides"></div>
  </div>
</section>
```

To bring the map to life, add a new file named `_map.js` to the `_scripts` directory. Import it in `main.js`.

```
// Main javascript entry point
// Should handle bootstrapping/starting application
'use strict';

import "core-js";
import "regenerator-runtime/runtime";
import $ from "jquery";
import { Link } from "../_modules/link/link";
import "../_charts";
import "../_map";

$(function() {
  new Link(); // Activate Link modules logic
  console.log('Welcome to Yeogurt!');
});
```

We should then import Leaflet into `_scripts/map.js` so that its tools are available in this file.

```
import * as L from "leaflet";
```

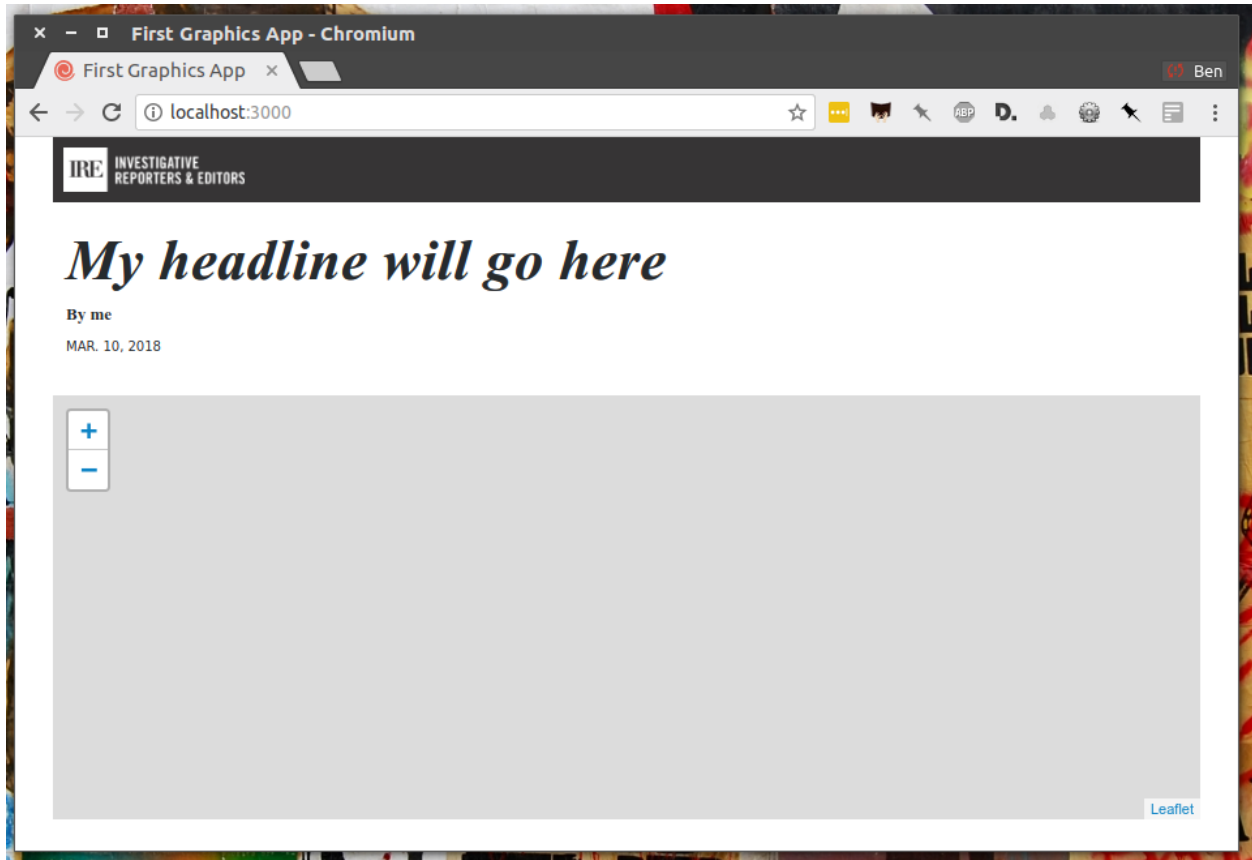
Now in `_scripts/_map.js` paste in the following Leaflet code to generate a simple map. It does three things:

- create a new map in the HTML element we made with “map” set as its ID;
- add a new map layer with roads, borders, water and other features from OpenStreetMap;
- finally, add the layer to the map.

```
import * as L from "leaflet";

var map = L.map('map');
var sat = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/satellite-streets-v9/
↪tiles/{z}/{x}/{y}?access_token=pk.
↪eyJlIjoibGF0aW1lcYIsImEiOiJjanJmNjg4ZzYweGtvNDNxa2ZpZ2lma3Z4In0.
↪g0lYVIEs9Y5QcUOhXactHA');
sat.addTo(map);
```

After you save, the index page should reload with a blank map.



To zero in on the area we're reporting on, we will need its longitude and latitude coordinates. Go to Google Maps and find 62nd Street and Harvard Boulevard in South LA. Hold down a click until it gives you the coordinates in a popup box. Paste those numbers into Leaflet's `setView` method with a zoom level of 15 included.

```
var map = L.map('map')
var sat = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/satellite-streets-v9/
↪tiles/{z}/{x}/{y}?access_token=pk.
↪eyJlIjoibGF0aW1lcYIsImEiOiJjanJmNjg4ZzYweGtvNDNxa2ZpZ2lma3Z4In0.
↪g0lYVIEs9Y5QcUOhXactHA');
sat.addTo(map);
map.setView([33.983265, -118.306799], 15);
```

After you save the file, your map should have relocated. Let's tighten up that zoom and save again.

```
var map = L.map('map')
var sat = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/satellite-streets-v9/
↪tiles/{z}/{x}/{y}?access_token=pk.
↪eyJlIjoibGF0aW1lcYIsImEiOiJjanJmNjg4ZzYweGtvNDNxa2ZpZ2lma3Z4In0.
↪g0lYVIEs9Y5QcUOhXactHA');
sat.addTo(map);
map.setView([33.983265, -118.306799], 18);
```

Now let's load some data on the map. We will return to the list of all homicides already stored in `_data/harvard_park_homicides.json`.

Import the homicides data into `_map.js`.

```
import * as L from "leaflet";
import homicides from "../_data/harvard_park_homicides.json";
```

If you look in the `_data/harvard_park_homicides.json` file, you'll see that every record has a latitude and longitude associated with it. We can use these to place each homicide as a point on the map.

```
{
  "case_number": "2017-04514",
  "slug": "eddie-rosendo-lino",
  "first_name": "Eddie",
  "middle_name": "Rosendo",
  "last_name": "Lino",
  "death_date": "2017-06-18T00:00:00.000Z",
  "death_year": 2017,
  "age": 23.0,
  "race": "black",
  "gender": "male",
  "image": null,
  "longitude": -118.304107484,
  "latitude": 33.9904336958
},
```

At the bottom add some JavaScript code that steps through the homicide list and adds each one to the map as a circle, just like the real Homicide Report.

```
var map = L.map('map')
var sat = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/satellite-streets-v9/
↪tiles/{z}/{x}/{y}?access_token=pk.
↪eyJlIjoibGF0aW1lcYIsImEiOiJjanJmNjg4ZzYweGtvNDNxa2ZpZ2lma3Z4In0.
↪g0lYVIEs9Y5QcUOhXactHA');
sat.addTo(map);
map.setView([33.983265, -118.306799], 18);

homicides.forEach(obj => {
  L.circleMarker([obj.latitude, obj.longitude])
    .addTo(map)
});
```

Save the file and you should now see all the homicides mapped on the page.

Next, extend the code in `_scripts/_map.js` to add a tooltip label on each point.

```
homicides.forEach(obj => {
  L.circleMarker([obj.latitude, obj.longitude])
    .addTo(map)
    .bindTooltip(obj.first_name + " " + obj.last_name);
})
```

Here's what you should see after you do that.

Next let's sprinkle some CSS in our page to make the circles match the orange color of the dots found on The Homicide Report. As we did with the charts, go to the `_styles` folder and create a new file. We'll call this one `_map.scss`. In that file, copy or write the following:

```
#map path {
  fill: #e64d1f;
  fill-opacity: 0.5;
  stroke-opacity: 0;
}
```

Just as before, that won't change anything until you import our new file into the main stylesheet. Again, use `@import` to introduce your CSS file into `main.css`

```
// Normalize Styles
@import 'node_modules/normalize.css/normalize';

// Import Modules
@import '../_modules/link/link';
@import '_charts.scss';
@import 'node_modules/leaflet/dist/leaflet';
@import '_map.scss';
```

After you save, here's what you'll get.

To make the tooltips visible all the time, edit the JavaScript in `_scripts/_map.js` to make the tooltips "permanent."

```
homicides.forEach(obj => {
  L.circleMarker([obj.latitude, obj.longitude])
    .addTo(map)
    .bindTooltip(obj.first_name + " " + obj.last_name, {permanent: true});
});
```

Here they are.

Alright. We've got an okay map. But it's zoomed in so close a reader might now know where it is. To combat this problem, graphic artists often inset a small map in the corner that shows the the area of focus from a greater distance.

Lucky for us, there's already a Leaflet extension that provides this feature. It's called [MiniMap](#).

To put it to use, we'll need to return to our friend npm.

```
$ npm install leaflet-minimap@3.6.1
```

Just as with other libraries, we need to import it into `_scripts/_map.js`

```
import * as L from "leaflet";
import MiniMap from "leaflet-minimap";
```

Its stylesheets also need to be imported to `_styles/main.scss`.

```
// Normalize Styles
@import 'node_modules/normalize.css/normalize';

// Import Modules
@import '../_modules/link/link';
@import '_charts.scss';
@import 'node_modules/leaflet/dist/leaflet';
@import 'node_modules/leaflet-minimap/src/Control.MiniMap';
@import '_map.scss';
```

Now that everything is installed, return to `scripts/_map.js` and create an inset map with the library's custom tools. We can set its view with the `maxZoom` option.

```
var map = L.map('map')
var sat = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/satellite-streets-v9/
↳tiles/{z}/{x}/{y}?access_token=pk.
↳eyJ1IjoibGF0aW1lcYIsImEiOiJjanJmNjg4ZzYweGtvNDNxa2ZpZ2lma3Z4In0.
↳g0lYVIEs9Y5QcUOhXactHA');
sat.addTo(map);
map.setView([33.983265, -118.306799], 18);

homicides.forEach(obj => {
  L.circleMarker([obj.latitude, obj.longitude])
    .addTo(map)
    .bindTooltip(obj.first_name + " " + obj.last_name, {permanent: true});
})

var sat2 = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/satellite-streets-v9/
↳tiles/{z}/{x}/{y}?access_token=pk.
↳eyJ1IjoibGF0aW1lcYIsImEiOiJjanJmNjg4ZzYweGtvNDNxa2ZpZ2lma3Z4In0.
↳g0lYVIEs9Y5QcUOhXactHA', {
  maxZoom: 8
});
var mini = new L.Control.MiniMap(sat2);
mini.addTo(map);
```

Save the file and the inset map should appear on your page.

Just for fun, let's add a couple creature comforts to map. By default, the scroll wheel on your mouse or laptop trackpad will trigger zooms on the map. Some people (Armand!) have strong feelings about this. Let's do them a favor and turn it off.

```
var map = L.map('map', {
  scrollWheelZoom: false
});
var sat = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/satellite-streets-v9/
↳tiles/{z}/{x}/{y}?access_token=pk.
↳eyJ1IjoibGF0aW1lcYIsImEiOiJjanJmNjg4ZzYweGtvNDNxa2ZpZ2lma3Z4In0.
↳g0lYVIEs9Y5QcUOhXactHA');
sat.addTo(map);
map.setView([33.983265, -118.306799], 18);

homicides.forEach(obj => {
  L.circleMarker([obj.latitude, obj.longitude])
    .addTo(map)
    .bindTooltip(obj.first_name + " " + obj.last_name, {permanent: true});
})

var sat2 = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/satellite-streets-v9/
↳tiles/{z}/{x}/{y}?access_token=pk.
↳eyJ1IjoibGF0aW1lcYIsImEiOiJjanJmNjg4ZzYweGtvNDNxa2ZpZ2lma3Z4In0.
↳g0lYVIEs9Y5QcUOhXactHA', {
  maxZoom: 8
});
var mini = new L.Control.MiniMap(sat2);
mini.addTo(map);
```

While we're at it, let's also restrict the zoom level so it you can't back too far away from LA.

```
var map = L.map('map', {
  scrollWheelZoom: false
})
var sat = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/satellite-streets-v9/
↳tiles/{z}/{x}/{y}?access_token=pk.
↳eyJ1IjoibGF0aW1lcYIsImEiOiJjanJmNjg4ZzZyYweGtvNDNxa2ZpZ2lma3Z4In0.
↳g0lYVIEs9Y5QcUOhXactHA', {
  minZoom: 9
});
sat.addTo(map);
map.setView([33.983265, -118.306799], 18);

homicides.forEach(function (obj) {
  L.circleMarker([obj.latitude, obj.longitude])
    .addTo(map)
    .bindTooltip(obj.first_name + " " + obj.last_name, {permanent: true});
})

var sat2 = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/satellite-streets-v9/
↳tiles/{z}/{x}/{y}?access_token=pk.
↳eyJ1IjoibGF0aW1lcYIsImEiOiJjanJmNjg4ZzZyYweGtvNDNxa2ZpZ2lma3Z4In0.
↳g0lYVIEs9Y5QcUOhXactHA', {
  maxZoom: 8
});
var mini = new L.Control.MiniMap(sat2);
mini.addTo(map);
```

Finally, let's preface the map with so a headline.

```
<h3>One corner. Four killings</h3>
<div id="map"></div>
```

Then an introductory paragraph.

```
<h3>One corner. Four killings</h3>
<p>The southwest corner of Harvard Park, at West 62nd Street and Harvard Boulevard,
↳has been especially deadly. In the last year-and-a-half, four men have been killed
↳there -- while sitting in a car, trying to defuse an argument or walking home from
↳the barber shop or the corner store.</p>
<div id="map"></div>
```

All wrapped up in a <section> tag.

```
<section>
  <h3>One corner. Four killings</h3>
  <p>The southwest corner of Harvard Park, at West 62nd Street and Harvard
↳Boulevard, has been especially deadly. In the last year-and-a-half, four men have
↳been killed there -- while sitting in a car, trying to defuse an argument or
↳walking home from the barber shop or the corner store.</p>
  <div id="map"></div>
</section>
```

Congratulations. You've created a custom map. Before we get on to the business of sharing it with the world, we need a couple more pieces here.

Hey. How about a headline?

```
{% extends '_layouts/base.nunjucks' %}

{% block headline %}A South L.A. neighborhood grapples with a wave of violence{%_
↪endblock %}
{% block byline %}By me{% endblock %}
{% block pubdate %}
  <time datetime="2020-03-07" pubdate>Mar. 7, 2020</time>
{% endblock %}
```

And a real byline.

```
{% extends '_layouts/base.nunjucks' %}

{% block headline %}A South L.A. neighborhood grapples with a wave of violence{%_
↪endblock %}
{% block byline %}By <a href="https://www.firstgraphics.app/">The First Graphics App_
↪Tutorial</a>{% endblock %}
{% block pubdate %}
  <time datetime="2020-03-07" pubdate>Mar. 7, 2020</time>
{% endblock %}
```

And let's write a lead.

```
{% block content %}
<section>
  <p>The area around Harvard Park was the deadliest place for African Americans in_
↪Los Angeles County last year, according to <a href="http://homicide.latimes.com/">
↪The Times' Homicide Report</a>. So far this year, six people have been killed. Most_
↪of the victims were black men.</p>
</section>
<section>
  <h3>One corner. Four killings</h3>
  <p>The southwest corner of Harvard Park, at West 62nd Street and Harvard_
↪Boulevard, has been especially deadly. In the last year-and-a-half, four men have_
↪been killed there -- while sitting in a car, trying to defuse an argument or_
↪walking home from the barber shop or the corner store.</p>
  <div id="map"></div>
</section>
...
{% endblock %}
```

Commit our work.

```
$ git add .
$ git commit -m "Added map, headline and chatter"
```

Push it to GitHub.


```
$ git push origin master
```

Now we're ready. Let's do it live.

Chapter 8: Hello Internet

In our last chapter, all the work we've done will finally be published online.

Our Yeoman framework, with its tools, structure and shortcuts, has served us well. It's been a great place to experiment, organize and develop our work. But it's useless to our readers.

The HTML, JavaScript and CSS files the framework generates are all they need. Without all the code running in our terminal, those files aren't be able to take advantage of Yeogurt, Gulp, BrowserSync and all our other tricks. But it won't matter. We can upload the simple files our framework renders to the web and they'll be enough for anyone who wants them.

That process — converting a dynamic, living website to simple files living on the filesystem — is a common strategy for publishing news sites. It goes by different names, like “flattening,” “freezing” or “baking.” Whatever you call it, it's a solid path to cheap, stable hosting for simple sites. It is used across the industry for publishing election results, longform stories, special projects and numerous other things.

Note: Examples of static news pages in the wild include:

- A wide array of [interactive graphics](#) by The Washington Post
- Hundreds of Los Angeles Times stories at latimes.com/projects
- Dozens more from The Seattle Times at projects.seattletimes.com
- Interactive apps by [The Dallas Morning News](#)
- [Live election results](#) published by The New York Times
- Data downloads from the [California Civic Data Coalition](#)

Lucky for us, Yeogurt is pre-configured to flatten our dynamic site. And GitHub has a hosting service for publishing static pages.

Open the `package.json` file at the root of the project. Scroll to the bottom. In the `config` section edit it to instruct Gulp to flatten files to the `docs` directory instead of `build`.

```
"config": {
  "//": "Entry files",
  "host": "127.0.0.1",
  "port": "3000",
  "baseUrl": "./",
  "directories": {
    "source": "src",
    "destination": "docs",
    "temporary": "tmp",
    "//": "Directories relative to `source` directory",
    "modules": "_modules",
    "layouts": "_layouts",
    "images": "_images",
    "styles": "_styles",
    "scripts": "_scripts",
    "data": "_data"
  }
}
```

Return to your terminal where the `serve` command is running. Hit `CTRL-C` or `CTRL-Z` to terminate its process.

Once you are back at the standard terminal, enter the following command to build a static version of your site. Rather than start up the local test server we’ve been using so far, it will instead save the site as flat files in `docs`.

```
$ npm run-script build
```

That saves the entire file to the `docs` folder. We’re doing that because it’s the folder expected by GitHub’s free publishing system, called “Pages.”

Commit and push to GitHub.

```
$ git add .
$ git commit -m "Built site to docs folder"
$ git push origin master
```

To take advantage of it. Go to the repository on GitHub. Click on the “Settings” tab. Scroll down to the “GitHub Pages” section. Select “master branch /docs folder” as the source. Hit save.

This will result in any files pushed to the “docs” directory of your repository being published on the web. For free.

Wait a few moments and visit <your_username>.github.com/first-graphics-app/. You should see your app published live on the World Wide Web.

Warning: If your page does not appear, make sure that you have verified your email address with GitHub. It is required before the site will allow publishing pages. And keep in mind there are many other options for publishing flat files, like [Amazon’s S3 service](#).

Congratulations. You’ve finished this class.