Table of Contents

Deploying the App

Working with Templates

Handling User Input

Routing and URLs

Using API Data

Application Architecture

Visual Feedback and Enhancement

Caching Data

Appendices

Introduction

Welcome to Practical JavaScript 2: Building Applications. This book is intended for people with no background in Computer Science and only introductory knowledge of JavaScript. The goal of this book is to help readers grow their knowledge about how modern single page applications are built while also developing practical skills for building JavaScript webapps.

This book uses Vue.js as our exemplar framework for exploring concepts of modern web applications. This framework has been chosen because of its accessibility for novice developers and its popularity in the web development world. It would be impossible to provide examples throughout this book in every possible framework. There are many viable web frameworks being used by development teams everywhere, and this choice should not be seen as a criticism of any of those other frameworks.

Throughout the book, a broad range of topics are covered. Most sections contain a blend of general information and specific instructions about how to accomplish those goals using Vue.js. The material comes quickly and it might take more than one review of the content to fully absorb it. Each section builds on the content from the previous section.

By the end of the book, the goal is that you are able to create and build a simple webapp using Vue.js and related tools. This is a lofty goal, and it will be difficult to get there. Don't lose confidence because it's difficult. There's no way to learn this stuff "easily". Software development is difficult. Writing JavaScript is difficult. But if you approach it with a clear head and some persistence, you can do it.

How to Use This Book

To get the most out of this book, read through each section and try to explore as many of the links as you can. In each section there is usually a set of Exercises. These typically involve either writing code or interpreting some code. Each section also contains a quiz, which is intended to help draw attention to some of the most important details in the content.

The most prominent feature of each section is the companion project, which is provided to help you practice concepts and skills discussed in each section. It's strongly recommended to work through each project as you complete each chapter. This will help you achieve more focused and robust practice.

What This Book Is Not

This book is not a guide to Vue.js, and it is not an exhaustive reference to building web-based applications. This book may not be suitable for people who are already well-versed in software architecture and application development. There are many aspects of web development that are ignored by this book because it would be impossible to cover the full breadth of tools and technique used by modern developers. The target audience for this book is using it as an early step down the path to becoming a web developer, and they will continue learning and growing their skills after completing this book.

Where This Book Comes From

This book was written as a companion to the Seattle University course, Building JavaScript Web Applications. The course is offered through the School of New and Continuing Studies. The projects referenced in this book use starter repositories hosted under the SU Web Development account on Github. These repositories are open and available for anyone to fork and build, and you do not need to be enrolled at Seattle University to use this book or the associated repositories.

Prerequisites and Assumptions

Before you get going with this process, you should be aware that we assume you possess some knowledge already. If you do not feel comfortable with the subjects listed below, then you should probably revisit those topics in another forum (there are many options for learning these skills online for free).

If you are missing any of the technology described below, then you should definitely install whatever apps or tools are needed.

Basic HTML and CSS

It's essential for you to know basic HTML and CSS. You should be comfortable marking up content and writing styles using traditional methods. We will advance those skills to allow you to use more modern methods of putting together HTML and CSS so that you can build better webapps.

Basic Javascript

You should know how to use JavaScript components on your pages, and how to write your own scripts to create the features you want. If you are at the "beginner" or "intermediate" level of JavaScript usage, then that's probably about right: You can manage your variables, functions, objects, and logic; you can leverage plugins and other more complex things made by other people; you can follow instructions to implement somewhat complex features such as asynchronous data calls to a third party API using `jquery.ajax()` or similar.

This book will advance your JavaScript skills and expose you to another way of thinking about how to organize your code and logic.

Basic command line

You should not be afraid of the Terminal app (although if you prefer to use iTerm2 you would not be alone). You will want to use a *nix console emulator like CMDER or the new "Bash on Ubuntu on Windows" feature of Windows 10 if you are using a Windows PC. (NOTE: You can do much of this in Powershell, too, but those techniques diverge significantly from the ones described in this book.) You may have already tried working with Linux or Unix and seen what the command line looks like.

Modern web developers use the command line all the time because it allows us to easily manage the many components of our projects, and it mimics the environment our projects run in on a public server. Many developers love the command line and the speed and ease of use. This book will regularly expect you to use the command line to manage your development tools.

Your preferred text editor

There are many opinions online about what editor is best. For this work, you do not need a "heavy" development tool (like Visual Studio or Eclipse). Many web developers prefer to use more streamlined text editors focused on providing a high quality code writing experience. You will need a text editor that you can use to edit files. This must be a plaintext editor (you cannot use Word or Pages).

Two popular choices of text editors today are Atom (a project sponsored by Github) and Sublime Text. Both editors are incredibly good, and either is an excellent choice. There are many other good choices out there, too, but this book is agnostic about which editor you use. As long as it allows you to comfortably make changes to text files, you should be fine.

Image and media editors / authoring

Any web application will benefit from lovely design and presentation. Lovely design and presentation often benefit from custom media (images, patterns, icons, video, etc.). Unfortunately, covering the ins and outs of how to make great media to go with a webapp would be a whole new book. The projects described in this book do not specifically require any media making. For anything requiring images or other media files, you can use publicly available resources (and links to sites where you can find such resources are provided where they are referenced).

You are encouraged to learn more about making great media files, and explore the many options for creating media for your webapp. You are also encouraged to explore the many options available to buy or commission custom media work from artists and makers who do that for a living. (There's no shame in reaching out to others to make your projects as great as they can be. Remember that building webapps is a team sport, in spite of the fact that you may be learning to code on your own.)

How We Build Web Applications

In 2017 we have a greater variety of websites and approaches to building web-based software than ever before. We use websites every day. We use web-based applications every day. Since 2004, when Gmail introduced the world to "AJAX-powered", application-like interfaces online the web has steadily evolved towards delivering experiences on-par with what we expect from personal computing devices.

When it was initially created, the web was envisioned as a collection of "pages." Websites were mostly concerned with delivering content. At first, the content was mostly textual. Then, websites began to embed images, audio, and video. Today it would be hard to imagine a web in which text, image, audio, and video did not cohabit the same pages. This blending of media types has fundamentally changed entire industries, which we can see by looking at the number of videos on a newspaper website, or the audio versions of articles available on magazine websites.

Over the years, more than just the content of web pages changed. As technology continued to develop, we were frustrated with the way in which web pages were slow and clunky to use. Every click on a website caused a new page to load, and if we wanted to see updated information we needed to refresh the page, which also took time.

As mentioned earlier, Google released the public beta of Gmail in 2004, and that introduced the broader web audience to a pattern known as "AJAX" (Asynchronous JavaScript and XML). AJAX was a development pattern that leveraged a feature of browsers called XHR (XMLHTTPRequest) which allows JavaScript to make a request to the server without refreshing the page.

The ability to receive new data from the server without refreshing the page meant that it was now possible to build websites that could truly function like native applications on your desktop. Moving between emails in Gmail was a lot like using Outlook on the desktop, and it made web-based email a much more enjoyable experience.

The adoption of this technique caused interest in JavaScript to grow rapidly, and it led to the development of major helper libraries like jQuery. All of this effort around using JavaScript to enhance web pages was rolled into a trend and labelled "Web 2.0." The Web 2.0 era ushered saw a significant rise in the adoption of web technology by the general public and the creation of major new players online including Facebook and Twitter.

So much has changed that it's useful to take a close look at how we build web applications today, and why we've adopted these tools and techniques.

Core Concept: Software Architecture

In software, the notion of "architecture" refers to the big-picture description of how the software is put together. This is a metaphor borrowed from other types of engineering, and we are probably more familiar with the term "architecture" as it relates to buildings.

When a building is designed, the architect comes up with all the parts of the building. It needs doors, foyers, rooms, hallways, stairways, and more. It's the architecture of the building that determines where all the features of the building are located and how they interact.

Similarly, software architecture is about identifying the core pieces of the software and how those pieces will go together. For websites and web-based applications, this architecture includes the application that runs the site, the databases that store data, and the related components (such as load balancers or message queues) that allow the site to function.

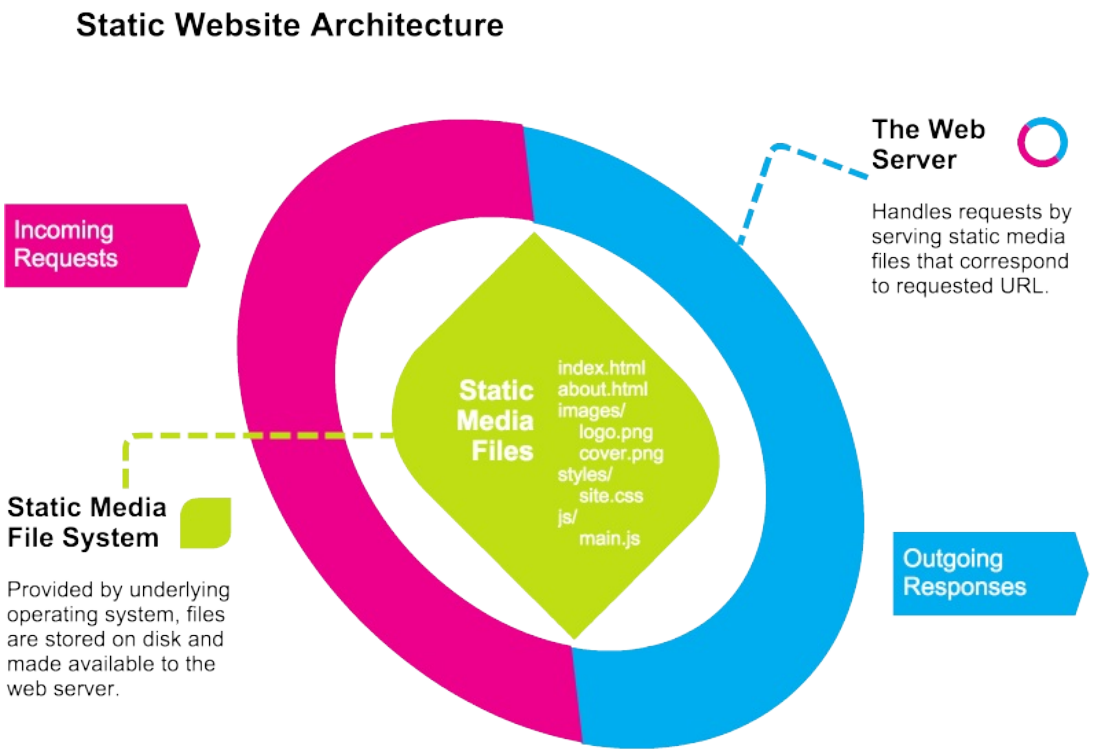
Software architecture is something that we can drill down into. We can talk about architecture at a site-wide level (the server, the database, the network), or we can talk about architecture at a simple component or app level (the parts of the app that handle different functions).

Throughout most of this book, we will discuss the architecture of a web application: What parts are involved in making the functionality we desire? But before we get into the specifics of an application, it's useful to look at how websites are architected at a broader level.

The summary definition of software architecture from Wikipedia is a good one: Software architecture is "the high level structures of a software system, the discipline of creating such structures, and the documentation of these structures." When we think of this in relation to building websites, we often talk about "website architecture."

Types of Website Architectures

There are many ways to build a website or webapp. For convenience, we will group architectures into three major styles of website architecture.



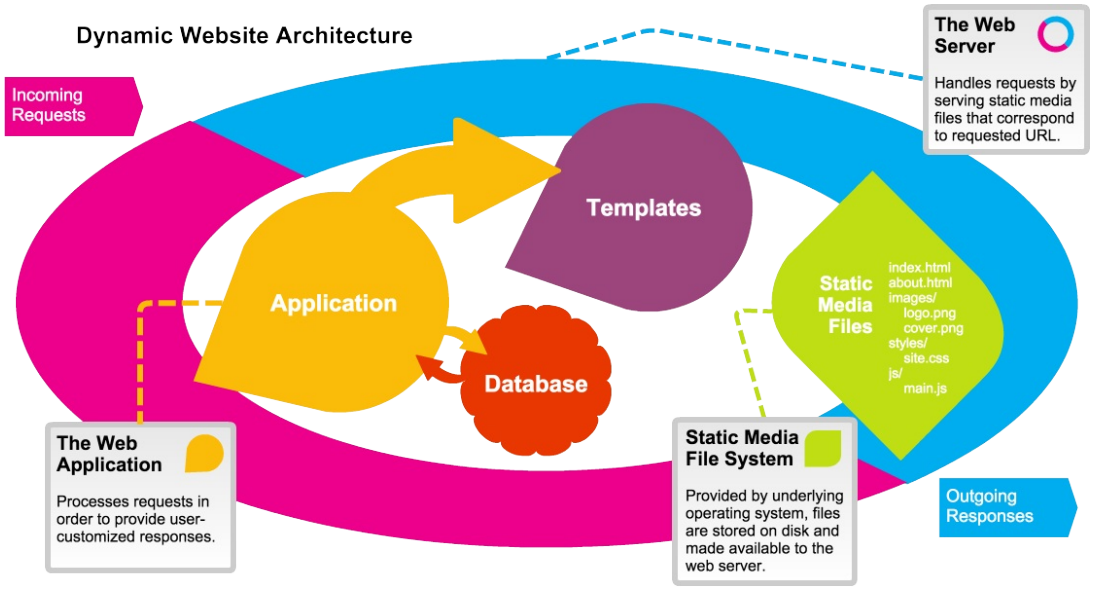
Static Website Architecture

In the simplest terms, a website can be a collection of files that do not change when people access them. These could be files of all types, including documents (such as html, xml, or docx files), media files (such as video, audio and images), or any other digital file. A web server can allow access to a hierarchy of files and directories and can serve up whatever media the user requests. Typically, a web server displays the "index.html" file in any given directory the user requests, which allows for the "index" files to serve as "homepages" for the content.

This form of website is simple and limited in terms of dynamic qualities we have come to expect (like favoriting or commenting on content). However, it is the primary use case the web was designed for: Serving content that can be referenced by a single, authoritative location (the URL). Serving static media works well with all of the different caching and performance mechanisms of the web, too, so static content tends to get to users quickly and reliably.

In fact, serving static content is so popular that there are several solutions, such as Jekyll and Pelican, that allow for content to be rendered to a static form for deployment to a static media server (such as Github Pages). Other services, which can be used via Javascript on the client side, may add features such as commenting or favoriting via the use of a third-party API.

Even in this modern era of web development, static media websites are more vibrant than ever, and there is more than can be done with them than ever before. They have also remained popular with a large group of web developers, which has insured a vibrant community.



Dynamic Website Architecture

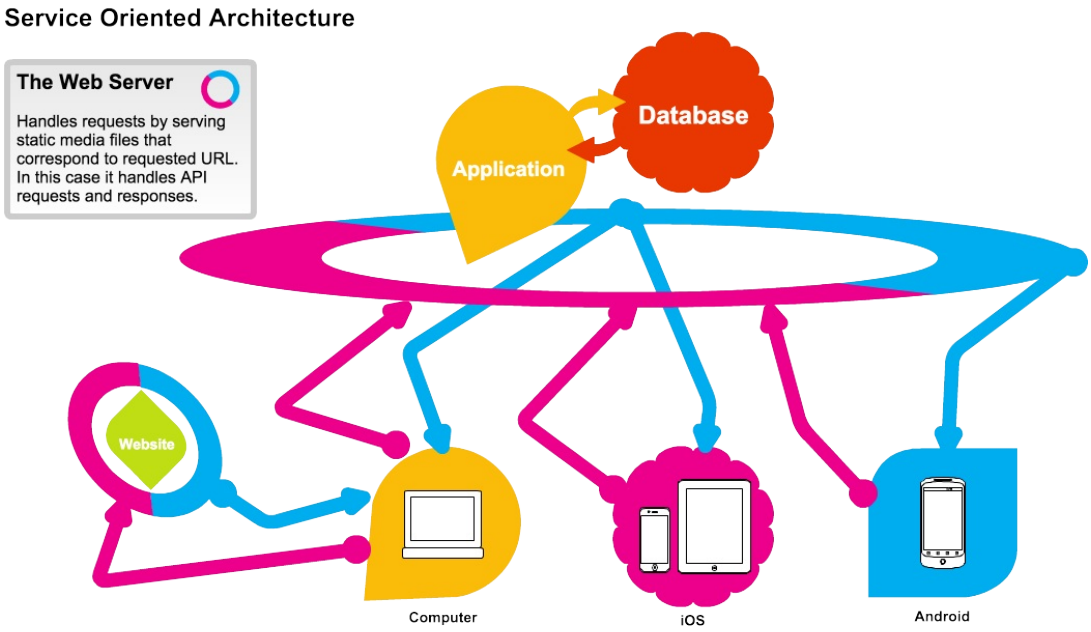
Most modern websites react to the user in some persistent way. By default, the web is "stateless"--web servers, for example, do not remember previous requests you made. They respond only to the data sent in the current request. However, we have become used to websites that allow us to login and customize content, save our favorites, share with friends, or comment on what we see. All of these features, and many others, require the addition of several functions that are not supplied by the web server, such as databases, logical processing of code, etc.

We call the websites that allow you to create an account and have some kind of customized experience "dynamic websites". They are websites that change per user and they typically store information about the user and what she has done over time.

In order to build a dynamic website it's necessary to combine several different layers:

- **The web server** still handles receiving and sending requests from clients, but now it hands those requests off to an application that determines how to respond to the user.
- **The application** is written in some backend programming language (such as Python, Ruby, PHP, Java, etc.), and each application will have it's own software architecture it uses to parse the request and create a response.
- **The database** is where the application stores persistent data such as user profiles, lists of favorites, comments on articles, etc. The application pulls data from the database in order to process the user's request.
- **The templates** provide a standardized formatting for the response. They are typically processed by the application to generate files that contain HTML, CSS and Javascript, which is sent to the user in the form of an HTTP Response.
- **The static media** includes all the images, supporting Javascript libraries, videos, audio and other binary files that do not get parsed by the application before being sent to the user.

These are the basic components of a dynamic website, but keep in mind that many dynamic websites and applications have many more components. Sometimes in order to bring your unique product to the user, it may be necessary to invent a whole new component. In other situations, you might make use of additional components to help improve website performance or to facilitate entirely new kinds of features.



Service-Oriented Architecture

Although dynamic websites have the great advantage of being friendly to all sorts of development, they have the tendency to become very unique to their operational environment and it tends to become difficult to leverage the valuable functions in the website and application across different platforms. As the desire has grown to support as many different platforms as possible, and as web technology has spread to all sorts of devices, it has become more and more important for us to deliver unique "frontend" components to each client and reuse the functional "backend" components across all the different platforms.

In order to accomplish this goal, we must separate the frontend component that gets delivered to the user's browser from the backend component that performs the business logic of our application. The backend component is turned into a "service" which means that rather than presenting a beautiful frontend for the user, the service will present an Application Programming Interface (API) that can be leveraged by any client. APIs allow two different applications to communicate and exchange data. APIs will be discussed further on the next page, but for now the important thing to keep in mind is that the API will allow our frontend app to talk to our backend service and use all the features that were previously only available to the web application in the Dynamic Website Architecture.

By reconfiguring the architecture like this,we decouple the business logic and data storage parts of our application from the presentation of the software to the user. This allows, for example, a website and a native mobile app to leverage the same set of functionality via the same API. New features can be developed at the API level, and then for each individual client in a way that makes sense for the business and product. Features can also be included and excluded on a per-platform or per-device basis. A huge degree of flexibility is achieved while enhancing maintainability and extensibility.

Many, if not most, large websites and webapps are currently using a Service Oriented Architecture. The complexity of these systems can vary dramatically depending on need and scale, but the core concept of isolating components into dedicated applications and then providing APIs to allow communication between the applications is a key pattern in modern web development.

Evolving Approaches to Web Development

As a teacher, I feel like watching students learn HTML, CSS and JavaScript looks a lot like reviewing the evolution of the Web. Over the past 25 years, we have learned how to build websites and webapps together.

Along the way we've come up with ideas that have helped us build better websites and webapps at the time. However, it's important as developers that we do not become complacent. There is always a new challenge, a new approach, and a new role to serve. How we are working in this book is the result of this continued evolution of web development.

The challenge of progressive enhancement

The generation of JavaScript libraries that birthed tools like jQuery in the mid-2000s was an era built around "progressive enhancement" (and, sometimes, progressive enhancement's less appealing cousin, "graceful degradation"). Developers could not be sure that all users would have browsers supporting JavaScript, and if they did standards adoption among browser makers could be spotty, so the Javascript might not work. Because of this, developers had to always guard against breakage.

To help with this, tools like jQuery were made to enhance the vanilla HTML and CSS that would normally be delivered to the page. Properly done, pages could be created that were totally usable in a traditional, "click and refresh", way if the JavaScript failed to load. But if the JavaScript succeeded at loading, then pages would come alive with background updates, fancy interfaces, and more.

This approach is still worthwhile, and for many purposes it may still be a great idea to think about a project from a progressive enhancement point of view. For example, if you are building a site that is dedicated to showing off specific content, or providing access to media files, there is a lot that can be conveyed in traditional HTML. That content can be loaded and accessible to a wide range of devices, then JavaScript, which is very reliable these days, can enhance the content to provide modern features.

But there are limitations to progressive enhancement. For example, what if you have a web application, and this application relies on the interaction of the user with JavaScript to have any value? We aren't imaginging a site that wants to show you a video; imagine instead a site that wants to help you make a video. What is the value of a "static" or "traditional" page in that context? Probably not much.

What webapps want

Dynamic webapps have a whole different set of considerations. If JavaScript is not enabled, then the entire app is going to be unusable. So why bother having robust fallback content? (Other than a helpful error message, of course.)

Other issues come up when you build webapps:

- With jQuery, we could make a quick call to see if, for example, a content item was favorited, or to send a new favorite to the server in the background. But what if we want to build an app where all of the screens have access to the data for objects in the system, and where we keep all that data synchronized across multiple views? This becomes very complex using only jQuery or similar solutions. We need a better way of **managing and modeling data** for use in our app.
- With progressive enhancement, we could enhance pages by making tab layouts or other custom effects. But what if we want to provide consistent headers and footers? What about slide-out menus that are the same across multiple views? These are the sorts of interface elements that exist in applications, but not so much in content-based websites. We need a better way to handle **templating views** so we can better manage our interface.
- In order to handle the data and render the views, we must write logic that applies to all views and all template

renders. With jQuery and a traditional approach, we could not easily divide logic for syncing data after changes, updating views when the user interacts, or modulate template rendering based on user preferences or application state. We could accomplish these things, but it usually required repetitive coding and created a huge opportunity for mistakes. We require a **control and logic mechanism** that allows us to abstract and encapsulate JavaScript logic into discrete, reusable components the application can utilize as needed.

- Since we do not want to make a page request to the server on each click, we cannot rely on standard links to be interpreted by a web server to move our users around our webapp. This provides us with a challenge of knowing which views to show our users as they move through our application. We require a **routes mechanism** that can be used to define specific "locations" inside our webapp.

These are just a few of the major challenges we face when we try to build more app-like websites, and the old methods just don't work.

Fortunately, developers are never short of "new methods", and we are amid a boon of what are commonly known as "Single Page Application Frameworks". Webapps (also often known as "Single Page Applications (SPAs)") are the way developers are thinking these days, and developers have many different approaches to solving all of these challenges.

Popular Frontend Frameworks

There are many Single Page Application Frameworks (SPAFs) available for building websites with JavaScript. These were created to ease the pain of making more responsive webapps that can run in the browser. The developers of these frameworks have borrowed, modified, imagined, and otherwise generated a wide range of approaches to addressing the challenge of building a webapp.

What follows is a short list of some of the popular SPAFs that are used in web development.

Vue.js

Vue.js is one of the newer frameworks to come out, but it has enjoyed serious adoption since its 2.0 release. Vue.js attempts to streamline the application development and creation process. The framework most closely resembles React, and it is compatible with many of the popular tools from the React ecosystem, but it strives to be more approachable for new developers.

Angular

Angular has been a popular JavaScript application framework for several years now. It features powerful templates (or "views"), a modern dependency management system, and robust community support. Angular is developed and managed as a Google project, so it enjoys the support of a large backer and it is used in many Google products.

Backbone.js

Backbone was one of the first JavaScript-based SPAFs to come out. It tends to be farily "unopinionated", which means that it does not make many requirements about how you organize your code. Backbone is still a very popular tool, and there are many supporting libraries and modules that can enhance Backbone apps (such as Marionette and Thorax).

Ember

Ember is a framework designed to speed development and remove mundane boilerplate code from your list of TODOs. It provides you with a more opinionated setup, meaning that it uses standard approaches to things and assumes you want to do that, too. It is built around some popular templating libraries, and boasts a full set of included features.

React + Flux

React provides a tool for making interactive JavaScript components for your webapp that keep track of their state and their data. Flux is an application framework used by Facebook to build their webapps. Flux makes use of React to create highly interactive and responsive interfaces.

Meteor

Meteor is both a client and server-side solution for building webapps using pure JavaScript. The server is designed to integrate closely with your webapp so you can quickly build custom APIs to interface with your data stores and then use them in your webapp.

TODO MVC Comparison

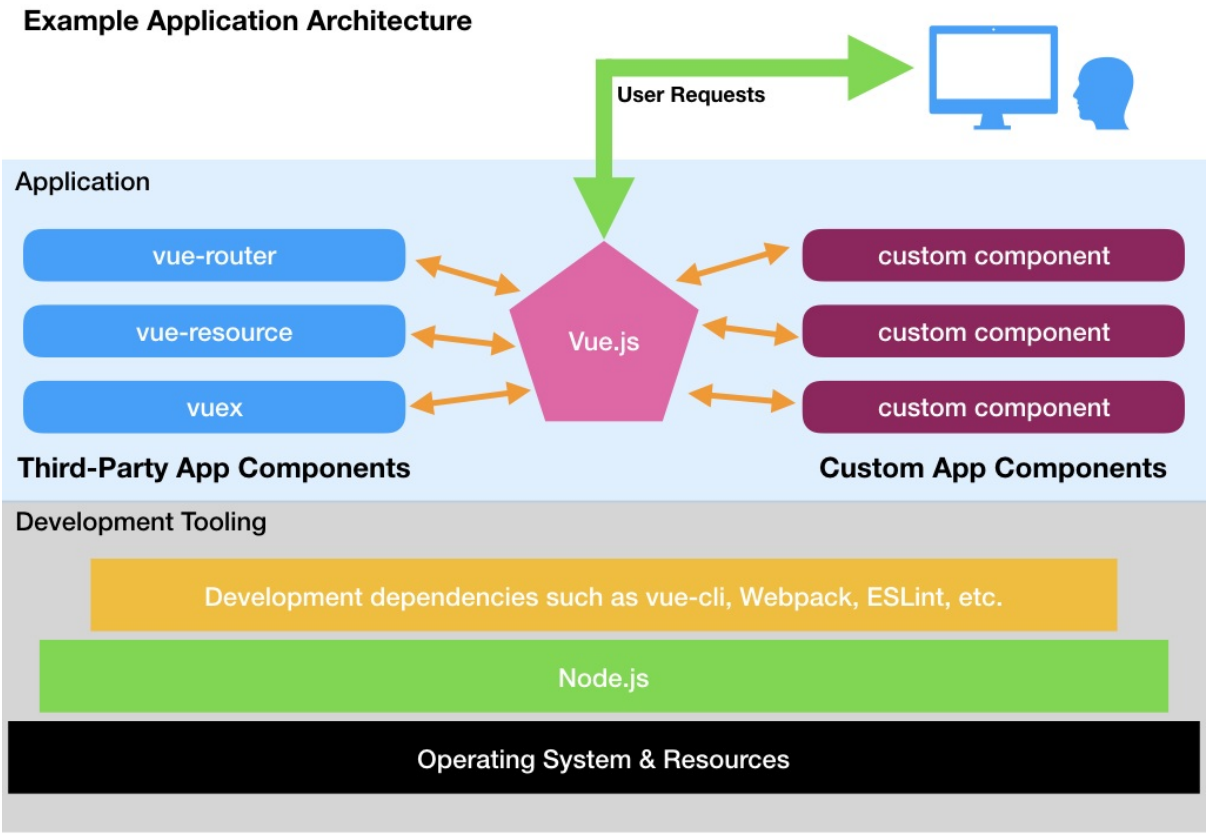
These are just a few of the popular JavaScript SPAFs that are in use today. You can find many more, along with a sample project to compare across SPAFs on the TODO MVC website. This site shows the implementation of the same TODO webapp in many different SPAFs. You can compare how the code looks in each one and get a great idea of the differences between frameworks.

Application Architecture and Logical Flow

Modern web development typically uses tools to provide convenient features for developers and to package and deploy code in a way that is optimized for end users. This means there are several components at work when we are developing web applications. These components go together in specific ways.

In this section we will outline some of the components that are going to be involved in the applications we will be building. These applications will use the Vue.js framework, but the general approach could apply to many of the SPAFs that we read about in the previous section.

For the rest of this section, we will discuss this sample application architecture.



Development Tooling

We will explore the image above from the bottom up. There are several components that are listed in the grey area labeled "Development Tooling". At the base of our development tools is the operating system. This could be Windows, Mac, or Linux, and it provides access to things we use for developing a site such as Git, filesystems, and the ability to read our code editors (to name a few).

On top of the operating system, it's common to use a fairly broad platform that enables more complex tooling. In our case, we will be using Node.js, which is a platform that allows for execution of JavaScript applications outside the browser. Many web developers have been attracted to Node.js, so they have written tools that work on the Node.js

platform. Having Node.js installed on your development system is akin to having Java installed on your computer: You might choose to write programs in Java (or for Node.js), but you are more likely to be just running programs that require those platforms.

Once Node.js is available to us, we can use a whole array of tools for helping us build, package, and deploy our web projects. In our case, we will be using `vue-cli`, which is a Node.js module created by the Vue.js team that will help us start our projects. We will also use Webpack to manage dependencies and process/package our files for delivery to end users. Webpack makes use of many things itself, and our projects will use tools such as ESLint and Babel to check our code or provide additional functionality for our development needs.

Since we are building apps that can be delivered using a static web server, we can deploy our apps using Github Pages. Because this form of deployment is so simple, we do not require robust deployment tools. But if we needed, we could also install Node.js modules that would help us deploy to whatever server we wish.

Application Components

On top of the Development Tooling box in the image above is the Application box (the blue box). This area shows the components involved in a sample web application.

The primary component is the Vue module, which allows us to declare a new application and begin working with code. For the most simple Vue apps, this might be the only component we have at the application level. However, in our example we imagine that we are using a few commonly-used third-party components: `vue-router`, `vue-resource`, and `vuex`. These components each provide features we will explore in more detail later. The important thing to remember now is that we will often use third-party components in our apps, meaning that we will use functionality that we did not create ourselves, but which we want to leverage in our applications.

On the right side of the Application box is a list of three "custom components." These components are not named, but they would correspond to features in our webapp. They might be "profile", "favorite", "post", "article", "video", "image" or whatever other content/function our webapp provides.

Although we have not specified exactly what features the custom components would add in this example, what is important to know is that these items represent components that we would write ourselves, and which would be unique to our application and its features. This is the heart of the code that we write.

File Processing/Packaging and Deployment

As mentioned above, our development tooling is concerned with processing files and packaging them for deployment, then actually getting the files to a deployed location. In the case of the most basic projects, this might simply mean pushing your repository to a specific remote server or merging a specific branch. In more complex projects, this could involve a lengthy sequence of events spanning multiple servers and services.

The needs for file processing and deployment will also change significantly depending on the technologies we are using in our web applications. It is often worthwhile to begin with a more simple approach to deploying our apps, but it's crucial to deploy as soon as possible so we can continuously test our applications in their real-world environment.

Quiz: How We Build Web Applications

Please enjoy this self-check quiz to help you identify key concepts, points, and techniques discussed in this section.

Visit Quiz Online

The quiz on this page has been removed from your PDF or ebook format. You may take the quiz by visiting this book online.

Setting Up the Workspace

In this section we will look at some of the tools for setting up modern web development workspaces. We will look at what we mean by the phrase "development environment", and why developers spend so much effort on setting up a good one. We will also install the tools we will need to work through all the concepts that the rest of this book explores.

It's important to realize that every system is unique, and there may well be challenges that you run into that are not covered in this book. To overcome those challenges will take ingenuity, research, and effort. Hopefully the links in this book combined with some good Google skills will help you get out of any tough situations.

Core Concept: Development Environments

As developers, we may work on many different kinds of software projects. Each one will require us to use different components and tools to build out. This combination of components and tools, and the various utilities, scripts and apps that are required to run them all together, make up a "development environment".

It is best to maintain a fluid notion of what a "development environment" is, or what tools are in that environment, because it is likely to change for every project you work on. It will even evolve in long-term projects over time as new techniques become available.

Some developers are keen to push their development environments further and further, which is great for the rest of us. Invariably, those developers invent new ways of doing things that become so convenient that everyone wants to adopt their methods. But it's not crucial that you become highly skilled in conceiving and constructing complex development environments.

For many other developers, the development environment is something that should fade away to the background of their work process. They don't want to think about all the things that are going into making sites build and run: Rather, they wish to focus entirely on the project of building a great site or application. For these developers, it can never be too simple or too convenient.

It's good to know a little bit about what sort of developer you think you are: Do you like to fiddle with the pieces? Or do you want to focus on the code? This is related to another decision we must make very quickly: Do you want to work in a hosted environment? Or would you prefer to work "local"?

Regardless of what the answer to that question is, there are a few things we usually look for in a development environment:

- **Development Server** We generally need to use a server to see the site we're working on. Since we don't want to trigger JavaScript security restrictions, we need to serve our files properly (as opposed to using the "file>open" dialog in the web browser). We also often want a development server to refresh when it detects a change. The Visual Studio Code plugin, live-server, and webpack's dev server will refresh with every file save.
- **Dynamic Compilation of Files** We usually want to edit smaller, shorter files that are then combined when served. The development environment should make that happen for us so we don't have to do it manually.
- **Providing Guard Rails** We often use tools like linters, validators, and test suites to make sure that we have not created any bugs or broken any features. We usually like to have these tools wired into our development environments so we can be sure to deliver higher quality code.

Hosted vs. Local

When setting up your development environment, you first have to choose whether you wish to work on your local machine (your laptop or desktop computer) or in a hosted environment (usually accessed through a web browser).

For many years, when we wanted to edit websites we logged in to a server and made changes using command line text editors like Vi and Emacs. Once desktop Linux became popular (which coincided largely with Macs converting to OS X, which is also a Unix-based operating system), it was possible for developers to begin working "local" -- on their own computers.

Working in a "local development environment" has a number of advantages: It allows for the fastest possible response from your test server (since that server is running on the same computer you are using), and it allows you to configure every small aspect of your environment. This is the way most developers work today.

The down side of local development is that it can often be complex, and, especially when working on a large project in a team setting, it can lead to wasted hours spent fixing development environments and keeping everyone running together. There are many tools and techniques that have been developed to mitigate those issues, so you may hear about things like "virtualization" and "containers" in relation to local development environments.

The alternative to local development environments is the hosted development environment. These services (like CodeAnywhere.com, CodeSandbox, and Cloud 9) offer hosted development environments that you can bring up with a few clicks of the mouse. They all offer some level of interface built on top of the development environment to allow you to make easier choices when provisioning your development environment.

On these systems, you typically create a "devbox" (or whatever creative term they use) with a base configuration. Those base configurations range from "HTML5" and "Python" to "Drupal" or "MEAN Stack". These configurations often have all the tools installed that we are installing during this first project.

How to Use This Book

This book assumes you are working in a local development environment. It provides most of the information you need to get various components up and running.

If you are using a hosted development environment, then you can probably bring up a "Node.js stack" or "HTML5 stack" and have access to the core components that we need here. For most of the hosted development environments, you can also run installation commands if, for example, you can bring up a Node.js devbox, but it doesn't have Yeoman and related apps installed. The instructions provided here for installing those components should also work on your hosted development environment.

One area where there may be differences about how to set things up is when it comes to setting up Git and connecting your hosted development environment to your Github repository. You will likely need to consult the documentation for whatever hosted development environment provider you are using. (Almost all of them support connecting dev environments to Github.)

Development Tooling

We will work in what is known as a "Node-based" environment. That is to say, a development environment built on the Node.js platform. We will use the Vue-CLI to bootstrap our apps, and that will allow us to use several popular tools for frontend development, including Webpack.

We will also use Git and Github to manage our code and deploy our projects.

About Node.js

Node.js is a development platform that runs Javascript outside of the web browser. This is noteworthy because there are no other major implementations of Javascript outside of the web browser.

The specific Javascript engine Node.js uses is called V8, and it is the engine that is used in Google's Chrome browser. This is a very fast and reliable Javascript engine, and Node is known for being a very fast platform.

Node allows developers to create tiny Javascript applications called "modules", and then to string those modules together into larger and more complex applications. There are thousands of "Node modules", as they are known. Developers manage modules with a tool called NPM (Node Package Manager), which is installed when you install Node.js.

We will use a variety of extra modules to add functionality to our website, and to do so we will use NPM. NPM is akin to `apt-get`, `homebrew`, or other tools you may have encountered working on Linux or Unix systems. These package managers are programs that can reach out to repositories on the internet and pull down code for use in development. Take a moment to explore npmjs.org, the web app that allows you to search the npm repository and read about the opensource JavaScript code that is available to you. Anyone who creates a valid package.json file for their code can submit it to the npm repository.

About Vue-CLI

We will use the Vue.js SPAF (single page application framework) as the example platform for this entire book. That means examples will be given in the context of Vue.js, and we will be looking at a lot of Vue.js-specific techniques and tools. This will allow us to learn both more general concepts and pragmatic skills we can use in day-to-day work.

In order to quickly get started on building a Vue.js application, we will use the Vue-CLI to bootstrap our project skeletons (we will talk more about that next week). The "CLI" in "Vue-CLI" stands for "Command Line Interface." This an application that you run on your local development machine. It takes arguments like any other command line application, and it produces project skeletons which we can use to quickly get going on a new website.

Node.js is a development platform that runs Javascript outside of the web browser. This is noteworthy because there are not other major implementations of Javascript outside of the web browser.

The specific Javascript engine Node.js uses is called V8, and it is the engine that is used in Google's Chrome browser. This is a very fast and reliable Javascript engine, and Node is known for being a very fast platform.

Node allows developers to create tiny Javascript applications called "modules", and then to string those modules together into larger and more complex applications. There are thousands of "Node modules", as they are known. Developers manage modules with a tool called NPM (Node Package Manager), which is installed when you install Node.js.

We will use a variety of extra modules to add functionality to our website, and to do so we will use NPM. NPM is akin to `apt-get`, `homebrew`, or other tools you may have encountered working on Linux or Unix systems. These package managers are programs that can reach out to repositories on the internet and pull down code for use in development. Take a moment to explore npmjs.org, the web app that allows you to search the npm repository and read about the opensource JavaScript code that is available to you. Anyone who creates a valid package.json file for their code can submit it to the npm repository.

About Webpack

Webpack is a multi-purpose tool primarily concerned with bundling the files of your website together for efficient delivery to the end user. In modern web applications, we don't want to force the user to download dozens of files; we don't want to have troubles with users caching old versions of CSS or JavaScript files; and, we want to make the smallest files possible so they download super fast for our users. Webpack helps us achieve all of this.

Depending on the configuration, webpack can gather all of our CSS and JavaScript, "minify" it down by removing unnecessary blank spaces and lines, and bundle it up into versioned files with a unique filename. This addresses all of the problems we listed above. Webpack can even connect into other tools for validating code or spotting bugs in code and automatically run those as it processes files. It can do a lot more, too, but for now we will keep our use of Webpack as simple as possible.

As you develop code on Vue.js, you'll be creating files that cannot run in a browser as they contain a combination of HTML, JavaScript and CSS in a non-standard format. Webpack can use "loader" scripts to convert these files into standard HTML, JavaScript and CSS.

We will use Webpack to build out our files for delivery as a static website. Because we are generating a static website, we'll be able to serve the application from github pages.

About Git

Git is a source control software used by many developers for web and many other types of projects. Git is open source and free, which makes it available to everyone.

Git is also "distributed", which means each person who has a copy possesses the entire history of the project, and each developer could serve as a Git server for any other developer. This allows for a great deal of flexibility when working on projects.

Finally, Git is well-known for the way it handles branching and merging. Although these concepts will only be lightly addressed in the content of this book, these are important features used daily by most developers.

About Github

Github is a hosting service with enhanced tools to handle code repositories using Git. People are often confused about Git and Github. Think of the way Flickr is a hosting service for your images. Flickr is not a camera or illustration software -- it is a tool that allows people to host their images, organize them into galleries, and connect with others who are doing the same thing. Likewise, Github is a tool that allows you to host your code, work with it using helpful tools (like issue management, or wiki pages for documentation), and then connect with other people who are doing the same thing.

Github offers several services that go far beyond simple "Git repository management", and one of those features is called Github Pages. Github Pages is a static website hosting service, meaning that you can use Github to publish HTML, CSS and Javascript to the web and make it available for the general public.

Install Node.js

Installing Node.js is a straightforward process for most systems. On any platform, you can install Node.js using one of their pre-built installers available at http://nodejs.org/.

Please Note: Mac users should use Homebrew to install Node.js. DO NOT use the Node.js installer on a Mac unless you know what you're doing. See below.

To install Node.js, simply download the installation package for your platform and install it like any other application.

However, for each operating system, you may consider an additional or alternative process.

Windows Installation

Windows users should navigate to the Node.js download page and choose the 64-bit Windows installer. Once this is downloaded they can open it (click on it) to run the install.

Mac Installation

Mac users should use Homebrew to manage packages like Node.js (and many other development tools). Homebrew is a popular package management tool for Mac and it makes installing many packages much easier. The Homebrew website (http://brew.sh) has a great installation tutorial to follow. Once you've installed Homebrew on your system, you can install Node.js with one command:

```
brew install node
```

Once that command completes successfully, you should have Node.js up and running on your Mac.

If you install Node.js using the installer on a Mac then it will likely lead to permissions errors later on and require you to use `sudo` way too much. Please use Homebrew to install Node.js on a Mac.

Testing Your Installation

Once you have completed your node installation, open a terminal on Mac or Gitbash on Windows and run the following command. You should see the latest stable version of node.

```
node --version
```

Installing node also installs npm so you can test for that now too.

```
npm --version
```

It's also nice to know where your code has been installed and you can find this out by running the 'which' command.

```
which node
which npm
```

Install Git and Setup Github

It's likely that you already have Git and Github set up in your local development environment. We will use Git to version our code, and Github as both a server to host our repository and as a hosting platform to host our projects.

If you have not yet set up Git and Github, follow the links below to get these set up on your development environment.

Please note: Hosted development environments may connect to Github in a variety of ways. Please consult the documentation for your provider in order to properly connect your development environment to your Github account.

Installing Git

One of the easiest ways to install Git on your computer is to install the Github application. Here are links for the application on Windows and Mac:

- Git for Windows
- Github App for Mac

It is recommended that if you are on a Windows or Mac machine and you have never set up Git before, you should start with the Github app. This app is useful to manage your code, and it also helps you set up your global Git preferences.

If you don't want to install the Github application, you may install Git directly using one of the downloads from the official Git website: http://git-scm.com/downloads.

If you need to download Git directly and install it, you may wish to consult the "Setting Up Git" guide from Github.

Setting Up Github

You will need a Github account to use Github for the purposes of these projects. You may, of course, use another service for hosting your Git repositories, but this book will also leverage the Github Pages feature to deploy our application. It is probably essential to have a Github account to complete everything in this book.

Github has several great resources for learning how to use it and how to set it up. The trickiest aspect of setting up your Github account is adding your development computer's SSH keys to your Github account. This is covered in their guide: Generating SSH Keys.

In addition to setting up the basics, you may want to bookmark the Github Guides homepage as a destination for later. Github guides cover basic topics of using Github and Git, and they are very good learning resources.

Please Note: The Github guides that are linked above contain instructions for every operating system. Look for the platform links below the title of the page if it does not automatically select your platform.

Notes for Windows Users

This book assumes a Unix-like development environment. This is sometimes a bit tricky to set up if you're new to web development and you use Windows. Macs and Linux run a bash interface, and after you install git on Windows you'll have access to git bash which is a windows application that provides a bash interface similar to what you experience on a Mac or Linux OS.

Please consider purchasing a new computer if you are a Windows user and can't upgrade to Windows 10.

Quiz: Setting Up the Workspace

Please enjoy this self-check quiz to help you identify key concepts, points, and techniques discussed in this section.

Visit Quiz Online

The quiz on this page has been removed from your PDF or ebook format. You may take the quiz by visiting this book online.

Project: Testing the Workspace

In order to determine whether our development environment is set up properly, it's useful to try cloning and running an existing project. Work along with the following directions to test your workspace.

This project uses the WATS 4000: Up and Running repository.

Fork and Clone

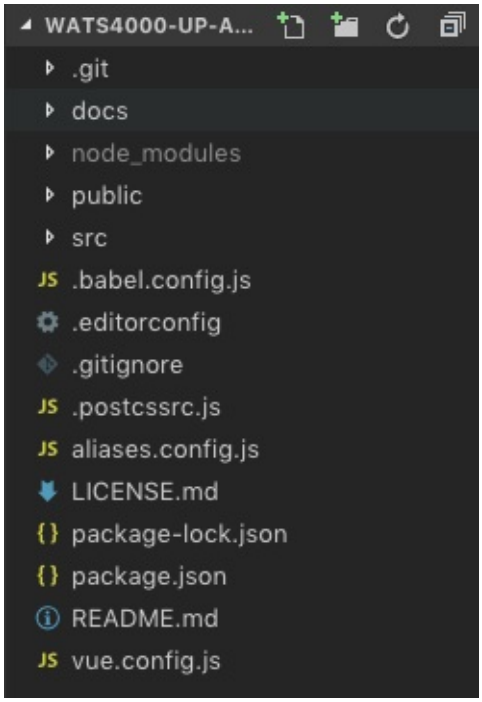
You will need a Github account to fork the repository. Visit the repository homepage and click the "fork" button. Once you have completed forking the repository, you can clone it to your local development environment. This will copy all the files from the repository to your workspace.

Install Dependencies

The `package.json` file in the root of your project contains information for retrieving software from npmjs.org. `dependencies` are the packages that will be used by your code and `devDependencies` are the packages that will be use to build your code into static files.
In order to get this project to work, we need to install the dependencies. To do so, run the `npm install` command in the project repository. You should see output in the form a a directory called `node_modules` indicating that NPM has installed quite a few packages (this may take several minutes to complete). The `node_modules` directory can be quite large and there is no reason to save it on github.com, so we include it in a .gitignore file. You may notice in your IDE, depending on how it is configured, that the `node_modules` directory is a lighter color indicating it won't be pushed to github.com.

Once you have finished installing dependencies, take a look at the project itself.

Project Files



Project Folders after cloning and installation of dependencies.

You will notice several key parts of the project here. Let's take a quick trip through what we see in this list and identify some of the key parts of the application. We will go quickly through this for now because we will revisit almost all of these pieces as we work through more details of building applications.

- `.git` – This is a hidden directory and it contains configuration information about your git project. You can list the contents of the config file in .git to see if you are connected to a remote repository on github.
- `docs` – This directory the static files that will be used to serve your application on gh-pages. This directory is built by using `vue CLI` commands and configuration information stored in `vue.config.js`.
- `node_modules` – This is where all of the dependencies the site uses are stored during development. These are the files that are downloaded from npmjs.org when you run `npm install` (Check it out; there are lots of these!)
- `public` – This directory contains the index.html and other static assets that are used to build your app. These assets can include files to be used as favicons.
- `src` – This is where the custom code we write for this unique project goes. Look inside and you'll see the components that make this site work.
- `babel.config.js` – This is a configuration file for the babel code transpiler. Babel is a dev dependency that is used to turn code written in `.vue` files into static HTML/JavaScript/CSS files.
- `.editorconfig` – This is a hidden config file for the Visual Studio Code editor. Depending on which editor you are using you may see a file with a similar purpose but named differently.
- `.gitignore` – This is a hidden file that contains names of files and directory which will not be pushed to git.
- `.postcssrc.js` – This is a hidden file that contains a module that will help build the CSS files.
- `aliases.config.js` – This is a config file that tell Vue.js that `@` and `@src` are aliases for the `src` directory. This is a coding convenience.
- `LICENSE.md` – This project using and MIT license.
- `package-lock.json` – This file is created, if it doesn't already exist, when you run `npm install`. It describes the exact code tree that was generated by the last call to `npm install`.
- `package.json` – This file contains the listings of npm dependencies and devDependencies. When you `npm install <package>` a new package it gets recorded in this file. The file also contains a `scripts` object that defines the command line scripts that can be run to build for development and production.
- `README.md` – This file contains a description of the project and instructions for working with it.
- `vue.config.js` – This is a config file used to specify options for the build. This is where you find the option to

build to the `docs` directory and you can turn the linter off in here. Any other build options can be added to this file.

Run the Dev Server

The `npm run` command looks at the `scripts` object in package.json to find the commands to run. If you look at the package.json script for this project you can see that it the commands map to a call to the `vue-cli-service` program. You can also see that this program is installed locally when you run `npm install` because it is in the `devDependencies` object.

We use the `serve` script to run our Dev Server.

As we work, we want to run the development server. When you do, you should see the app running. You can check it out in your web browser and test out the simple functionality.

To start the dev server, go to the terminal and type:

```
npm run serve
```

The output of this command indicates that we have a dev server running at `http://localhost:8080`. This is an in memory server and there are no files written to disk.

```
Rebeccas-MacBook-Air:wats4000-up-and-running becky$ npm run serve

> up-and-running@0.1.0 serve /Users/becky/projects/4000/wats4000-up-and-running
> vue-cli-service serve

 INFO  Starting development server...
 98% after emitting CopyPlugin

 DONE  Compiled successfully in 4541ms


  App running at:
  - Local:   http://localhost:8080
  - Network: http://10.0.0.75:8080

  Note that the development build is not optimized.
  To create a production build, run npm run build.
```

While we have the dev server running, we can open up our editor and modify some of the files. If we open `src/App.vue` we will see the template that includes the `<h1>` tag for the page. We can modify the contents of that tag and save the file to see the live reload capability of this development server. (We can also play with the CSS in this file if we wish.)

Note: We have not gone over how a Vue.js app works yet, but it's still worthwhile to poke around and see. If things break, we can always reset your clone of the repository.

Once we done experimenting with the development server, we can quit the server using the `ctrl-c` keyboard combination in the terminal.

Build the Site

As we mentioned earlier, the goal of using development environments is that we can work with source code that is more friendly and still deliver highly optimized code to our users. To get a feel for how this works, run the following command:

```
npm run build
```

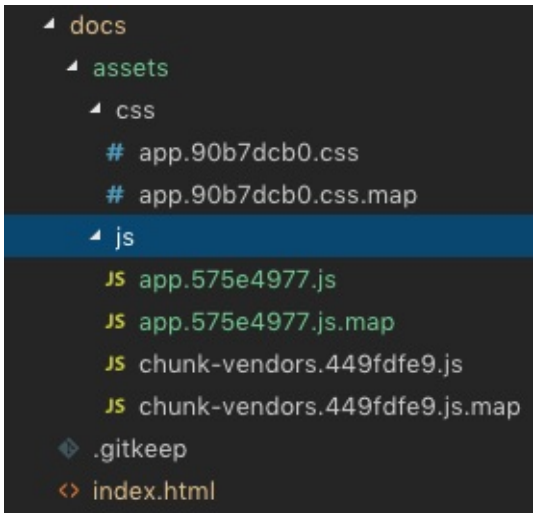
This command builds and compiles all the code in our project and puts it in a directory called `docs` in the root of the project.

```
Rebeccas-MacBook-Air:wats4000-up-and-running becky$ npm run build

> up-and-running@0.1.0 build /Users/becky/projects/4000/wats4000-up-and-running
> vue-cli-service build


⠙  Building for production...

 DONE  Compiled successfully in 6597ms

  File                                  Size             Gzipped

  docs/assets/js/chunk-vendors.449fdfe9.    88.01 KiB        31.82 KiB
  js
  docs/assets/js/app.575e4977.js            2.62 KiB         1.25 KiB
  docs/assets/css/app.90b7dcb0.css          0.66 KiB         0.39 KiB

  Images and other types of assets omitted.

 DONE  Build complete. The docs directory is ready to be deployed.
 INFO  Check out deployment instructions at https://cli.vuejs.org/guide/deployment.html
```

Webpack Build Report in Terminal

In the terminal view of the Webpack Build Report we can more easily see that webpack has combined files and organized them by "chunks". We can see the file size and the file name that has resulted from the processing of these files.

If we look inside the `docs/` folder, which was created inside our repository by the build process, we can see that those files have been saved under `dist/static/` (as well as the other files that make up our site.



Files in the `docs/` Folder

These files are the results of combining many source files into just a few files for delivery to our users. There is also some transpiling taking place to turn a `.vue` file into HTML/CSS/JavaScript. This cuts down on the number of request a user makes to the server in order to download our application. We also see that the files have been "versioned" with the addition of "hashes" that represent a unique version of that file. These hashes are updated when the contents of the files that are being combined changes. Changing the names of these files when the contents change helps us avoid issues of browsers and networks caching files and serving users old code. (We never want to tell a user to "clear the cache" in their browser again!)

Continue Exploring

It's worthwhile to continue exploring the code to get a better understanding of what parts we're dealing with. Think of this as moving into a new neighborhood: There are already buildings and streets here, so we don't need to build everything from scratch. But we do need to find our ways and get comfortable in this new environment.

Much of the rest of this book will serve like a GPS as we learn about building applications. We will get details that should make all of this doable at a fundamental level. But please take detours and try things outside what is prescribed in this book.

If we don't take some ambling walks around our new neighborhood we will never find the better paths, interesting scenery, or potential for enjoyment that exists.

Bootstrapping a New App

In this section we will discuss the concept of the "application framework" and how we use tools to create basic structures that we can then fill up with functionality. It's important to realize that we usually stand on the shoulders of developers who have come before us, and for most of us it's better to learn to use the amazing tools that other developers create than for us to create those tools from scratch.

There's an old axiom among software developers that the best line of code we could write is the one we didn't write. This is because any time we are writing code we are creating the possibility of making a mistake. When we approach large, complex tasks, the likelihood of doing something wrong becomes even more significant. It's best to tackle really big software as a group, and it make sure we adhere to some good review and feedback processes so we can be sure to deliver high quality, bug-free code.

Since most of us, especially when we're just starting out, are working solo or in small teams, it's difficult to take on those great big challenges that produce things like reusable application frameworks. It's often tempting for developers to just begin writing code with what they know, but if the goal is to build a modern web application things will quickly get out of hand if we don't have a clear plan and approach. It turns out that it's usually too difficult to write our own frameworks, and it's too difficult to write apps without a framework.

So what do we do? We use someone else's solution.

Application frameworks, styling frameworks, and other large, generic components are created (usually) by groups of people working together to solve the problem. Even in the cases where a project is primarily spearheaded and stewarded by a single individual (such as with Vue.js, which only has one full-time project member, it's creator, Evan You), larger frameworks are used by a community of people who participate in testing and development of the codebase.

In this book we will use the work of developers who have come before us in order to hit new heights of production. We will use Vue.js as the primary example of an application framework, and we will combine many other tools (including Node.js, NPM, Webpack, and more) to build our apps. Rather than inventing our own approach to solve the problems of how to build a great application, we will strive to learn and understand the ways that others have solved these problems.

Core Concept: Application Frameworks

An application framework is a body of code designed to make it easier to create applications. Frameworks are usually language-specific and purpose-driven. For example, Django is a web application framework for Python, and .Net is a framework for writing Windows applications in C#. Frameworks also tend to be driven by some sort of design philosophy. This means that the framework usually represents a way of thinking about writing code and organizing applications.

These factors combine to give a framework its "feel" and features. For example, in a game development framework, it would not be odd to focus on an event-driven architecture, since that might better suit writing certain kinds of games. We would also expect that a game framework would give us the ability to manage things like character animations and adding/removing entities on the screen.

In a web application framework, we expect to have features that allow us to create URLs that have specific functionality, process templates to generate pages that contain user-specific data, and manage the basics of security, speed, and general best practices. Whether we are working with Ruby or Django, we expect the same core features (and both of those frameworks deliver many desirable features).

Difference Between a Framework and a Library

Sometimes it becomes murky to see the difference between a library (like jQuery) and a framework (like Vue.js). It often feels like libraries are "smaller" than frameworks, although some libraries (again, like jQuery) are extremely large, and some frameworks (like Backbone.js) are tiny. In fact, there are three primary characteristics that make a body of code a framework as opposed to a library. Quoting from Wikipedia, these characteristics are:

- **inversion of control:** In a framework, unlike in libraries or in standard user applications, the overall program's flow of control is not dictated by the caller, but by the framework.
- **extensibility:** A user can extend the framework - usually by selective overriding; or programmers can add specialized user code to provide specific functionality.
- **non-modifiable framework code:** The framework code, in general, is not supposed to be modified, while accepting user-implemented extensions. In other words, users can extend the framework, but should not modify its code.

So when we use a framework, we give up some of the control over how our application is organized and which parts get invoked first. We must adapt to the methods of the framework itself in order to make a functional piece of software. When we use a framework, it doesn't make sense to try to undermine the basic logic of the framework. We generally do not touch the core code of the framework itself. We want to keep our underlying frameworks as clean and unmodified as possible in order to enable better future maintenance and expansion of our applications.

Of course, the framework is supposed to enable a reasonable variety of projects, so the point about extensibility is important. Rather than modify the core code of the framework, we add our own components and configure the details of how features work in order to extend the functionality of the framework to fit our needs. A framework should give us an easy way to write our application logic and to bring in additional features unique to our requirements.

Easing the Pain of Getting Started

When we use frameworks, one of the benefits we get is that we can usually start a project more quickly. There are many things all web applications do, and setting up the basic foundations for those core features is repetitive and time-consuming. With a framework, we can often run a simple "bootstrap" script and get directly into writing our own code.

In the case of Vue.js, there is a command line application called Vue-CLI (the "CLI" stands for Command Line Interface) that we can use to manage our projects. The Vue-CLI comes with several "official" project templates (and it allows developers to write their own project templates, too), so we can quickly bootstrap a "project skeleton" based on one of the project templates. That's a lot of jargon, so let's unpack it.

A "project template" is a set of files that represent the beginning state of a generic Vue.js project. Templates might vary depending on the type of project they are trying to represent, and it's possible to create our own custom templates that we could use for our unique requirements.

A "project skeleton" is what we have after we "run" or "bootstrap" a project template. We are given a bare bones application that has the core features the framework gives us, and which we can then extend to build out our unique functionality.

Bootstrap vs. "bootstrap"

Many new developers become confused by the use of the word "bootstrap" in web development circles. This is because we have Bootstrap, which is a popular style framework for use on projects. But the word "bootstrap" has been used by developers for ages to refer to all sorts of "getting things started" situations.

We often refer to "bootstrapping" a project, in the sense described earlier on this page. We also refer to "bootstrapping" a server, meaning powering up and configuring a server, or "bootstrapping" an application, meaning setting up the environment and dependencies for an application that already exists. In a generic sense, "bootstrapping" can refer to any technical process of getting something going.

That's why the team that created the Bootstrap style framework chose that name: Their goal was to create a framework that would help developers get going more quickly with a visual style that, although a little generic, would look better than no styling at all and which would put developers in a good position to write high quality styles in their projects. The name Bootstrap was inspired by the fondness we developers have for the term "bootstrap" in general.

Choosing a Framework

It can be difficult to pick a framework, especially when we face the kinds of lists we saw in the Popular Single Page Application Frameworks section of this book. It's well worth the research effort to try to pick a more suitable framework for our projects. Consider these questions when reviewing information about frameworks:

- What languages are we interested in using? (Or, if that's not clear, what languages are used for building these kinds of applications?)
- What frameworks are used to build similar applications?
- What frameworks are developers that we know using? (Being close to a user community can really help learning and speed development.)
- How big is the online and local communities for the framework we're considering?
- How many resources are available to help us learn the framework?

Notice that there are a couple of questions left out of here that others might bring up. Specifically, we are not concerned with "performance" and we are not concerned (too much) with "feature set." These are consciously left out of our consideration.

For the most part, popular frameworks perform at similar levels. Different frameworks may require different types of optimization, and there may indeed be some cases where a specific framework will outperform the rest, but our decisions about which tools we use should be made based on how effective those tools will make us. Using a highly performant, but very confusing, framework will slow us down more than using a less speedy but much more understandable framework. It will also make it more possible to get help from collaborators and teach others what we've done.

Similarly, the feature set of any given framework is likely to cover most of the basics, just like the other frameworks in the space. It's unlikely that our decision will be swayed because one framework is much more feature rich than others (and if we find that unique case, then it might be an indicator of which framework we should use). We expect that a framework will allow us to extend functionality, and we anticipate that we will be writing functionality. So we are not restricted to the features the framework offers, and we can build whatever else we need.

Vue.js was chosen for use in this book using similar reasoning to the above. It is a framework that generally functions like the other major frameworks popular today (React and Angular), and it has been gaining in popularity for a couple of years. It is used to build some impressive web applications, and it contains many of the features we expect out of the box. It is just as performant as React and Angular, too, making it a good choice for delivering quick web applications.

The feature that pushed Vue.js over the top was primarily its accessibility for new developers. This book is meant to teach people with very little web development experience. We need something that contains a good amount of "batteries included" (meaning, we do not have to add in every feature and component we want to use individually), and we need something that strikes a nice balance between introducing tooling (such as Webpack and development servers) without going too far. React and Angular each strive for a modularity that is desirable for advanced developers, but which can be confusing for new developers. They both also assume very large, complex projects, and, again, a great degree of developer expertise to put together components to serve a project.

For these reasons, we are using Vue.js in this book. Most of the concepts used by Vue.js are the same as those used in React and Angular. Learning how to use Vue.js will make it easier to learn React and/or Angular in the future, should we choose to do so. In the meantime, we will build some amazing projects.

Creating a New Vue.js Application

Creating a new Vue.js application is a fairly straightforward process, and it is something that will become very easy once we understand the basics. It's often worthwhile to bootstrap a brand new project in order to try out some technique or idea outside of the complexity of a larger software system. Sometimes when debugging it's good to make a fresh project and then move parts of our code into that project in a methodical way in order to find the pieces that break. There are many other reasons why being able to quickly get up and running with a project is valuable.

Install Vue-CLI

In order to use the Vue-CLI tool, we must first install it. This software requires Node.js to work, and it can be installed using NPM. To install Vue-CLI in our development environment, run this command:

```
npm install -g @vue/cli
```

This will install Vue-CLI globally so we can use it anywhere. Once the installation is complete, we can check the version that was installed by running `vue --version`. If everything went smoothly, we should see the version number displayed. We can also run `vue --help` to see a list of commands we can use, or to get more information about specific commands.

vue-cli

Installing "local" Versus "global" Dependencies

When we set up our development environments, we want to keep in mind that we may be working on many different projects throughout our career as developers. This means we need to keep a development environment that is robust enough to work for us, but not so specific that it won't work on the next project we undertake. Because of this, we usually install "local" versions of the dependencies our project requires.

Consider the way we installed all of those dependencies into the `node_modules/` directory in our project in the last section. All of those dependencies were "local" to the project because they were installed in a directory within the project repository.

However, there are some tools we might need to use across multiple projects. This often happens with tools like Vue-CLI, which can be used to bootstrap and work with many different projects. In these cases, we want to install the tools "globally" so that we can use them in any project.

When installing applications with Node Package Manager (NPM), we use the `-g` flag to install any Node module "globally" so that it can be used in any project. When we install Vue-CLI we use the `-g` flag so that we can run the `vue` command from anywhere.

NOTE: In this book we are using version 3 of the CLI. Version 2 of the Vue CLI was available in a package called `vue-cli`. NPM provides the `@<organization>/` prefix to create scoped packages. Scoped packages provide namespacing so that an organization can create multiple packages under a single name. If you have installed version 2 of the Vue CLI you will need to uninstall it using this command `npm uninstall vue-cli -g`.

> It's important to see the distinction between the Vue framework code which is not installed globally and the Vue CLI code which is installed globally. You will see an entry in your package.json file to install Vue locally.
>
> Once you have installed @vue/cli globally you can keep it updated without uninstalling it. Just run `npm update -g @vue/cli`

Using Vue-CLI

The Vue-CLI tool allows us to quickly bootstrap projects using project templates. These templates can be created by anyone, so if we find ourselves needing a very specific or exotic template, we can create that and use it with Vue-CLI. Since we are just beginning, we don't really have any exotic or highly specific needs. Instead, we can use one of the included "official templates" maintained by the Vue.js community.

Throughout this book it is recommended to use the `webpack` template, which defines a project with all the features we will explore throughout the rest of this book. Webpack is a popular application bundling and build tool, and it is used with many technologies (including React and Angular). Webpack handles dependency management (making sure all our third-party modules are available to our application when we deploy) and build process (all the tasks that must happen to package our application for deployment). Webpack is advantageous because it handles things that other tools (such as Browserify) rely on external helpers (like Gulp or Grunt) to handle.

In short, Webpack keeps the burden on us, as developers, very low, and the way that the official Vue.js template for a Webpack-based application is configured is comfortable to use. We will be investigating how this template is put together and how we can use it more throughout the rest of this book, but for now it might be interesting to look through the documentation of the Vue.js Webpack Template to get an idea of all the things that are in there. Keep in mind that we don't need to understand or master all of this stuff right now. As we work through building apps and making them available to the public, we will touch on many of these things.

We'll provide some configuration files that can override some of the default template configuration. These configuration files can help with issues like deploying to a `docs` directory instead of a `dist` directory for deployment on github.com's gh-pages.

In order to create a new app with the Vue-CLI, change directory into your Projects area and run this command:

```
vue create test-project
```

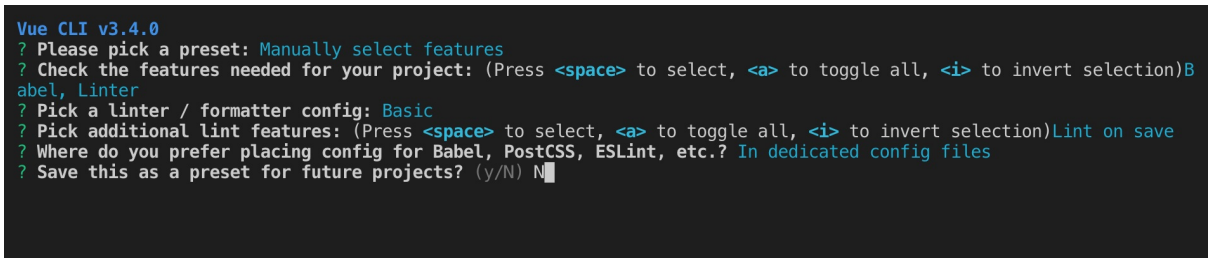
If you're using `git bash` with **Windows** you will need to run this command to create a project:

```
winpty vue.cmd create test-project
```

You can read the vue-create docs regarding running this command with `git bash`. You can add an alias to your `~/.bashrc` file by pasting in `alias vue='winpty vue.cmd`. If your `/.bashrc` file doesn't exist you can create it using the bash touch command `touch ~./bashrc`.

Once initiated, we will be presented with a choice as to whether to accept the default

... or manually make some configuration choices. The image below show manually choosing the default. You use your up/down arrow keys to select in the command line. If you choose the manual selection, you will be presented with a series of choices. The choices below will load babel and a linter.



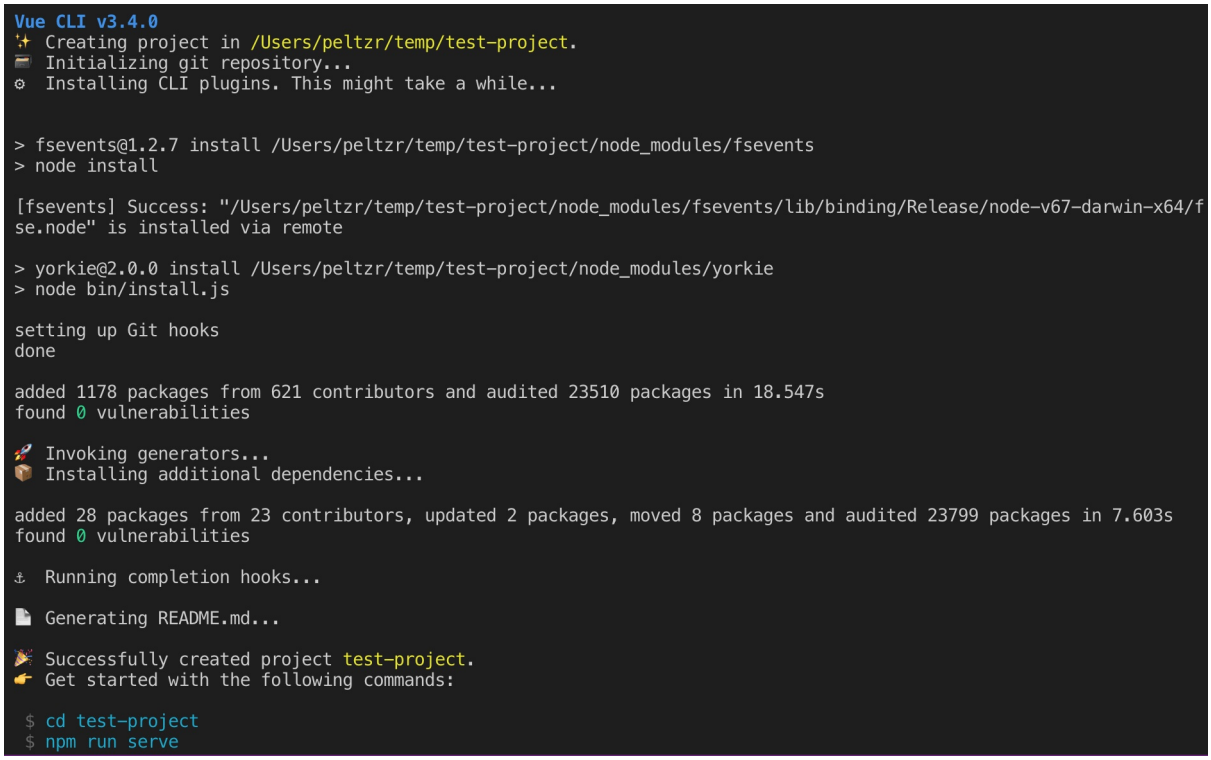
For the work in this course you will usually be able to choose the default. It's possible to modify these choices after the initialization through configuration and/or additional commands.

The default choices we are making provide the following functionality;

- `babel` — code libraries that can transpile `.vue` files which can contain HTML, CSS and JavaScript into static HTML, CSS and JavaScript files as required by the browser
- `linter` — code library that provides formatting and linting (syntax and style error detection)

If you accept the original default, you won't get a linter config file.

Here is a screenshot of what the process looks like when completed, and we can see a set of answers that will create a minimal project with no test frameworks or extras.



The results of the `vue create` command.

This process will create a new directory called `test-project/` (we can call our projects whatever we'd like). We can also see that directions for getting going with development are printed after the command finishes. Change directory into the `test-project/` directory and we should be able to see all the files created for us.

Inside the repository, we should see the following files and directories:

```
becky-mac:temp peltzr$ cd test-project
becky-mac:test-project peltzr$ ls -la
total 856
drwxr-xr-x   14 peltzr  staff     448 Feb  8 15:18 .
drwxr-xr-x   12 peltzr  staff     384 Feb  8 15:18 ..
-rw-r--r--    1 peltzr  staff      33 Feb  8 15:18 .browserslistrc
-rw-r--r--    1 peltzr  staff     353 Feb  8 15:18 .eslintrc.js
drwxr-xr-x   13 peltzr  staff     416 Feb  8 15:18 .git
-rw-r--r--    1 peltzr  staff     214 Feb  8 15:18 .gitignore
-rw-r--r--    1 peltzr  staff     365 Feb  8 15:18 README.md
-rw-r--r--    1 peltzr  staff      53 Feb  8 15:18 babel.config.js
drwxr-xr-x  799 peltzr  staff   25568 Feb  8 15:18 node_modules
-rw-r--r--    1 peltzr  staff  406634 Feb  8 15:18 package-lock.json
-rw-r--r--    1 peltzr  staff     517 Feb  8 15:18 package.json
-rw-r--r--    1 peltzr  staff      59 Feb  8 15:18 postcss.config.js
drwxr-xr-x    4 peltzr  staff     128 Feb  8 15:18 public
drwxr-xr-x    6 peltzr  staff     192 Feb  8 15:18 src
```

We can see that we have some configuration files, including a `babel.config.rc` that sets up how the Babel will work for us. There is also a `eslintrc.js` that configures how the linter will work. We also have a `package.json` file that lists all the Node.js modules our application depends upon. Inside the `src` directory is the actual content of our

application. There is also a `public` directory that will contain static media files but which do not require post-processing by the build tools but that need to be packaged with the site and the `index.html` file that will be served to the browser.

Over the next several sections we will look at many parts of this project skeleton and examine the features of this project template. For now, it's important to follow the remaining steps to get the project ready for development.

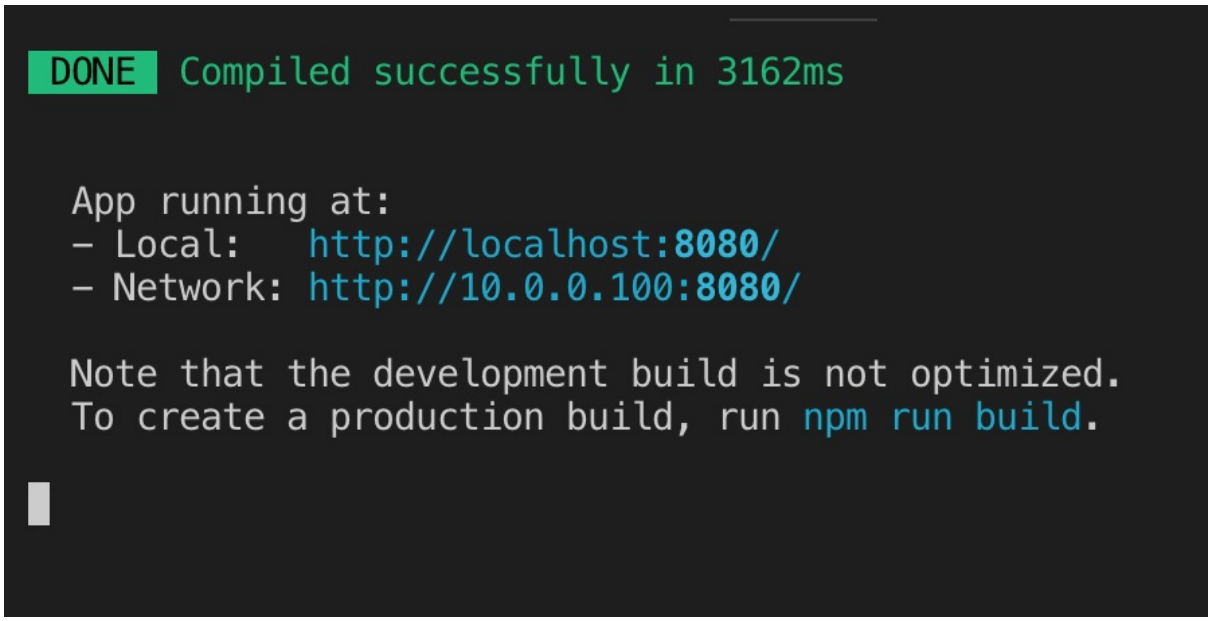
The create process has run `npm install` for us and there is a node_modules directory created. There are also `.git` directory which means we have a local git repo and a `.gitignore` file which contains directories and files to exclude from git.

Run the following command to install all of the Node.js modules our site depends upon:

Now that we have everything installed to run our project, we can test it out by running the development server with this command:

```
npm run serve
```

The output of the serve command is show below. A server using the 8080 port is created and it show using localhost and the ip address on your network. This ip address allows you to share your dev code with anyone running on your network. This is a live server in that, if you view it in your browser, modifying the code will automatically update the browser.



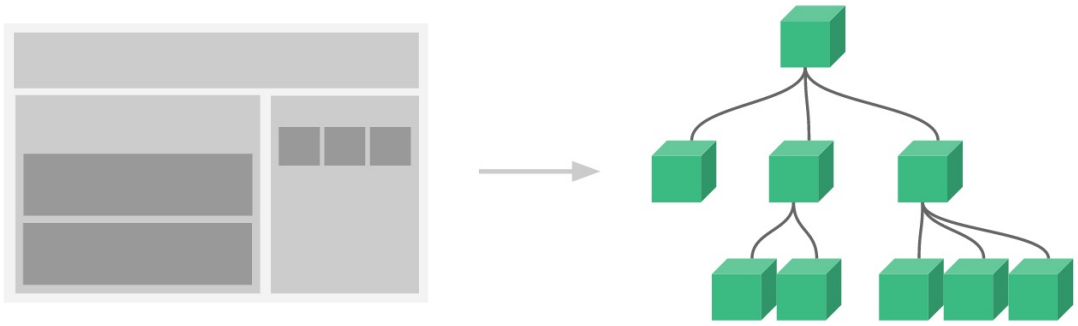
We will want to keep that command handy, because that is how we will run the development server whenever we want to do work. The development server will run while we are working and will automatically refresh the page when we make changes to our files. It will also alert us to many issues that might come up in our code as we develop. You might have noticed that the last part of the output created by the `vue create` command reminds you that you can change directory to your new project and run `npm run serve`.

Remember from the earlier exercise that this server will run until you press CTRL-c to end it.

Once we have the site up and running on the development server, we can poke around and get to know our Vue.js app a little better.

Getting to Know Vue

Vue.js is an application framework primarily based on the concept of Web Components. It is designed to be approachable for newer developers, but it allows for extension to become as feature rich as any other application framework out there. This approach is mirrored at many levels within the Vue.js project: The system itself is meant to be lean and then augmented by adding additional modules. And the projects we build with Vue.js are meant to be composed of simple components that combine to create more complex applications.



Vue.js system illustration

As we can see in the illustration above, everything we see in the interface of our application is related to a specific Component within our application code. A "Web Component" is a self-contained set of HTML, CSS, and JS functionality that is revealed through a unique, custom HTML tag. In the illustration, we can see the wireframe representing components on the page. Those components are related within the logic of the Vue.js application, too, which is what is shown in the hierarchical diagram to the right.

Lego Simpsons Minifigs Parts (All rights belong to owner.)

This is probably the correct place to bring up a metaphor based on a favorite modular toy. If we imagine that a framework is like a box of Legos, for example, then each block type would be a Component. We could build a Lego person by combining a head block, body block, and legs block. With Legos we even have modular hair blocks that we can customize to get the character we want.

Our Vue.js applications are sets of special blocks that we can put together to create the unique experience we need. We can create our own Components, or we can use Components created by others. Just like with the Lego figures, we can achieve a vast array of different effects by varying the elements that build our entire application.

It can be tricky to really understand how components in Vue.js applications work together. So let's take a closer look at the code we just generated in the project skeleton.

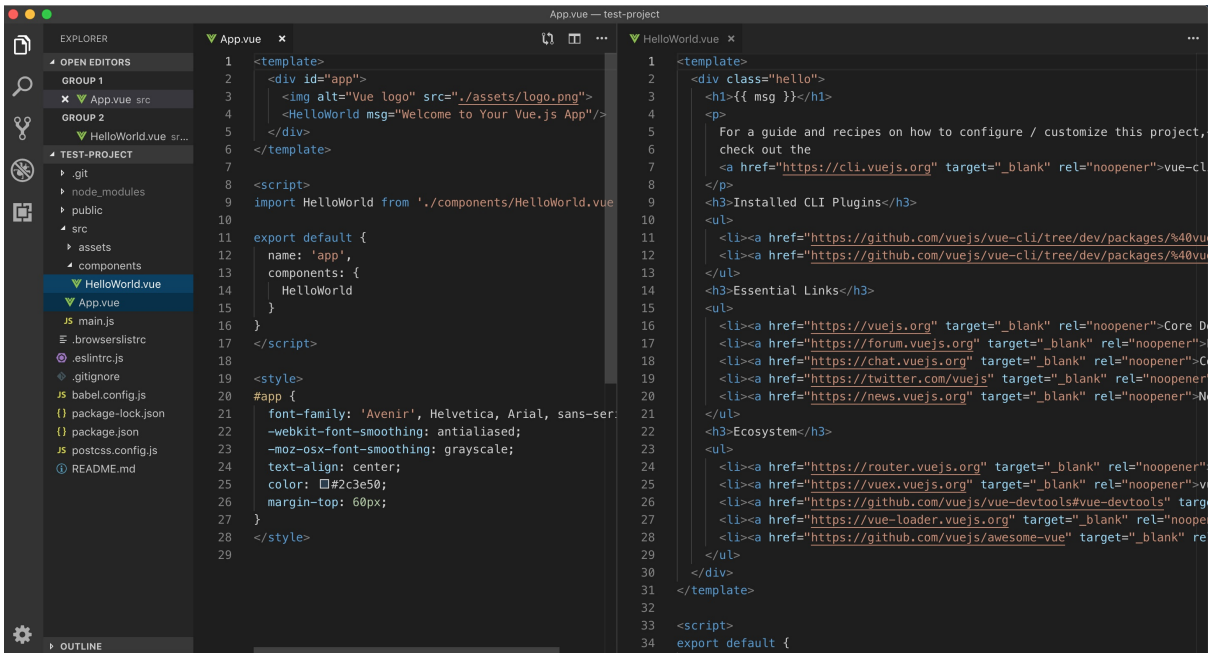
Project Components

Inside the `src/` directory of our project is where all of the code that makes our project unique lives. If we look inside, we see the following:

```
drwxr-xr-x   6 shawnr  staff   204B Sep 24 13:09 .
drwxr-xr-x  15 shawnr  staff   510B Sep 24 15:11 ..
-rw-r--r--   1 shawnr  staff   446B Sep 24 13:09 App.vue
drwxr-xr-x   3 shawnr  staff   102B Sep 24 13:09 assets
drwxr-xr-x   3 shawnr  staff   102B Sep 24 13:09 components
-rw-r--r--   1 shawnr  staff   320B Sep 24 13:09 main.js
```

Looking at this listing, we can see a `main.js` file and an `App.vue` file. The other two items listed are directories: `assets` and `components`. Inside `main.js` is the code that instantiates a Vue.js application. It imports the `App.vue` file, as well as the core Vue.js framework. The result of this code is that the `App` component is inserted into the `index.html` file (in the root of our project repository) and then executed.

When the `App` component is executed, it runs the code contained in `App.vue`. This code defines a `<template>` tag, a `<script>` tag, and a `<style>` tag. When working with Vue.js components, we keep our HTML, JavaScript, and CSS in one location. Our CSS is automatically scoped so it will only affect the specific component, which helps prevent issues of CSS overlapping between components. The logic that makes the component function is included between the `<script>` tags, and that logic is applied to the template defined in the `<template>` tag.



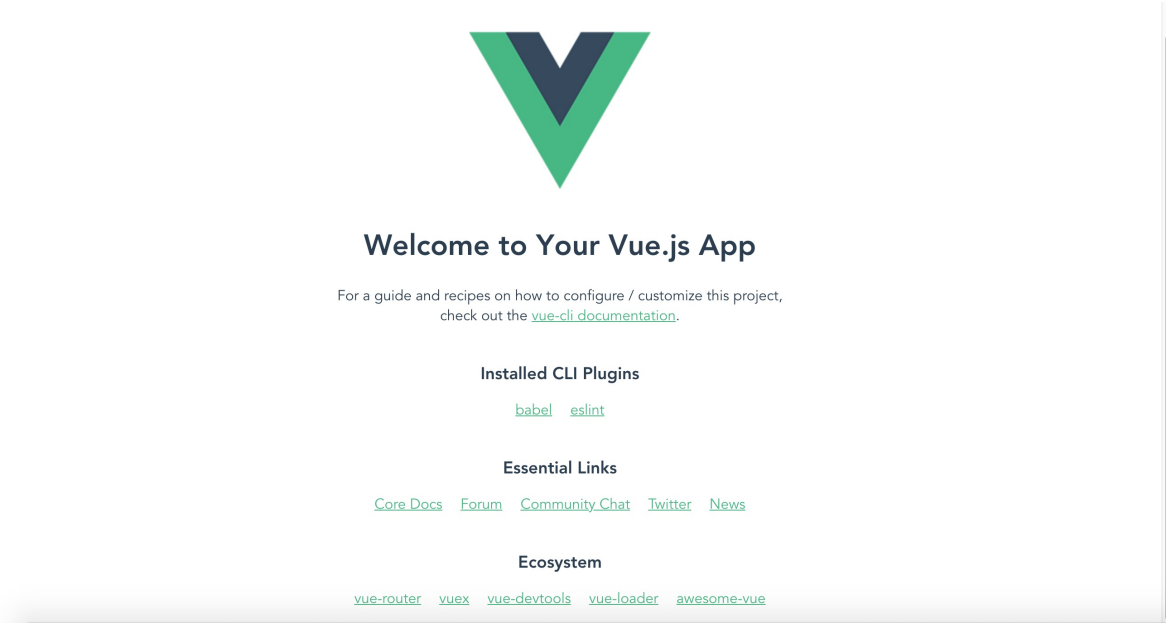
Code for App and HelloWorld Components

Inside the `components/` directory is the `HelloWorld.vue` file, which contains the `HelloWorld` component. This component is referenced inside of the `App` component. As we can see in the illustration above, the `App` components lists the child components it's using in the `components` property of the `App` object. `Hello` is the only component listed. We can also see that in `App.vue` there is an import statement:

```
import Hello from './components/HelloWorld'
```

The import statement is how we can let modern JavaScript files know about other files we are working with. Notice that this import statement gives a name to the object being imported (`HelloWorld`), and it specifies the file that is to be imported (`./components/HelloWorld`). The `.vue` extension on the file name is not needed in the import statement, although it is needed on the file itself. The import statement understands to fill in the extension.

We can see, based on the highlights in the image above, that the `HelloWorld` component is referenced in the template for the `App` component. The `<HelloWorld/>` line indicates where the content for the `HelloWorld` component should be shown. It can be difficult to imagine what this looks like when it is displayed to the user. This next screenshot should help.



Vue.js App Components

The image above shows the default page rendered when running the project skeleton. The logo and Welcome message are in the coded in the `App` component. The `HelloWorld` component is inserted inside of the `App` component. All of the links are encapuslated in the `HelloWorld` component.

Templates and Data (and Methods)

In each Vue.js Component, we will probably provide some HTML for a template. Within templates we can invoke different kinds of logic from within the script controller of the template and we can output data values to display to the user. There is a lot of power within templates, and we will explore the power of templates in greater detail soon. For now, here are a few things to keep in mind as you're looking at and playing with this new project skeleton.

Output Data Values

In a Vue.js template, we can output data values using the "double curly brace" syntax. This is often called "mustache" templates (because curly braces look like sideways mustaches). Here is an example from the `HelloWorld.vue` file:

```
<h1>{{ msg }}</h1>
```

This example will output the value of the `msg` property defined in the `props` object. The `props` object allows a child component to receive data from its parent. Notice that the `msg` attribute contains the message that is named in the `HelloWorld` `prop` .

We'll also see a `data` object can be used to create data within a component. Most components will define some properties in their `data` function. Managing data and binding data properly to template elements is key to using templates effectively.

Directives

In addition to just displaying data, templates can contain some logic. This logic is made available through the concept of "directives". Directives in Vue.js look like HTML attributes. They are written in the same style as HTML attributes in HTML tags. Here is an example:

```
<ol>
  <li v-for="todo in todos">
    {{ todo.text }}
  </li>
</ol>
```

We can see that the `v-for` directive is being used to create a loop. The loop will duplicate the list item for each `todo` in the `todos` Array. (Don't worry if this seems confusing now. We will look more closely at using loops and other directives in an upcoming section of the book.)

Another important directive is `v-if`, which allows us to write a conditional statement like this:

```
<div v-if="showResults">
  ... show some information ...
</div>
```

In addition to directives that allow looping and conditionals within a template, there are also directives that allow us to respond to events. The `v-on` directive allows us to specify an event and the "method" (function) that will be invoked when the event occurs. Here is an example:

```
<button v-on:click="refreshData">Refresh Results</button>
```

In the example above, we have a button that will execute the `refreshData` method defined in our component whenever the button is clicked. (Again, this is a lot to understand without additional information, but the goal here is just to begin recognizing what these parts do. We will learn more about them soon.)

Using all of these core template features, plus some of the other features provided by the Vue.js framework, we have the ability to create dynamic interfaces and useful functionality that our users will appreciate.

Quiz: Bootstrapping a New App

Please enjoy this self-check quiz to help you identify key concepts, points, and techniques discussed in this section.

Visit Quiz Online

The quiz on this page has been removed from your PDF or ebook format. You may take the quiz by visiting this book online.

Project: Bootstrap an App

To practice with making an application using the Vue-CLI tool, let's bootstrap an app that we can work with in sections 5 (Debugging the App) and 6 (Deploying the App) to practice working with the project skeletons we can create.

Check for Vue-CLI

If we have Vue-CLI installed properly, then we can run the command `vue --version` from anywhere on our command line and see the version of the Vue-CLI that we have installed. We should see a number at or above `3.4`. If we see an error (something about the `vue` command not existing), then we should install Vue-CLI by running this command:

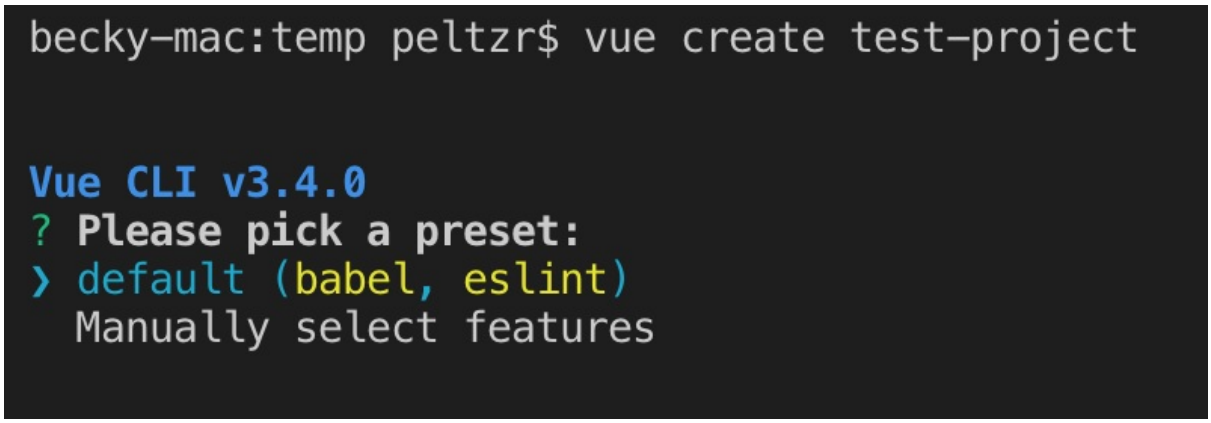
```
npm install -g @vue/cli
```

Bootstrap the Application

Once we have Vue-CLI properly installed, we can bootstrap our project skeleton. This will create all the files we need to get started writing our application. Create a new app using the `webpack` template:

```
vue create test-project
```

Answer the questions like we see in the screenshot below:



The results of the `vue create` command.

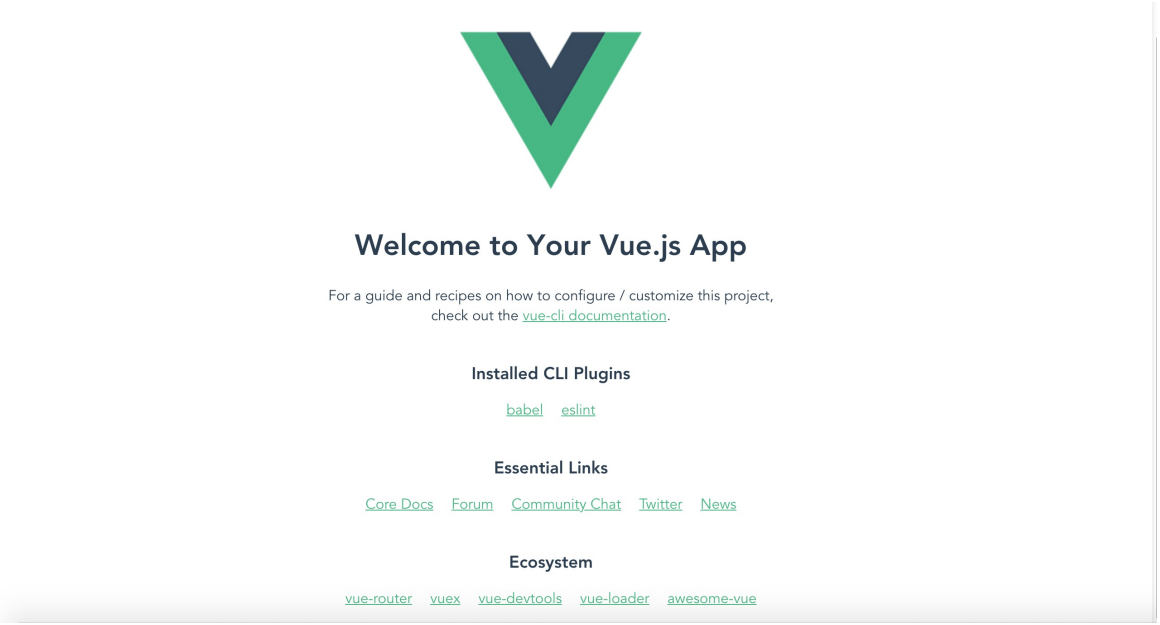
Once the project skeleton is available, `cd` into the directory where your project was created.

Start the Dev Server

Once the installation is complete, we can test the project by running:

```
npm run serve
```

We should see the development server start up. You can CMD (Mac)-click or CTRL (Windows)-click to open the browser.



Default screen from Webpack project template

If we see a screen that looks like the one above, then we have successfully installed our dependencies and our project is up and running. Now that we have a working project, we can begin to explore some of the parts of the application and see how they work together. The next steps are meant to expose us to different aspects of the software, but rest assured that we will cover these concepts and techniques in more depth in future sections.

Modify HelloWorld

In the default application, there are two Components at play: `App` and `HelloWorld`. In order to experiment with the application, we will modify the `HelloWorld` component. Open the file `src/components/HelloWorld.vue` and look at the parts. Each `.vue` file is broken into three main areas: The template, the scripts, and the styles. These are denoted by corresponding tags.

The Template

Let's modify the template code to reflect our own content. Here is the original template code from `src/components/HelloWorld.vue`:

```
<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
    <p>
      For a guide and recipes on how to configure / customize this project,<br>
      check out the
      <a href="https://cli.vuejs.org" target="_blank" rel="noopener">vue-cli documentation</a>.
    </p>
    <h3>Installed CLI Plugins</h3>
    <ul>
      <li><a href="https://github.com/vuejs/vue-cli/tree/dev/packages/%40vue/cli-plugin-babel" target="_blank"
rel="noopener">babel</a></li>
      <li><a href="https://github.com/vuejs/vue-cli/tree/dev/packages/%40vue/cli-plugin-eslint" target="_blank"
rel="noopener">eslint</a></li>
    </ul>
    <h3>Essential Links</h3>
    <ul>
      <li><a href="https://vuejs.org" target="_blank" rel="noopener">Core Docs</a></li>
      <li><a href="https://forum.vuejs.org" target="_blank" rel="noopener">Forum</a></li>
      <li><a href="https://chat.vuejs.org" target="_blank" rel="noopener">Community Chat</a></li>
```

```
      <li><a href="https://twitter.com/vuejs" target="_blank" rel="noopener">Twitter</a></li>
      <li><a href="https://news.vuejs.org" target="_blank" rel="noopener">News</a></li>
    </ul>
    <h3>Ecosystem</h3>
    <ul>
      <li><a href="https://router.vuejs.org" target="_blank" rel="noopener">vue-router</a></li>
      <li><a href="https://vuex.vuejs.org" target="_blank" rel="noopener">vuex</a></li>
      <li><a href="https://github.com/vuejs/vue-devtools#vue-devtools" target="_blank" rel="noopener">vue-devto
ols</a></li>
      <li><a href="https://vue-loader.vuejs.org" target="_blank" rel="noopener">vue-loader</a></li>
      <li><a href="https://github.com/vuejs/awesome-vue" target="_blank" rel="noopener">awesome-vue</a></li>
    </ul>
  </div>
</template>
```

We can see from the page in the browser that this code is creating most of the content on the page. Let's alter that content:

```
<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
    <h2>2 Things that are difficult in JavaScript</h2>
    <ol>
      <li>naming things</li>
      <li>recursion</li>
      <li>off-by-one errors</li>
    </ol>
  </div>
</template>
```

These changes result in the following changes in the browser:

After changes to the template

Vue.js component templates can have any HTML in them. We can create whatever structures we need, and they can even include other component tags (as with the `src/App.vue` file, which uses the `HelloWorld` component in its template). Any HTML that shows up between the `<template>` tags will be inserted into the app when this component is executed.

The Styles

Inside each Vue.js component is also a style block, defined by the `<style>` tags. We can see in the screenshot above that the list items are not numbered and they are laid out horizontally. Let's replace the numbers and make them go vertical again.

Here is the original code:

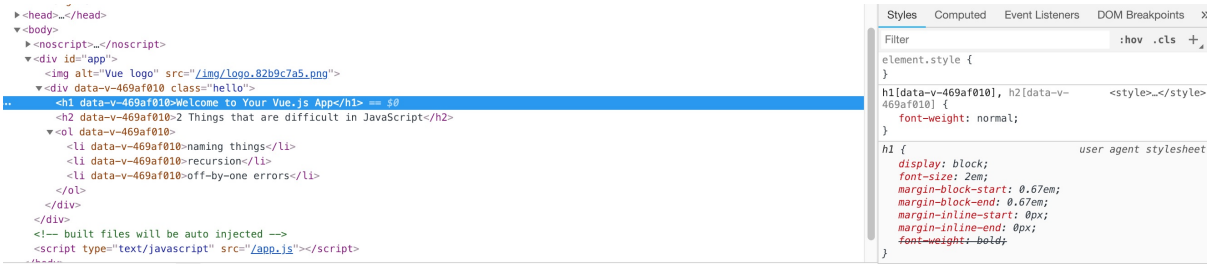
```
<style scoped>
h3 {
  margin: 40px 0 0;
}
ul {
  list-style-type: none;
  padding: 0;
}
li {
  display: inline-block;
  margin: 0 10px;
}
a {
  color: #42b983;
}
</style>
```

The `scoped` attribute on the `<style>` tag insures that these styles will not apply to anything outside of the component itself. This is very handy on large projects where overlapping styles can be problematic. Because of this tight scoping, we can approach each component on its own terms, name the parts of the component in a way that makes sense, and generally pursue a more modular approach to styles.

In order to change the list the way we want, let's make these changes:

```
<style scoped>
h1, h2 {
  font-weight: normal;
}

ol {
  list-style-type: decimal;
  width: 40%;
  margin: auto;
}

li {
  display: list-item;
  margin: 0 10px;
}

a {
  color: #42b983;
}
</style>
```

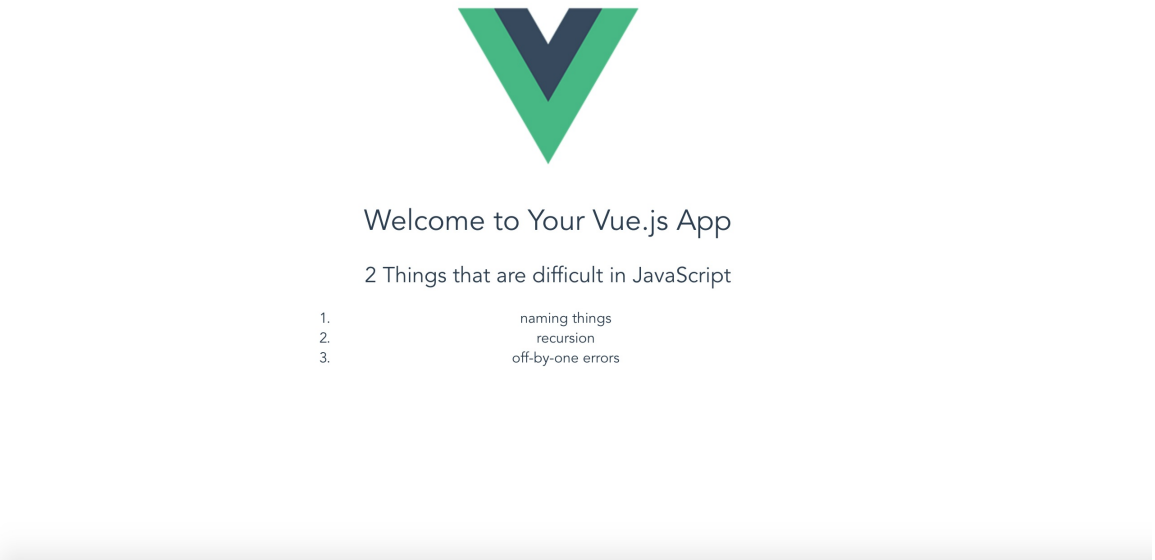
There is nothing special about these styles, but it's interesting to note that if we inspect our styles in our developer tools, we can see how the styles are scoped to the specific component using the `data` attribute and attribute selectors to implement scoping :



Viewing the scoped styles in developer tools

By using the attribute selector these style definitions are sure to never apply to any other elements on the page. So if we write a style for `p` or `div` or `ul` it will only apply to those elements when they show up inside this specific component template.

Here is what our page looks like after we make these style changes:



The list after style changes

These are not the most amazing styles, but hopefully they help bring how things work into focus.

The Logic

The last part of the component that we have to explore is the logic itself. For the most part, this logic is pretty simple when it is generated in the project skeleton. The script tags in the default `HelloWorld` component contain the following code:

```
<script>
export default {
  name: 'HelloWorld',
  props: {
    msg: String
  }
}
</script>
```

props

This logic does not do too much except define the `props` function with an object that contains the `msg` property. The `props` object is what gets revealed to the template context for processing.

data

We can add a `name` property to this component using the `data` function. The data function must return an object which can contain properties (key:value pairs). Any property of the `data` object is accessible as a variable inside the template. The `name` property can be added to the `HelloWorld` component template to create the content of the `<h3>` tag:

```
<script>
export default {
  name: 'HelloWorld',
  props: {
    msg: String
  },
  data() {
    return {
      name: 'Shawn'
    }
  }
}
</script>
```

Then add an `<h3>` header to the template.

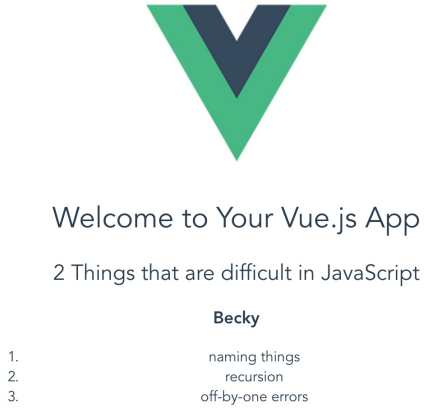
```
<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
    <h2>2 Things that are difficult in JavaScript</h2>
    <h3>{{ name }}</h3>
    <ol>
      <li>naming things</li>
      <li>recursion</li>
      <li>off-by-one errors</li>
    </ol>
  </div>
</template>
```

Notice how we use the double curly braces (`{{ variableName }}`)to output the value of a variable in a template. This is a common convention among templating languages, especially in JavaScript frameworks.

We can change the name by altering the definition of the `name` property in the script. Replace `<put your name here>` with your own name.

```
<script>
export default {
  name: 'hello',
  props: {
    msg: String
  },
  data () {
    return {
      name: '<put your name here>'
    }
  }
}
</script>
```

Once we have made that change to the data being piped into our application, we can see the change in the browser:



Altered H1 content

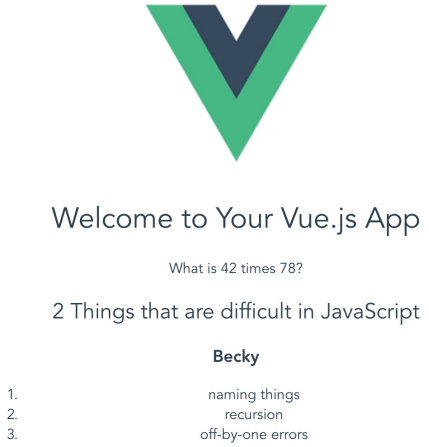
We can even add additional data to the object and then refer to those variables in our template. First, we update the script:

```
<script>
export default {
  name: 'hello',
  props: {
    msg: String
  },
  data () {
    return {
      name: 'Becky',
      num1: 42,
      num2: 78
    }
  }
}
</script>
```

Then we update the template by asking about multiplying 2 numbers:

```
<template>
  <div class="hello">
  <h1>{{ msg }}</h1>
  <p>What is {{ num1 }} times {{ num2 }}?</p>
  <h2>2 Things that are difficult in JavaScript</h2>
  <h3>{{ name }}</h3>
  <ol>
    <li>naming things</li>
    <li>recursion</li>
    <li>off-by-one errors</li>
  </ol>
  </div>
</template>
```

This results in the following display in the browser:



New variable in the template

Of course, now that we have those new variables in the template, we can try doing even more.

Event Handling

Let's finish out exploring this application by going the extra mile and adding a button to calculate `42` times `78`. To do this, we will define an event listener on a button. We will do that in the template. Modify your template code to match this:

```
<p>What is {{ num1 }} times {{ num2 }}?<span v-if="product">{{ product }}</span></p>
<button v-on:click="calculateProduct">Calculate</button>
```

We have used a couple of new things here. First, we have added a `<button>` element to the template. This button uses the `v-on` directive to define a `click` event handler. There are many other events we could handle, but we will begin with this simple click. The `v-on` directive then specifies the name of the method that will be executed when this event is detected. So when the user clicks the button, the `calculateProduct` method will be executed.

We have also added a `<span>` tag that contains the "product" (the result of multiplying `42` times `78`). The `<span>` contains the `v-if` directive, which is a conditional statement. If the `v-if` evaluates to `true`, then the `<span>` and its contents are shown. If not, then the `<span>` and its contents are hidden. This will allow us to not show anything until the answer is populated by our application.

Now that we've added that code to the template, we must update the script logic. Here is the updated script logic:

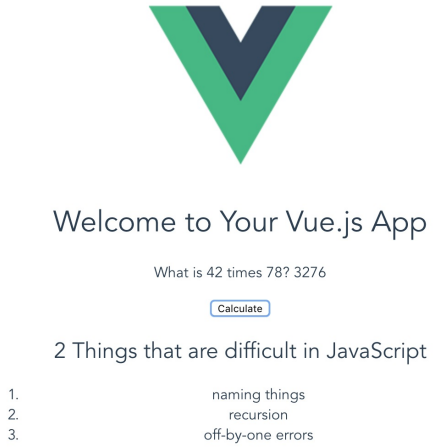
```
<script>
export default {
  name: 'hello',
  props: {
    msg: String
  },
  data () {
    return {
      name: 'Becky',
      num1: 42,
      num2: 78,
      product: null
    }
  },
  methods: {
    calculateProduct: function(){
```

```
        this.product = this.num1 * this.num2;
      }
    }
  }
</script>
```

As we can see, we have added another property to the data object for `product`. This will allow us to refer to that value in our templates and our logic. We will initialize it to `null` since that will evaluate to a "false" in the `v-if` conditional.

We have also added another property called `methods` to our component definition. The `methods` property contains an object that is populated by named methods. These methods can be executed from within component logic or templates. In this case, we are defining a method called `calculateProduct`, which multiplies the `num1` and `num2` values together. Note that within methods we use the `this.variableName` syntax similar to the syntax used in ES6 Class methods.

Once we put this script in place, we can try our page in the browser and see the result:



Calculation performed

We can see that the calculation has been successfully performed in the screenshot. We should be able to see no product when the page loads, and then have the product populated when we click the button. If all of this is working then we have successfully altered our first Vue.js project skeleton, and we are ready to move on to learn methods for debugging and deploying our apps.

Changes Repository

If you need to look at a full set of code to see how all the changes described above can go together, please review this project repository on Github.

Debugging the Application

One of the most important skills to develop in web development is the skill of debugging. This is the skill of tracking down what exactly is not working and then fixing it. Debugging is difficult to learn, and difficult to teach. Much of our expertise in debugging comes from experience. Part of why debugging is so difficult is that we are really trying to solve two problems.

When we notice a problem in a software project, we must be able to make that problem occur on purpose. If we cannot reproduce a problem, then we cannot ever test that we have actually solved the problem. This is often murky area in software development: People see intermittent problems and blame them on the wrong things. As developers, we can only fix the problems we can identify and reproduce, so we know that we are actually fixing the correct issue.

Once we've determined a way to reproduce a bug and we know exactly how to cause it to happen, we must then solve the second problem: Actually fixing the bug. Sometimes this is the easiest part. It can be difficult to figure out how to cause a bug to happen, but then it may turn out that the bug is simple to fix once it has been identified. Other times the error itself is quite complex and difficult to fix.

However we approach debugging, it's a crucial step in software development. Spending time truly solving problems in our applications will help us become much better developers.

Core Concept: Debugging Techniques

Anytime we write code, we expect to encounter errors. In code, we call errors "bugs" and we call the process of tracking down and fixing errors "debugging". These bugs can come from syntax errors (when we misplace a comma, semicolon, brace, etc.) or from logical errors (when we make a mistake in calculating data or accessing a variable). Syntax errors are often the most frustrating for new developers because they consist mainly of things that are difficult for the untrained eye to spot. We have great tools for helping us find syntax errors, and once we get used to a language, those errors are not nearly as frustrating as they are to begin with. However, we still face the tricky task of figuring out our logical errors. Logical errors often require a more nuanced approach to debugging.

The process (and art) of debugging is enough to fill an entire book on its own, but we will review some methods that will often help find and fix bugs in our code.

Always Look at the Console

Whenever something isn't working properly in our JavaScript, we should always open up the developer tools in whatever browser we're using and check out the JavaScript console. The console will usually show what errors have been triggered by the page, and they will often give us a specific filename and line number where the error has been found. The console is an incredibly useful tool for developing JavaScript, so don't forget to consult the console (and our other developer tools) whenever something goes wrong.

Syntax Errors

Syntax errors consist of errors that are caused by some bad text in our code. Here are some common syntax errors:

```
let myString = 'Some text here; // Syntax error: no closing quote.
let myArray = [1, 2, 3,]; // Syntax error: trailing comma in Array definition.

if (x < 12) {
    console.log(x);

// Syntax error: conditional statement has no closing brace.
```

These errors are annoying, but they tend to be the easiest to fix.

Check the Console

Like the tip above says, we always begin by checking the console. In the case of syntax errors the console will usually lead us to the specific file and line number, often with a helpful link. Sometimes syntax errors actually exist on the line before the one that triggers the error, so be sure to look around the line noted in the console if we don't find the error right away. Here are some tips for where to look to resolve syntax errors:

- Look on the line (or lines) before the one indicated in the console. For many syntax errors they register on the line after the missing quote, semicolon, etc.
- When the console indicates the last line of the file, look closely at the curly braces and how we've closed all of the code blocks.

- Go through and fix up all of our indentation and spacing. We often find syntax errors when we clean up our style, and lining up our code blocks helps figure out where we've forgotten to close things.

Prevent the Error

If we use tools like linters in our editors, we can be alerted to syntax errors before we see them in the browser. Sometimes if we see a line number indicated in an error in the JavaScript console we can return to our editor and notice a warning or error that will help track down the error. Using tools in our editor to alert us to the simplest syntax errors is a great way to avoid needless frustration. Anticipate that everyone makes typos and mistakes, so it's no shame to rely on tools to help us discover those bugs.

Logical Errors

Logical errors are much more difficult to track down and fix. They involve a mistake in our logic that leads to an incorrect value or some erroneous logical flow. Because these errors do not trigger syntactical errors, and often do not trigger any error at all in the JavaScript interpreter, they can be very difficult to track down and fix.

The first step to fixing any bug of this sort is to **reproduce the error**. It is impossible to fix a bug that we cannot reproduce, and if we cannot reproduce a bug then we can never verify a fix. If the error happens every time, then we are lucky. The toughest bugs to fix are those that only show up in very specific (and possibly rare) circumstances.

When working to reproduce a bug, try to follow these steps:

- Note exactly what happened to cause the bug.
- Come up with a set of inputs or steps that lead to the bug and write them down.
- Work with a colleague to have them reproduce the bug on their own using the steps outlined above.
- If it is impossible to reproduce the bug, be sure to capture as much information about the system and actions whenever the bug does occur.

Remember: **If we cannot reproduce a bug then we cannot fix the bug.**

Once we can make a bug happen again, we can begin to track down a fix.

Console Logging

Throughout the code in this book we have used the `console.log()` command to output information to the JavaScript console. This is a valuable tool that can allow us to emit messages to the JavaScript console to track the movement of information and logic in our programs. We can use `console.log()` to help verify that things we can't otherwise "see" are happening. Here is an example:

```
let maxNum = 20;

for (let i = 1; i<=20; i++) {
    if (i % 2 === 0 ) {
        console.log(`${i} is even`);
        // additional program logic
    } else {
        console.log(`${i} is odd`);
        // additional program logic
    }
}
```

In the example above, the `console.log()` command is used to ping some information out to the console that we can use to verify our conditional is working properly. If we start seeing messages saying things like `"3 is even"` then we know that something is wrong.

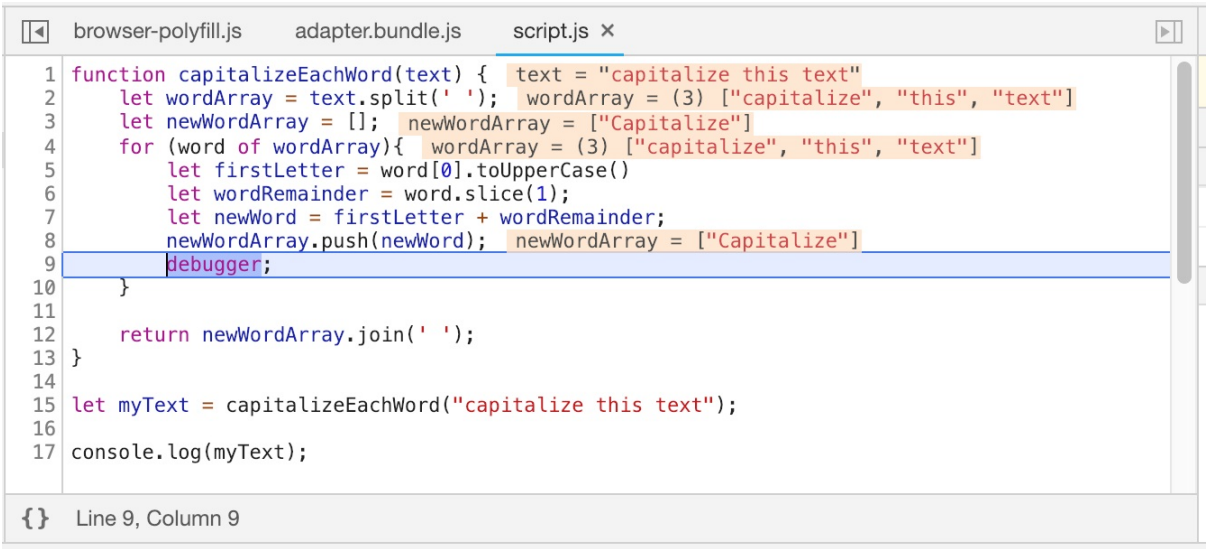
Debugger

When `console.log()` is not enough, we can use the JavaScript Debugger, which exists in the developer tools for every major browser. When we use the debugger, we invoke a "breakpoint", which is like putting a big stop sign in our code. When the JavaScript interpreter hits the breakpoint, it stops and "freezes time" in the execution of the code. This can be very useful because we can then look at all the values of the code at that moment in the execution. This is extremely helpful in figuring out what went wrong.

Exactly how this works will vary just a little bit from browser to browser. Below I've linked to suitable tutorials covering the developer tools in all of the major browsers. But by way of an example, consider this:

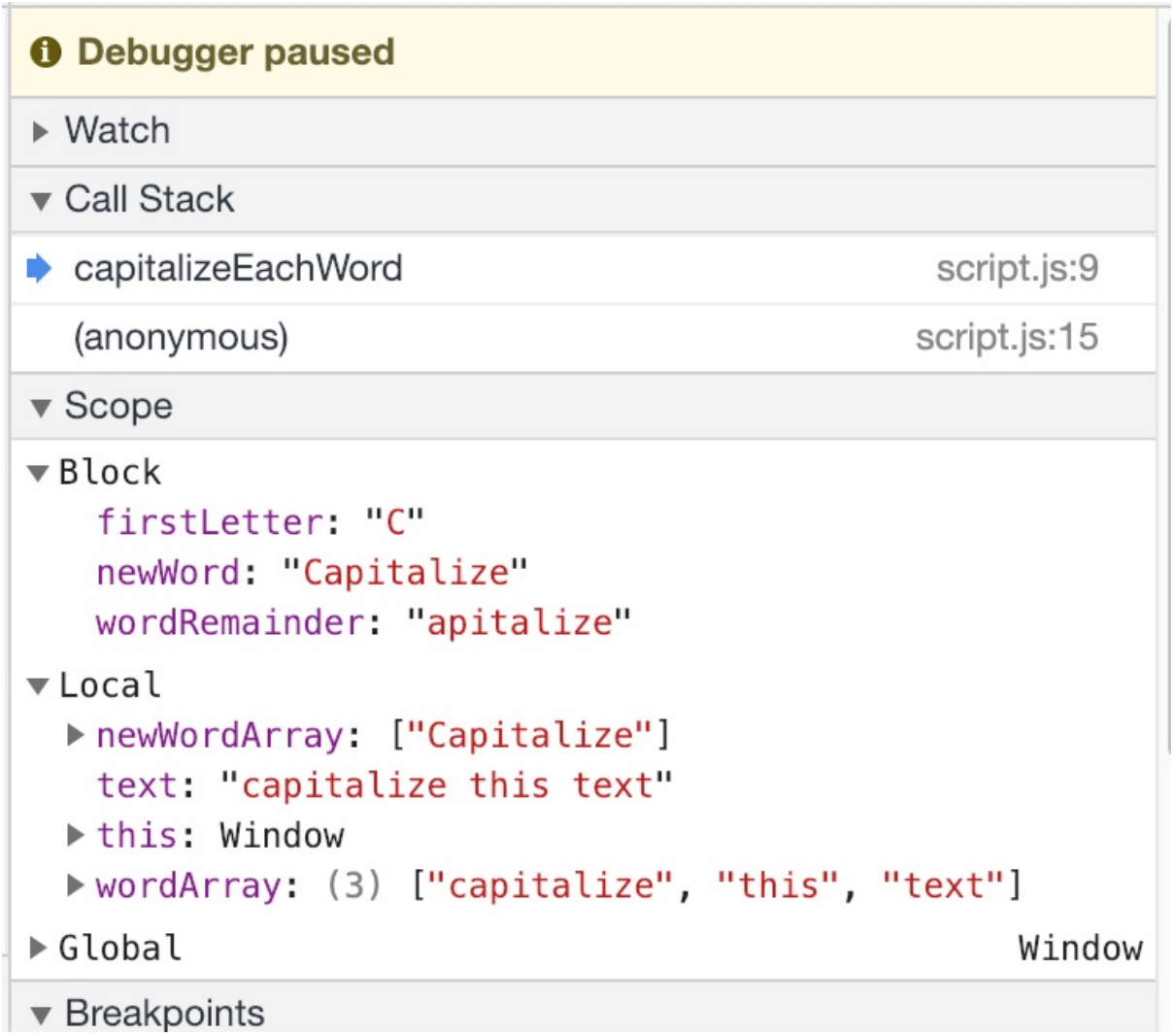
```
function capitalizeEachWord(text) {
    let wordArray = text.split(' ');
    let newWordArray = [];
    for (word of wordArray){
        let firstLetter = word[0].toUpperCase()
        let wordRemainder = word.slice(1);
        let newWord = firstLetter + wordRemainder;
        newWordArray.push(newWord);
        debugger;
    }

    return newWordArray.join(' ');
}

let myText = capitalizeEachWord("capitalize this text");

console.log(myText);
```

That code snippet has a `debugger` command in the `for` loop within the `capitalizeEachWord()` function. This will freeze the execution of the code like this:



Debugger Stopped at Breakpoint

In the image above we can see that the debugger has stopped at the line with the `debugger` command. This has caused the JavaScript debugger in Chrome Devtools to show us the current values of the variables in use around this line of code. We can take a closer look at these values in the "scope" panel:



Debugger Scope Panel

In the "scope" panel we can see the values of the variables available at the moment that the `debugger` line has been executed. We could alter these values using the JavaScript console and we can use the big "play" button at the top of the debugger window to continue executing code. Since our `debugger` command is in a `for` loop, it will pause the program each time it executes. In this way, we can watch the values change on each iteration of the `for` loop.

Learn About Dev Tools

Although we can use the `debugger` command in pretty much any browser, there is always a danger that we could forget to remove that line from our code after we've figured out the problem. This is fairly common, and, as a result, developers tend not to use the `debugger` command to set a breakpoint.

Every major browser provides developer tools of some kind, and these all allow us to set breakpoints by clicking on the line number in the code view within the developer tools. Setting breakpoints using the interface in our browser's developer tools is safer because we avoid making any changes directly to our code. It allows us to quickly set and remove breakpoints as we try to find and fix bugs. Consult the resources listed below for more information about how to get the most from our debugging tools.

Additional Resources

The amazing developers and community of people who make and use different tools to help with development and debugging also create a lot of educational resources. Check out these resources for more information about how to debug code.

Suitable for Beginners

Chrome

- Chrome Devtools Overview
- Get Started Debugging JavaScript with Chrome Developer Tools
- Pausing Code Execution With Breakpoints

Firefox

- Firefox Debugger Overview
- Debugging JavaScript

MS Edge and IE

- F12 Devtools Guide (MS IE and Edge)

More Advanced Resources

- JavaScript Debugging Reference
- JavaScript Debugging Tips

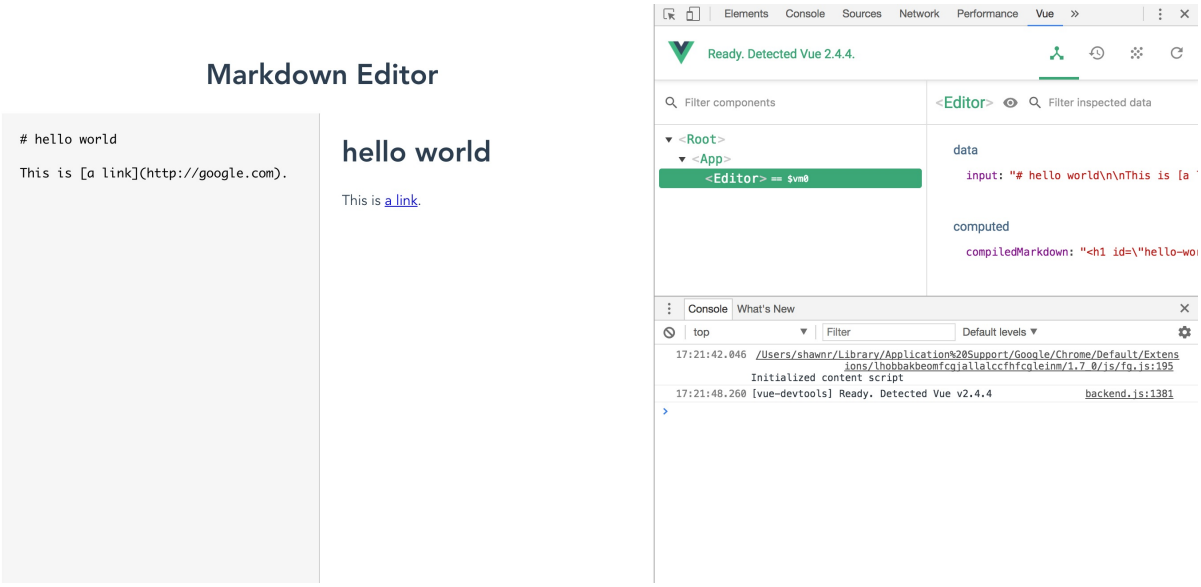
Using Vue Devtools

One of the helpful aspects of Vue.js is that there is a specialized tool called Vue Devtools that we can use to help us debug our Vue.js applications. Vue Devtools can be installed in Chrome, Firefox, or Safari (via a workaround), and it is incredibly useful for debugging applications.

Install Vue Devtools for your browser of choice:

- Chrome Extension
- Firefox Addon
- Safari Workaround

Once you have installed Vue Devtools into your browser, you can test the tool by loading your Vue project. Open the Developer Tools panel in your your browser and you should see the Vue Devtools tab available. (Consult the documentation for the specific browser you are using if you have trouble finding this tab.)



Vue Devtools in action.

When we bring up the Vue Devtools tab, we can inspect the Vue application to see the current values of all the components we've defined in our application. In the example screenshot above, we can see that we have one component (the Editor) and it has a single data value. This data value can be changed by typing into the editor textarea, and the changes will be computed and reflected on the right side of the screen. We can see how the Vue Devtools information updates in real-time as we type.

Inspect Data

Vue Devtools updating values as they are changed.

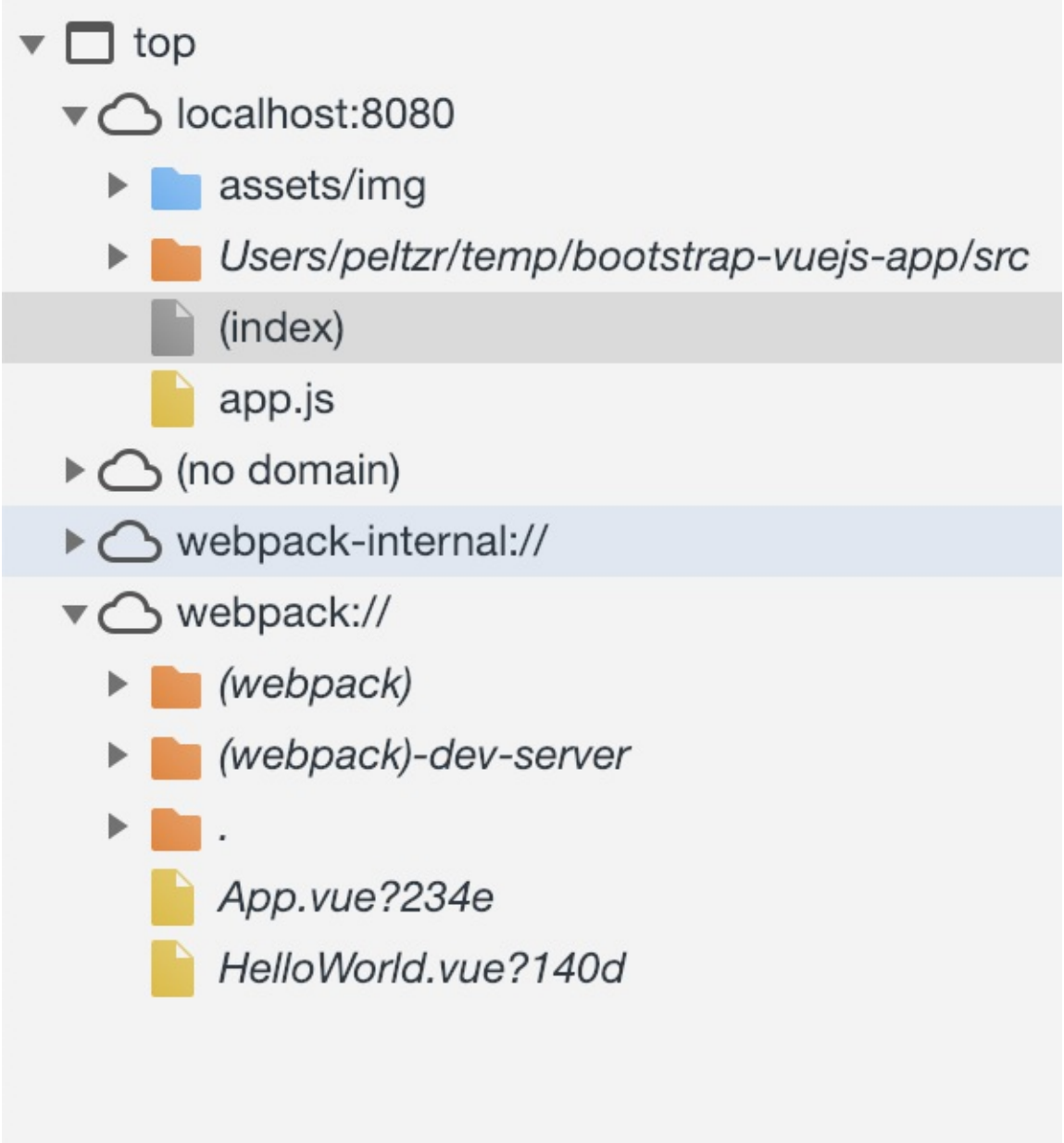
The updates to any value in the context of our application components is almost immediately reflected in our Vue Devtools tab. We can easily see how data is moving and transforming through our application, which assists with debugging and development.

The Vue Devtools offers additional insights into how our apps are functioning that will be more useful when we broaden our understanding of the framework. It will track events in real-time as they are signaled, and it will show us the full history of the state management tool, Vuex.

These are powerful features for debugging our Vue.js apps, and Vue Devtools is a major reason why developers enjoy their experience with Vue.js. We will make good use of Vue Devtools throughout this book, so be sure to get it installed and test it out with some simple projects. When things go wrong in our apps, we will want to look at Vue Devtools to help us figure out how to fix it.

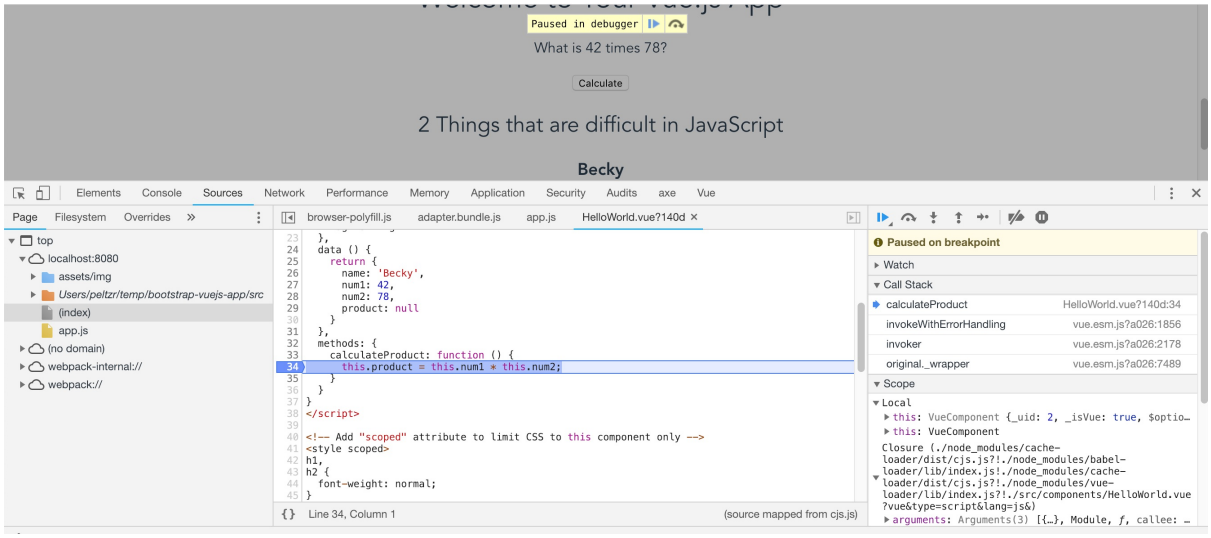
Inspect Code

With Vue Dev Tools installed you can also debug your code even though it's being served in minified form. The build process creates a `.map` file that Vue Dev Tools can use to recreate the original code so that it can be debugged during run time.



Find the original code by click on `webpack://` in the lower left pane of the Dev Tools under the Sources tab. You can drill down there to find your code as written.

You can then set breakpoints in this code. When you refresh the page or trigger an event, like clicking on a button, you can step through the code.



Clicking the `Calculate` button runs the `calculateProject` function.

Quiz: Debugging the App

Please enjoy this self-check quiz to help you identify key concepts, points, and techniques discussed in this section.

Visit Quiz Online

The quiz on this page has been removed from your PDF or ebook format. You may take the quiz by visiting this book online.

Project: Practice Debugging

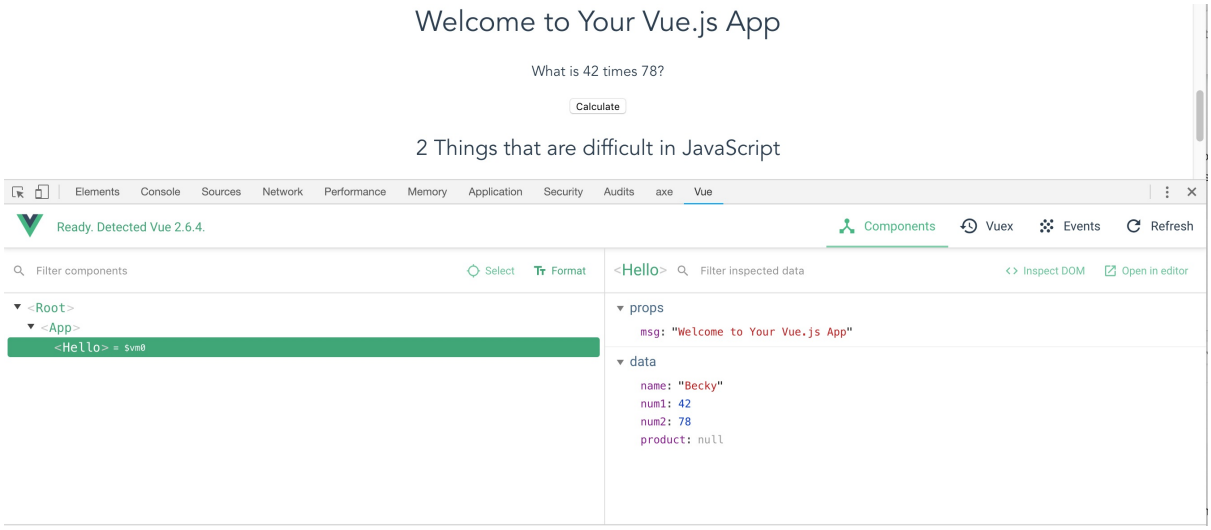
In order to practice debugging, it is useful to practice some debugging using the Vue-Devtools in our browser. We can work from the same project we used for the last section, or if we prefer we can clone out the finished version of that project from the Github repository.

Once we have the project skeleton we want to work with, we can begin exploring some of the ways we can use our debugging tools. Run the development server so we can see our project in the browser, and then open up Vue-Devtools so we can inspect our application more closely.

NOTE: You can only use the Vue-Devtools when running a project with a Dev Server.

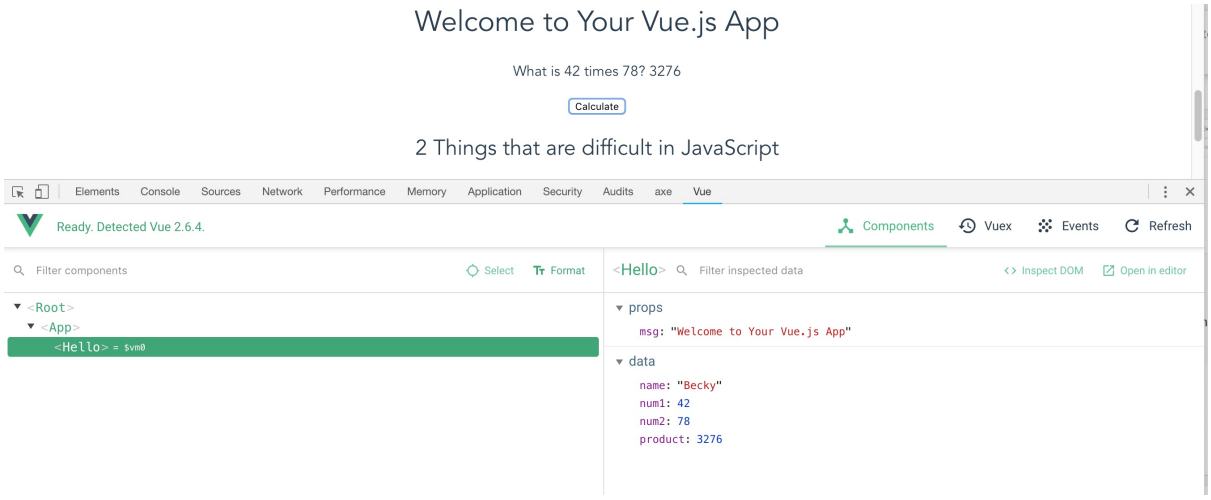
Watching Values

In the Vue-Devtools main tab we should see the components of our application. When the page initially loads the `product` value has not yet been calculated. It has been initialized to `null`. We can see that in the Vue-Devtools:



App values before product calculation

If we keep the Vue-Devtools panel open, we can watch the value change when we click the `Calculate` button:



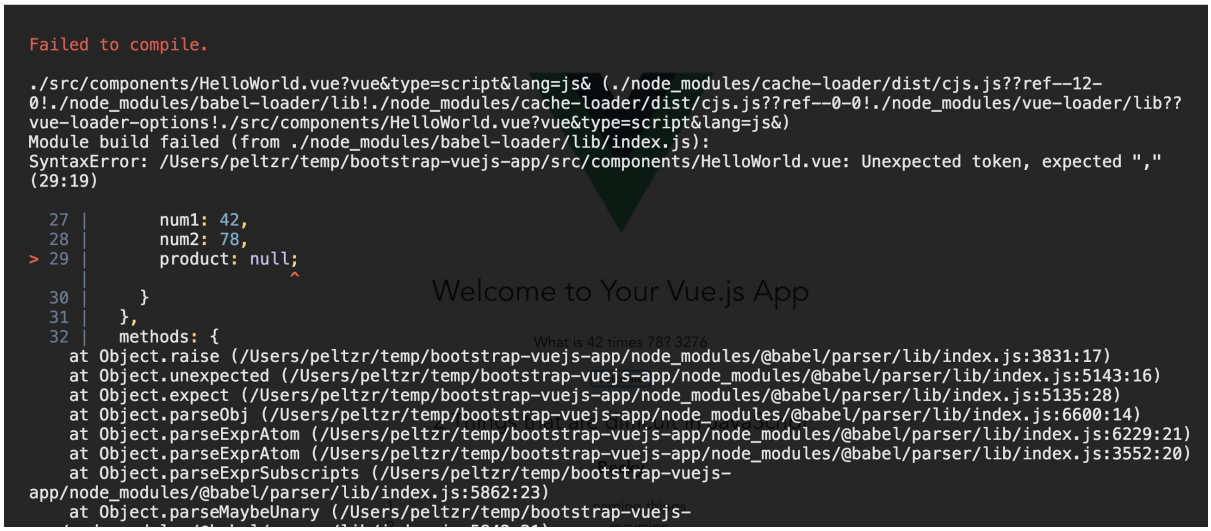
App values after product calculation

As we can see, the values that are exposed to our templates are always shown to us and updated in real-time. This allows us to watch those values change as we interact with our applications. This an immensely helpful feature as we develop new components and functions.

Catching Syntax Errors

Some of the most frustrating errors as new developers are syntax errors. These are caused by misplaced commas, semicolons, curly braces, etc. Every piece of punctuation matters to the JavaScript interpreter, and it is limited in how much it can "just figure things out" when we get fast and loose with our syntax.

If we add a syntax error to our file (in this case a semicolon has been improperly added inside the `data` object), we will get the error reflected in the web browser:



Syntax error showing in browser

This is a very handy feature of the way the development server is configured on the project skeleton. We can see that the error message actually makes it very clear where the syntax error has occurred. The line and column numbers are given in text (`SyntaxError: Unexpected token, expected , (29:19)`), and then the actual lines of code are shown with a red caret (`^`) indicating the position of the error.

Not all syntax errors are as obvious as this one, but the ability to see these errors made prominent in the web browser will help us be alerted to mistakes we make and correct them more quickly.

Deploying the Application

As we write code, and once we're ready to "release" our application, it's crucial to deploy our code to a production server for testing and observation. This accomplishes several goals: It allows us to share our work with others; it lets us make sure everything works when it is compiled and sent to the server; it lets us to collaborate with more colleagues to test and improve the application.

There are many ways to deploy an application. We could move files to a location using a manual tool like SCP or SFTP. We could use a version control software to clone a repository or branch to a specific location. There are complex build and deployment systems that require us to write custom scripts. We could package our code up into containers and use various container management tools to get our code to the users.

In the end, which deployment strategy is correct for our project will depend on our team, our requirements, and our resources. There is no one solution that will work for everyone and every project. In this section we will explore some concepts behind deploying applications, what happens when we build and deploy our webapps, and how we can actually get the projects from this book up and online for the public to access.

Core Concept: Deployment Tooling

There are many ways to get a website or application online and available for view by the public. Depending on the complexity of our project and the resources at our disposal, our deployment strategy could be very different. Regardless of how we get the code online, we also need to think about exactly what code is going to be delivered to our users. Although the simple methods we begin learning work OK for educational purposes, it's generally important to adopt more robust methods of deploying our work for professional products.

Before Deployment: Building and Packaging

Before we deploy our sites, we must first "build" and "package" the code. Sometimes this might be referred to "compiling" the code. All of these terms describe the process of putting together the source files we use in development and processing those files to be optimized for the user.

This means that each individual `.vue` file in the project is going to be broken apart and turned into JS, HTML, and CSS. These files will also be "minified" and possibly "uglified", processes that remove all of the extra whitespace and even replace names with one-letter abbreviations. These processes are performed to get the file size of the combined files as small as possible.

In addition to processing all of the code we've written for our projects, our build process must also gather and combine all of our dependencies. Those dozens of modules in the `node_modules` directory are combined into a tiny package to deliver to our users. It is often larger than the code we write ourselves, as the build report may show.

Once all these files are combined and minified, they must be renamed with version numbers so we can better control the caching of our files by our users' browsers. We don't ever want to have to tell a user to "clear the cache" and try again. That process of renaming is applied to all of the generated JS and CSS files, as well as any static media used in the site (images, etc.). When all the files have been renamed, then another pass must be made through the HTML so references to files can be properly updated.

This whole process is quite complex. It is often handled with a combination of several different tools, each of which specializes in a specific processing or optimization. Depending on our needs and goals, we may decide to include or exclude specific steps in the process. But in the world of modern development, it's unlikely that we would forgo these pre-processing, build, and packaging steps entirely.

Deployment

Once all of our files are processed and packaged, we need to actually get the files to the server. This can happen in many different ways. In the old days we might use `scp` or `rsync` or even `sftp` to upload files manually. These days we prefer to use systems that are more integrated into the development process as a whole.

It is very common to set up our version control systems to handle deployments in certain cases. It could be that merging changes into the `develop` branch of a project automatically triggers a build to the `development` server. Or we might still manually manage things by logging into the `staging` server and pulling the latest changes to the `master` branch for testing.

There is no way to properly summarize all of the ways in which deployment can be managed. The world of web operations and infrastructure management continues to grow rapidly, so it would not be unusual to have other systems such as containers, Docker, or Kubernetes involved in deployments. Tools ranging from Ansible to Salt Stack to Fabric, Chef, and Puppet have all been popular options for controlling the movement of files to a server (and the provisioning of those servers).

The rule of thumb that we will cleave to in this book is that we want to keep deployment simple and reliable so we can deploy our code repeatedly through the development process. We should be able to set up deployment of our static websites (that's what we're building in this book) as easily as possible, and we want to be able to deploy anytime we make changes.

Aprés Deployment

Many systems have tools that kick off after deployment. It's not unusual to have a continuous integration that will run tests against the newly deployed code. It might be desirable to have a system that will notify team members that a deployment has happened. Some organizations want to gather a full set of performance benchmarks every time new changes hit production.

There are many reasons why we would need to run something after deployment, and it is common to do so. In our case here, **we will be deploying to Github Pages**, which is a popular static website hosting service. (It is also free, so it's a great place for new developers to build a portfolio.) Github Pages will provide us with a status message on our project's settings page where we can see if there is an error deploying our code and how our site is configured. We could easily leverage Github's tools to add even more post-deployment tests or functions.

Configuring Deployment

To configure the deployment of our Vue.js application, we will leverage the ability of Github Pages to serve a website from the `docs` folder in a project repository. This means that we need to alter our configuration so that Webpack builds the code into the `docs/` directory instead of the `dist/` directory (which is the default location Webpack uses).

vue.config.js

To make this happen we will need to add a `vue.config.js` file that provides instructions for building to the `docs` directory. Create this file in the root of your project. The designation of `publicPath: ''` and `outputDir: './docs/'` will allow us to us deploy to a path under an account on `github.io` . Setting `assetsDir: 'assets'` means that out `css` and `js` folder will be built under the `./docs/assets` directory.

```
const path = require('path');
module.exports = {
    configureWebpack: {
        resolve: {
            //allow for @ or @src alias for src
            alias: require('./aliases.config').webpack
        }
    },
    chainWebpack: config => {
        //turn off elint for webpack transpile
        config.module.rules.delete('eslint');
    },
    runtimeCompiler: true,
    css: {
        sourceMap: true
    },
    publicPath: '',
    //build for docs folder to enable gh-pages hosting
    outputDir: './docs/',
    assetsDir: 'assets'
}
```

The command to "delete('eslint') is included as a convenience. Linting is very helpful for detecting syntax errors, but sometimes the rules can be strict or seemingly arbitrary and if they are in place can break the build. They are turned off here by the following command in the file above.

```
chainWebpack: config => {
//turn off elint for webpack transpile
config.module.rules.delete('eslint');
},
```

If you remove that command or comment it out, you will turn linting on. Feel free to experiment with this if you are interested.

aliases.config.js

The reference to the `./aliases.config.js` file means we need to add the file to our root as well. This file provides the convenience of references files that are stored in the `src` directory with the `@` symbol. For example following imports are equivalent:

```
import HelloWorld from '@/components/HelloWorld.vue'
import HelloWorld from 'src/components/HelloWorld.vue'
```

Create the `./aliases.config.js` file and paste the following instructions into it.

```
const path = require('path')
function resolveSrc(_path) {
  return path.join(__dirname, _path)
}
const aliases = {
  '@': 'src',
  '@src': 'src'
}
module.exports = {
  webpack: {},
  jest: {}
}
for (const alias in aliases) {
  module.exports.webpack[alias] = resolveSrc(aliases[alias])
  module.exports.jest['^' + alias + '/(.*)$'] =
    '<rootDir>/' + aliases[alias] + '/$1'
}
```

Once we have created these files we should be ready to build the project for deployment.

Building and Deploying the App

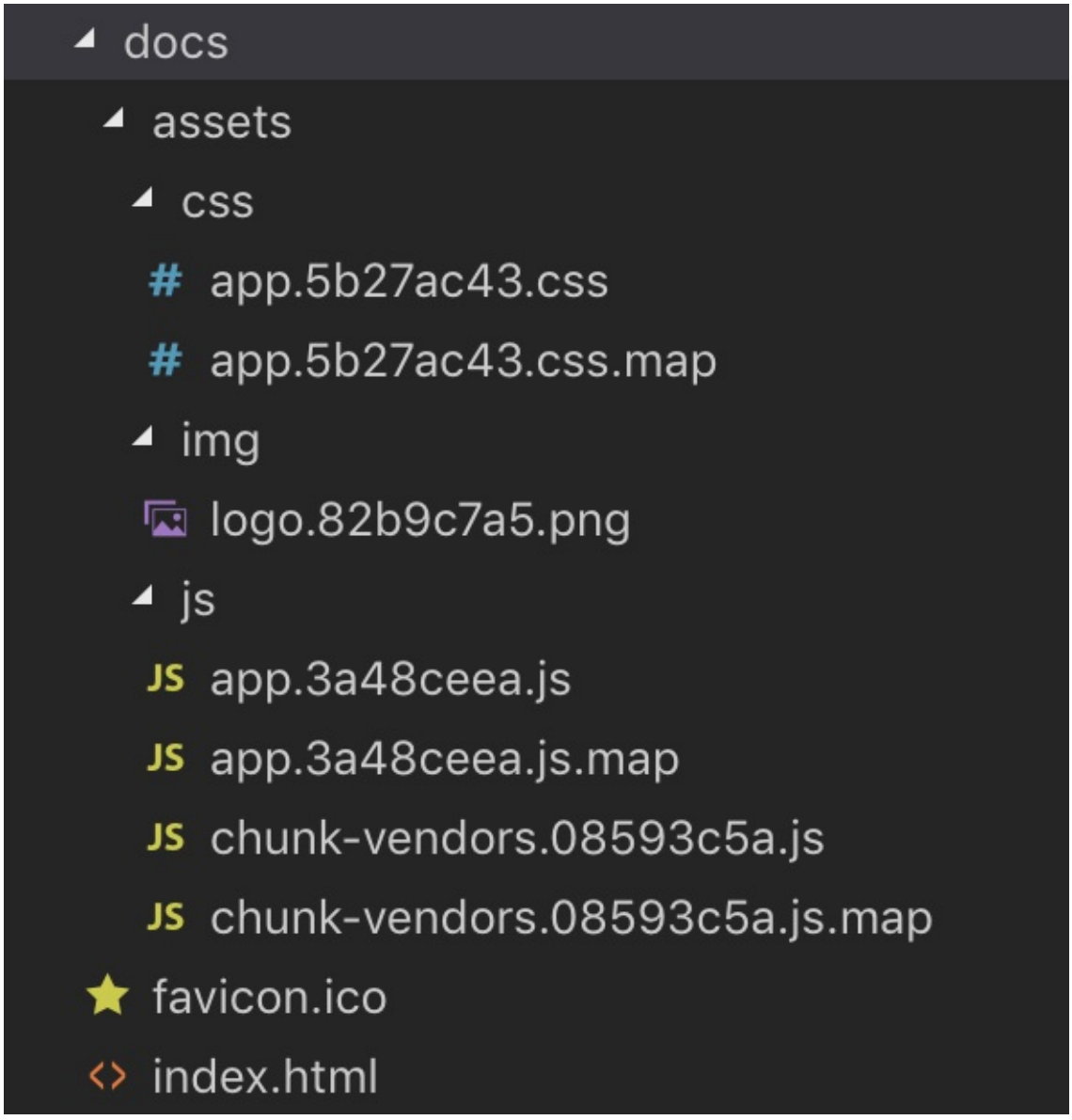
Now that we have our application configured for proper building, we can actually execute the commands to make the build happen, and then to deploy the application.

Build the Code

In order to process our code and build the site, execute the build command:

```
npm run build
```

Since we made the changes to redirect output of the build process to the `docs/` directory, we should see that directory was created. If we look inside it, we should see something like this:



Directory listing of `docs/`

Our files have been properly combined, processed, and built into a static website. We are now ready to deploy the site.

Deploy the Site

Since we are using the Github Pages feature of our repository, all we need to do is commit our code and push it to Github. This might require us to make a new repository on Github and follow those directions to connect everything. Once we push the code with the site built out in the `docs/` directory, Github Pages will automatically pick that up and deploy it to the static website servers. If we look at the Settings page for our project repository we can see the URL where our site is deployed.

When we want to deploy the code we just built, we can run the following commands:

```
git add docs
```

Then:

```
git commit -m "Built new version of site."
```

Finally:

```
git push origin
```

Commit Code Without Deploying the Site

It's often necessary to work on a feature for awhile before we want to deploy the new code. In these cases, we need to be able to commit our code and push it to Github for review/collaboration without deploying the site.

Remember that the only way to update the site code is to run the `npm run build` command. If we do work elsewhere in the app, but we do NOT run the build command, then we can safely commit and push our code to Github without changing the site that is deployed to the public.

Pro Tip: Advertise Your Link

Github Description Link

When we have a repository on Github, we can add a link that will show at the top of the repo homepage. This is very handy for putting the link to the deployed version of our code. Adding a link here also helps our repositories work better as portfolio pieces when we're looking for jobs.

Quiz: Deploying the App

Please enjoy this self-check quiz to help you identify key concepts, points, and techniques discussed in this section.

Visit Quiz Online

The quiz on this page has been removed from your PDF or ebook format. You may take the quiz by visiting this book online.

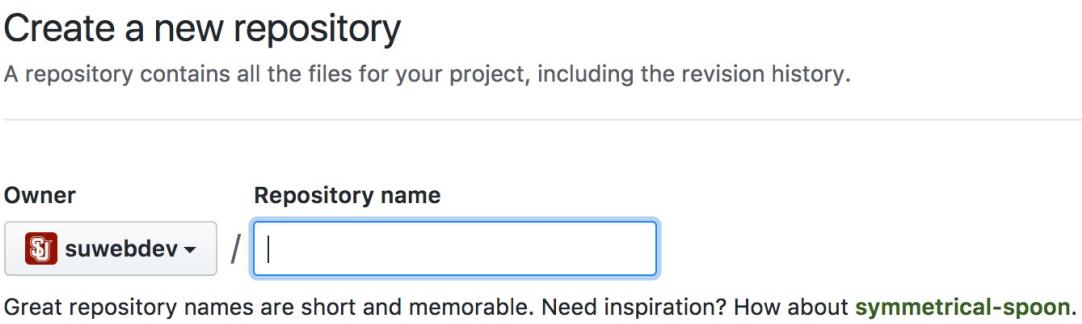
Project: Deploying the App

In order to practice deploying apps, we can update the configuration in the project skeleton we created for the past two sections. We will deploy our site to the Github Pages server, which is a popular, and free, static website server. Since all of our files are compiled and built by Webpack, we can deliver them as static files to our end users.

In order to make all of this happen, we will need to add the configuration files as described earlier in this section, then set up Github to server our pages from the `docs/` directory. Once we set all of that up, then we need only to commit and push our code to deploy our updated build.

Create a Repository

If we have not already created a Github repository, let's do so. On github, click the big green "New" button that shows up on our profile page. This should bring you to the Create New Repository page.



Create New Repository screen

Give the repository a name, and then select private or public. We do not need to add a README.me or any LICENSE.md files because we already have a bunch of files.

Once the repository is created, follow the directions to create the new repository from files on the command line:

New repository instructions

The only thing we should do different is instead of doing the command `git add README.md` we should add everything with `git add -A`. Once we have followed these directions, we should be able to see our repository on Github.com. Verify that all of our files are there, and then move on to the next step of configuring Webpack.

Configuring Webpack

Once we have created our repository on Github, we must configure Webpack to build our files into the `docs/` directory. This requires us to add the `./vue.config.js` and `./aliases.config.js` files. Create the two new files in the root of your project and copy the contents below into them.

vue.config.js

```
const path = require('path');
module.exports = {
    configureWebpack: {
        resolve: {
            //allow for @ or @src alias for src
            alias: require('./aliases.config').webpack
        }
    },
    chainWebpack: config => {
        //turn off elint for webpack transpile
        config.module.rules.delete('eslint');
    },
    runtimeCompiler: true,
    css: {
        sourceMap: true
    },
    publicPath: '',
    //build for docs folder to enable gh-pages hosting
    outputDir: './docs/',
    assetsDir: 'assets'
}
```

aliases.config.js

```
const path = require('path')
function resolveSrc(_path) {
  return path.join(__dirname, _path)
}
const aliases = {
  '@': 'src',
  '@src': 'src'
}
module.exports = {
  webpack: {},
  jest: {}
}
for (const alias in aliases) {
  module.exports.webpack[alias] = resolveSrc(aliases[alias])
  module.exports.jest['^' + alias + '/(.*)$'] =
    '<rootDir>/' + aliases[alias] + '/$1'
}
```

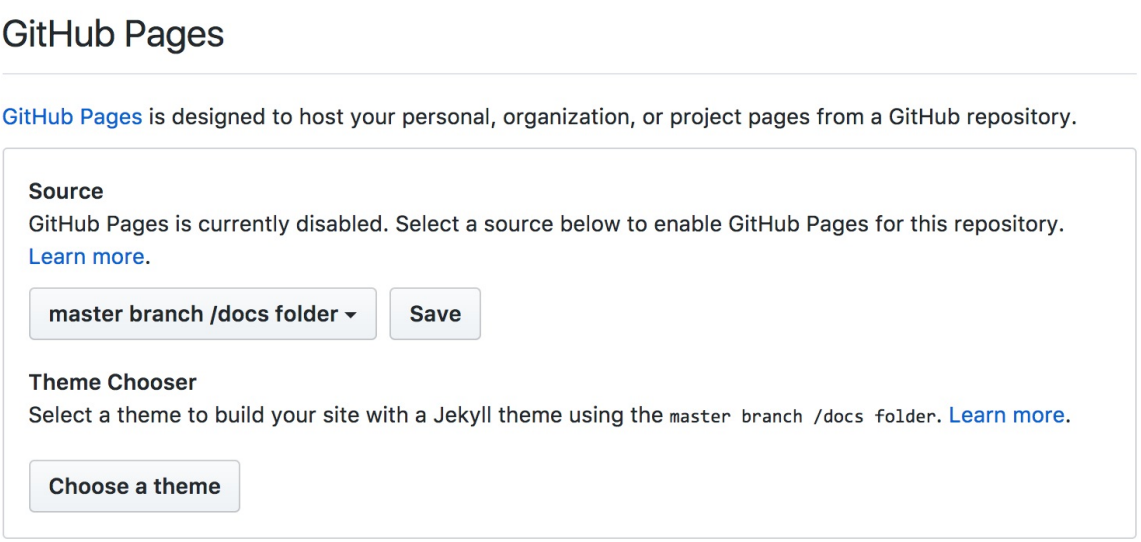
After creating the config files, run the build command:

```
npm run build
```

We should see build output in the console (but not the web browser). Take a moment to confirm that no errors occurred. If everything looks good, then commit and push our changes to Github.

Configure Github Pages

In order to deploy our site to Github Pages, we must alter the settings on our repository. From the repository homepage, click "Settings". Scroll down on the main page of the Settings until you find the Github Pages area:



GitHub Pages configuration

The Source setting should be "master branch /docs folder". This will allow our users to view our project at a URL that matches the pattern `https://username.github.io/project-name/`. When we click "Save" next to the GitHub Pages Source setting, the Settings page will refresh and it will show you the URL where our site has been deployed. Give it a few moments and the bar should turn from blue to green, indicating that the site is ready to be viewed.

Click the link and witness the awesomeness of our first deployed Vue.js application!

Full Repository

If you need to review a working repository with full code, look at this working repository to see all of the edits we made in this project.

Working with Templates

As we learned when we were first exposed to JavaScript, we can use JS to add, remove, and modify elements from the DOM. This is a very powerful capability, but it quickly becomes tedious to write all of those `document.createElement()` statements and properly combine multiple levels of HTML tag hierarchy. In addition to the repetitive strain of creating elements individually, it's possible to hurt the performance of our applications with DOM manipulation.

Changing elements in the DOM is an intense process for our computing devices. If we change elements often, or if we change many elements on the screen, then we can slow down the functioning of our site and the web browser (and possibly our entire computer). The reason for this performance issue is that many DOM manipulations cause the web browser to re-render the entire page. When that happens there is a (potential) risk of serious impacts on your device performance while the rendering is taking place. Because of the performance risk coupled with the tedious nature of manually manipulating the DOM, we have created templating systems that can help with both of these problems.

Using a template allows us to define an HTML structure and use placeholder "template tags" to represent data we want to output. The template can be processed by our chosen application framework and handled in a way that is somewhat more intelligent than how we might otherwise write our DOM manipulations. In the case of Vue.js, the templating system uses a "virtual DOM" that allows the JavaScript system to calculate all the changes that would happen in the DOM and then apply them in one chunk. That process minimizes the number of times the page must be re-rendered, which makes for a more performant experience. The user will perceive the site as being faster, and it is less likely to cause slowdown issues on our user's computer.

Virtual Dom vs. Shadow Dom

There are a lot of things evolving in web technology today, and one area of development is the continued evolution of the DOM. There is a web standard growing around a concept called the "Shadow DOM", and that concept is often confused with the idea of the "Virtual DOM", which is used by application frameworks such as Vue.js and React.

Do not be confused: Virtual DOM and Shadow DOM are two different things.

To fully describe the way that either the Virtual DOM or the Shadow DOM work would take an entire book in itself. They are both powerful concepts that benefit web developers in different ways. The simplified explanation of the difference between Virtual DOM and Shadow DOM is this:

The Virtual DOM is a software design pattern that uses a secondary representation of the DOM in order to collect a batch of DOM changes that can be applied at, essentially, the same time. The goal behind this pattern is to **increase performance** by consolidating DOM changes and reducing the number of times the page must be re-rendered in the browser.

The Shadow DOM is a web standard that defines the way any individual HTML element can implement its own DOM structure encapsulated within itself. This is a complex idea, but the bottom line is that the Shadow DOM is concerned with allowing us to define attributes (such as visual styles) on DOM elements without having those attributes spillover and affect other DOM elements. The Shadow DOM is about **encapsulation of information**.

The bottom line difference is that the Virtual DOM is a tool for increasing performance of our websites, while the Shadow DOM is about encapsulating logic within components of our website software.

In addition to the performance benefits and convenience of using templates, these systems usually give us access to a set of tools that can be used to control the presentation of information to the user. Not only can we display dynamic data from our system, but we can also use conditionals and loops to control that display, easily set up event handlers on HTML elements, and perform other functions that help our websites look and work better.

We will explore templates in-depth in this section of the book and practice working with them in the project.

Core Concept: Template Fundamentals

There is a need in any web-based project to output HTML that can be rendered by the browser. Whatever functionality we write in JavaScript must be exposed to the user through HTML elements. We can do this in native JavaScript using methods like `document.createElement()` to manually create each HTML element and then append it to the proper parent so it appears in the page. But this approach is tedious and prone to error.

In order to solve the problem of "how do we generate HTML for the user?" we have come up with the concept of "template engines." A template engine is a software tool that allows us to make patterns of HTML. Those patterns can define whatever HTML structure we need in our website or application, and as we define the patterns we note areas where data should be output. By processing the template against a set of data, the template engine can fill in all of those placeholders. The result is the final HTML with all of the dynamic data our application has generated.

The concept of template engines, templating HTML (or other markup), and processing templates against a set of data is not isolated to the world of JavaScript application frameworks. Template engines are used in virtually every web-based application framework, whether that framework uses JS, Python, Ruby, PHP, or other languages. Template engines usually allow for basic features that make it possible to output common data structures (Objects and Arrays, primarily) with tools like loops, conditionals, and the ability to add some form of customized functionality (called "directives" in some JavaScript framework template engines).

Like every other component we use to build websites, template engines are particular and specific. Different developers, different developer communities, and different projects tend to prefer certain approaches or philosophies of design. Although there are many variations in the details of how different templating engines work, this section tries to focus on some of the generally consistent aspects of these engines. All of the examples in this section are written with the Vue.js template syntax in mind.

Template Syntax and Vocabulary

Each templating engine uses a slightly different syntax. This syntax is necessary because we must be able to determine the difference between the code meant to define the behavior of the template rendering, and the HTML code meant to be output as part of the final HTML result.

To get an idea of how template syntax might vary, here are examples from Vue.js and the Django framework (which is a server-side Python framework). Each of these examples shows a template snippet that will result in a list of search results showing the result name as a link. We assume there is an Array called `items` available in the template context. The `items` Array contains a list of Objects that each have `id` and `name` properties. Each of these templates demonstrates basic data output inside a `for` loop.

Django Template

```
<ul id="search-results">
    {% for item in items %}
        <li class="result">
            <a>{{ item.name }} - ID: {{ item.id }}</a>
        <li>
    {% endfor %}
</ul>
```

Vue.js Template

```
<ul id="search-results">
    <li class="result" v-for="item in items">
        <a>{{ item.name }} - ID: {{ item.id }}</a>
```

```
    </li>
</ul>
```

In these two examples, we can see that although these frameworks are quite different their templating languages use the same fundamental concepts and very similar approach. The syntax varies a little bit: Django templates use "template tags" instead of directives to invoke features like a loop. The Django template will loop all content between the `{% for %}` and `{% endfor %}` template tags.

The Vue.js template uses directives instead of template tags. The `v-for` attribute on the `<li>` element is the directive that establishes the `for` loop. It will create a copy of the HTML element the directive is attached to, and anything inside that element will be part of the looped HTML generation.

The output of both of these template snippets would be something like this:

```
<ul id="search-results>
    <li class="result">
        <a>Betty Holberton - ID 12</a>
    </li>
    <li class="result">
        <a>Anita Borg - ID 9</a>
    </li>
    <li class="result">
        <a>Frances Spence - ID 17</a>
    </li>
    <li class="result">
        <a>Grace Hopper - ID 51</a>
    </li>
    <li class="result">
        <a>Leah Culver - ID 42</a>
    </li>
</ul>
```

The specific content of the list would depend on the data in the system, but assuming we were searching for famous female software developers, this is a possible set of returned results.

Each time we wish to output the value of a variable in the template context, we use the double curly braces syntax:

```
<a>{{ item.name }} - ID: {{ item.id }}</a>
```

This syntax indicates that the value of variable named between the double curly braces should be output into the template. This double curly braces syntax is often called "mustache" syntax because the curly braces look like mustaches. Some templating languages use other syntax to output individual variables, but this syntax is very common in templating engines today.

When we process the data in the template context against the template, we call that "template rendering," or just "rendering" for short. We render a template that is injected into the browser (or, in the case of Django templates, which is then served to the user as an entire HTML page).

It is important to learn the syntax of the specific templating engine we are using, but most engines allow for the same basic kinds of features.

Data in Templates

The data used to render a template is called the "template context" or "context" for short. This is basically the scope of the template: All the things in the software application that the template knows about. Usually, our templates do not need to know about most of the parts of our systems. We only want to render a comparatively small set of data that is

used within a specific page or component. By keeping our template context small and free of superfluous data we speed up the processing of the template.

Data can be referred to in templates in any location to provide dynamic HTML generation. These can be used to add classes to elements, construct URLs, loop through data structures, or create virtually any HTML structure we need. Standard dot-notation referencing can be used within most templating engines to refer to properties of data in the context.

Here is an example where we have a few different values in the context. (This example, and the rest of the examples on this page, use Vue.js template engine syntax.) This example imagines a page listing weather and events for Seattle, WA.

Data in the Template Context

```
{
    city: "Seattle",
    state: "WA"
    forecast: {
        temps: {
            high: 83,
            low: 58
        },
        summary: "Mostly sunny."
    },
    events: [
        {
            name: "Film Festival",
            start: "8:00pm",
            end: "10:00pm",
            location: "12th Ave Arts"
        },
        {
            name: "Art Opening",
            start: "7:00pm",
            end: "10:00pm",
            location: "Seattle Art Museum"
        },
        {
            name: "Community Town Hall",
            start: "6:00pm",
            end: "7:30pm",
            location: "City Hall"
        }
    ]
}
```

Template Code

```
<h1>Events for {{ city }}, {{ state }}</h1>
<aside class="weather">
    <p class="summary">{{ forecast.summary }}</p>
    <ul class="temps">
        <li class="high">{{ forecast.temps.high }}</li>
        <li class="low">{{ forecast.temps.low }}</li>
    </ul>
</aside>
<ul class="events">
    <li v-for="event in events">
        <h2>{{ event.name }}</h2>
        <p class="times"><span class="start">{{ event.start }}</span><span class="end">{{ event.end }}</span></p
>
        <p class="location">{{ event.location }}</p>
    </li>
</ul>
```

The results of this code would be the output of all of the data in the data context described above. We can see that the data object is revealed to the template, and the root properties of the object (`city` , `state` , `forecast` , and `events`) are accessible using their names. To access the properties of those objects, we use the same dot-notation references that we use in our JavaScript: `forecast.temps.high` or `forecast.summary` .

Because `events` is an Array, we can loop through it. We should discuss loops in more detail.

Looping

Looping is a core feature in pretty much any templating language. Part of the huge benefit of templates is that we can loop through our data and output many copies of the same HTML structure. This allows us to save time not writing all of that code by hand, and it also allows us to make sure we are consistent with how each item in an Array is presented to the user.

Creating loops in templates is usually syntactically sparse: We usually don't have to write a lot to make a loop happen. From the example above, assuming we have an Array called `events` in the template context, we could write this:

```
<ul class="events">
    <li v-for="event in events">
        <h2>{{ event.name }}</h2>
        <p class="times"><span class="start">{{ event.start }}</span><span class="end">{{ event.end }}</span></p
>
        <p class="location">{{ event.location }}</p>
    </li>
</ul>
```

In this example we see a `<ul>` element with a class attribute. The "events" class does not affect anything in the template and would be used by the CSS to style the element. The for loop is attached to the `<li>` element because that is the element that we actually want to copy for each item in the `events` Array. The loop is invoked with the `v-for` directive, which is written like an attribute on the HTML element that we want to duplicate for each item in the loop.

Loops in templates generally repeat a specific set of HTML for each item in the Array they are looping through. The elements that are children of the HTML element with the loop attached will also be duplicated. So in this example, the result would be an unordered list with three list items:

```
<ul class="events">
    <li>
        <h2>Film Festival</h2>
        <p class="times"><span class="start">8:00pm</span><span class="end">10:00pm</span></p>
        <p class="location">12th Ave Arts</p>
    </li>
    <li>
        <h2>Art Opening</h2>
        <p class="times"><span class="start">7:00pm</span><span class="end">10:00pm</span></p>
        <p class="location">Seattle Art Museum</p>
    </li>
    <li>
        <h2>Community Town Hall</h2>
        <p class="times"><span class="start">6:00pm</span><span class="end">7:30pm</span></p>
        <p class="location">City Hall</p>
    </li>
</ul>
```

We can see that all of the elements that were within the `<li>` elements are also duplicated. This allows us to do much more work fine-tuning the presentation of an individual instance of our repetitive HTML structures, so we are able to write better code that can be more useful to our users.

Within the loop it's important to realize how we are accessing the data in each item. As the template engine iterates through the loop, each item in the Array is handled. On each iteration one item from the Array is assigned the name we have specified in our code.

In this example, we have used a common convention. We have an Array named with a plural noun (`events`), so we set up our loop to use the singular form of that noun to reference each item as it comes through the loop:

```
<li v-for="event in events">
```

This name is determined by us. We could write `v-for="item in items"` or `v-for="posts in blogPosts"` or any other combination of terms that makes sense in the context of our application. What is important is that we consistently use whatever name we assign to the item as it moves through the loop code. If we call it `event` then we must reference `event.name` and `event.location` . If we wrote `<li v-for="oneEvent in Events">` then we would need to reference `oneEvent.name` and `oneEvent.location` .

Conditionals

Conditionals are used within templates to "turn on and off" elements. We often use them to modulate between states, or to show/hide things like administrative or editing controls. Consider, for example, the common situation where we wish to show a user a login link when they are not logged in and an account link when they are logged in. It is not unusual to have a template snippet like this in a web application:

```
<ul id="identity">
    <li v-if="username">
        <a href="/account/">{{ username }}</a>
    </li>
    <li v=if="!username">
        <a href="/login/">Login</a> or <a href="/signup/">Sign Up</a>
    </li>
</ul>
```

In this example, we assume that when a user logs in to our site the variable `username` is made available to the template context. This is a common approach for many websites, and it allows us to use conditional directives to determine whether we should show a personalized link to the account management page, or if we should show links to login or make an account.

We can also use conditional elements to accomplish show/hide features, to help manage tabbed information displays, and all sorts of other interface customizations. The speed of template rendering in JavaScript application frameworks tends to be very fast, so it is usually reliable to use these conditionals in combination with visual effects and other processing to create highly interactive and responsive interfaces.

And More

Templating engines are each unique, and most of them give developers a way to extend their functionality. In Vue.js, developers are given easy ways to implement template filters, which allow us to create reusable data formatting or presentation functions that can easily be applied within the template. In other templating engines the methods might be different but there is usually a way to add more to the abilities of the engine.

It's worthwhile to dig into the features of our templating engine to be sure we are using all of the power in our tools. These simple features provide a huge range of capability and once we master these concepts we can accomplish a lot through our use of templates.

Using Data in Templates

One of the primary goals of using templates is to interpolate data into the template. It's critical to be able to insert data into the context of the template and to update that data when the system changes. For the most part, the process of using data in templates is just a matter of referencing the value using the mustache syntax (double curly braces). But there are some finer points that are worth knowing about.

Basic Data Output

As we saw in the previous section, we use the mustache syntax to output data in our template HTML:

```
<a>{{ item.name }} - ID: {{ item.id }}</a>
```

The data in the template context comes from the `data()` function defined in the Vue component itself:

```
<script>
export default {
  name: 'hello',
  data () {
    return {
      msg: 'Hello, World!',
      item: {
        id: 12,
        name: "Foo"
      }
    }
  }
}
</script>
```

The object returned by the `data()` function is revealed to the template context. The "root" properties of the object are accessible as named variables. In the example above, `msg` and `item` are the "root" properties of the data object, so they can be referenced as `{{ msg }}` and `{{ item }}` in the templates. Of course, the `item` property references another object, and those properties can be accessed as `{{ item.id }}` and `{{ item.name }}` (as seen in the other example above).

Interpolating variables from the template context like this sets up a two-way binding: The data is rendered in the HTML for the user to see, and it will update if the system updates the value. So if the user performs some action our code can update the values of any data being rendered on the page and the user will see that information be updated. This process of connecting our application logic and template output is called "binding."

We can use the mustache syntax to output basic JavaScript expressions, too. The following are all valid data output:

```
<p>The next page is: {{ page + 1 }}</p>
<p>A ternary statement: {{ ok ? 'YES' : 'NO' }}</p>
<p>We can call data type methods: {{ message.split('').reverse().join('') }}</p>
```

All of the expressions above are allowed, which offers an immense range of output options when interpolating data into a template. Although most data processing should be done inside the Vue Component, these expressions can be handy for formatting and optimizing presentation of data for the user.

These methods work fine for outputting text data between HTML tags. But there are some additional considerations when the situation gets a little more complex.

HTML Data

We should generally avoid the output of data values that contain HTML markup. It's usually more advantageous to have, for example, the text of the headline without the heading tags rather than the headline with tags baked into the system's data. Sometimes we want to output the headline of an article as the main title of a page, and other times we want to output the headline as the title of a list item in the sidebar. In different situations, we need different HTML structures, so having HTML saved as part of the text data is problematic. This is why we use templates.

However, sometimes we have data objects such as messages, blog posts, and other components that might involve saving HTML formatting in the system data. In these cases, we need to use a slightly different technique to tell the Vue.js template engine that we wish to output HTML that should be rendered as part of the template. To do this, we will use the `v-html` directive:

data object

```
<script>
export default {
  name: 'hello',
  data () {
    return {
      message: `<p>Nullam nulla eros, ultricies sit amet, nonummy id, imperdiet feugiat, pede. In dui magna, po
suere eget, vestibulum et, tempor auctor, justo.</p>
               <p>Fusce commodo aliquam arcu. Fusce fermentum odio nec arcu.</p>`
    }
  }
}
</script>
```

template code

```
<div v-html="message"></div>
```

rendered template

```
<div>
  <p>Nullam nulla eros, ultricies sit amet, nonummy id, imperdiet feugiat, pede. In dui magna, posuere eget, ve
stibulum et, tempor auctor, justo.</p>
  <p>Fusce commodo aliquam arcu. Fusce fermentum odio nec arcu.</p>
</div>
```

As we can see in this example, the `message` data property contains a string with HTML formatting. In order to properly render this HTML in the template, we use the `v-html` attribute. Anytime we use a directive we use the equals sign and then put the argument in quotes, so they look like normal HTML attributes. In this case, the variable `message` will be used for the content of this HTML element.

Use Caution with HTML Content

It is possible to introduce security vulnerabilities in our projects by outputting user-provided HTML with the `v-html` directive. Malicious users could insert scripts into comments or other user-generated text and those could attack our other users. This is why most commenting or review systems either disallow all HTML formatting, only allow a small set of safe tags that are validated upon submission of the content, or use an alternative markup formatting to do basic text enhancement.

> Using `v-html` should be reserved for content we create and verify is safe before use. And, where possible, we should avoid HTML formatting in our internal content

Data in HTML Attributes

It's often the case that we want to use data from the template context to augment HTML attributes. For example, we may want to construct a URL that includes a user's username. Or we may need to build a details link that includes the item's ID:

```
<a href="item/12">View Details</a>
```

Since we cannot use the mustache syntax in HTML attributes, we must use the `v-bind` directive to insert the data from the template context. This creates a data binding between the information inserted in the attribute and the logic of our application, so if the values change the HTML will be updated accordingly.

Assuming we have a data object similar to the one used in the example above, we could write our link like this:

```
<a v-bind:href="'item/' + item.id">View Details</a>
```

We can see that the `v-bind` attribute uses the Vue.js method of adding parameters to a directive. In this case, we need to tell the `v-bind` directive *which* attribute we want to bind. We are binding the `href` attribute. As with all directives, we provide arguments between quotes after an equals sign. In this case, we are using a JavaScript expression to concatenate the String `'item/'` with the value of `item.id`.

Another common example is when we have image URLs provided in our data, and we want to put the images into an HTML structure for display:

data

```
<script>
export default {
  name: 'hello',
  data () {
    return {
      image: {
        src: "images/mypicture.jpg",
        title: "Placeholder Image"
      }
    }
  }
}
</script>
```

template

```
<div class="image">
  <img v-bind:src="image.src" v-bind:alt="image.title">
</div>
```

rendered template

```
<div class="image">
  <img src="images/mypicture.jpg" alt="Placeholder Image">
</div>
```

In this example we can see how we can use the `v-bind` directive to populate the image `src` and `alt` attributes. These substitutions don't require any JavaScript concatenation of the text and data. This kind of manual binding is common as we work with images and other elements in our templates.

Shorthand for `v-bind`

Although using the `v-bind` directive is fairly straightforward, we might do it often in our templates, and we might want to stop typing `v-bind` so often. Vue.js provides us with a shorthand we can use to set a manual binding on an attribute. The following code is equivalent:

```
<img v-bind:src="image.src">

<img :src="image.src">
```

We can abbreviate `v-bind:attributeName` to simply `:attributeName` in order to save a few keystrokes. This shorthand does not change anything about how the directive functions, and there is no positive or negative effect beyond removing the need to type `v-bind` .

Conditionals in Templates

Conditionals are a core part of any templating engine. These allow us to make decisions in the context of the template rendering, which we can use to alter the content we display. We can use conditionals in templates the same way we use them in programming languages: Vue.js templates support the standard `if`, `else if`, and `else` conditions. In this section we will take a look at how to use these in a template.

Conditional Syntax

Writing conditionals in Vue.js templates involves using the `v-if` directive along with directives that correspond to the typical conditional clauses: `v-else` and `v-else-if`. These operate the way we would expect, and they can be nested and combined in just about any way. Here is an example:

data

```
<script>
export default {
  name: 'hello',
  data () {
    return {
      item: {
        type: "ticket",
        title: "Show Ticket"
    }
  }
}
</script>
```

template

```
<div v-if="item.type === 'ticket'">
  <p>Ticket added to cart.</p>
</div>
<div v-else-if="item.type === 'album'">
  <p>Album added to cart.</p>
</div>
<div v-else>
  <p>Item was neither a ticket nor an album.</p>
</div>
```

rendered template

```
<div>
  <p>Ticket added to cart.</p>
</div>
```

As we can see from this example, we write the conditional statement between the quotes of the directive assignment. The conditional must begin with a `v-if` directive, which provides the initial conditional statement. If this evaluates to true, as it does in our example, then it will render the element attached to the `v-if` directive (and any child elements contained within). However, if the `v-if` conditional statement were not true, then the template processing would continue to the `v-else-if` statement.

The `v-else-if` statement would be evaluated just like the `v-if` statement was evaluated. If it is true, then that HTML element is rendered along with whatever content it contains. If it is false, then the processing would continue to the `v-else` statement. The `v-else` element is rendered whenever the preceding conditional statements are not true. The `v-else` directive does not require a conditional statement because it will only be rendered when the previous conditional statements evaluate to false.

These conditional structures can be attached to any HTML elements we need to put in our templates. We can nest them if we need to, and by combining them in clever ways we can create all kinds of interesting functionality. We often use conditional statements to modulate the content of our HTML, but if all we want to do is make an element show or hide, there is a shortcut way to do that.

Show/Hide Content Shortcuts

Vue.js gives us the `v-show` directive, which can be applied to any element. This directive accepts a value that will evaluate to `true` or `false`. If the condition is `true`, the element is made visible. If it is `false`, then the element is hidden. Here is an example:

data

```
<script>
export default {
  name: 'hello',
  data () {
    return {
      showMe: false
  }
}
</script>
```

template

```
<div v-show="showMe">
  This content should be shown when showMe is true.
</div>
```

In this example, the `showMe` value is set to `false`, so the entire `<div>` will be hidden from view. It will still be rendered in the DOM, and it will exist in the HTML in the user's browser, but it is hidden with CSS. This technique is suitable for situations when you want to be able to toggle show/hide on the element very often. It is also suitable when the content of the element does not make as much of an impact on template rendering.

In situations where we need quick show/hide functionality, the `v-show` directive can be a useful tool.

Deciding Between `v-if` and `v-show`

Without going into the deeper technical details of the difference between `v-if` and `v-show`, it is possible to use a rule of thumb to determine when you should use one directive over the other. The `v-if` directive is a more substantial feature. It requires more effort to render, but it also completely adds and removes the content in the template. If we are using event listeners on content items within the conditional, then the `v-if` directive allows those listeners to be completely removed from the template when they are not needed. This can improve the overall performance of our applications.

When we use `v-show`, the process is not as involved: The `v-show` directive never adds or removes the content, it only makes the content visible or not. That means that `v-show` is a little more intense when we first load the page, but then it can toggle the show/hide possibly more quickly than the `v-if` directive.

If we are using the show/hide to reveal simple content that does not include many event listeners, then `v-show` will probably be a better option to create a fast and responsive application. However, if we are trying to add/remove content with event listeners, or more complex template rendering impacts, then we may be better to use the `v-if` directive in order to avoid those computing costs.

Looping in Templates

It is very common when building applications to have the need to loop through sets of objects. In JavaScript frameworks, we often work with data that contains Arrays, which are iterable lists that we can process in a loop, or we want to loop through the properties of an Object in order to enumerate the data for the user. Looping in templates allows us to create consistent repetition of HTML structures in order to output better-formatted information to the user.

Basic Syntax

The loop that we have in Vue.js to work with is a `for` loop. It is invoked with the `v-for` directive. The `v-for` directive is applied much like the other directives we've looked at in this section. The directive can be applied like so:

data

```
<script>
export default {
  name: 'hello',
  data () {
    return {
      messages: [
        "Hello, world!",
        "42 is the meaning of life.",
        "Question authority."
      ]
  }
}
</script>
```

template

```
<ul>
  <li v-for="message in messages">{{ message }}</li>
</ul>
```

rendered template

```
<ul>
  <li>Hello, world!</li>
  <li>42 is the meaning of life.</li>
  <li>Question authority.</li>
</ul>
```

In this example, we can see that an Array called `messages` is being looped through. The loop is attached to the `<li>` tag, so that tag is duplicated. As with other directives, the loop would actually duplicate the tag and any content that is inside the tag, including other HTML elements. It is common practice to use the "singular in plural" naming pattern for loops, but we could use whatever names we wish.

Sometimes it's beneficial to know what iteration of the loop we are on. We could modify this example slightly to demonstrate that:

data

```
<script>
export default {
  name: 'hello',
```

```
  data () {
    return {
      messages: [
        "Hello, world!",
        "42 is the meaning of life.",
        "Question authority."
      ]
  }
}
</script>
```

template

```
<ul>
  <li v-for="(message, index) in messages">{{ index }}: {{ message }}</li>
</ul>
```

rendered template

```
<ul>
  <li>1: Hello, world!</li>
  <li>2: 42 is the meaning of life.</li>
  <li>3: Question authority.</li>
</ul>
```

By adding `index` as a value in the declaration of the loop, we can use that within the loop. We could write conditionals to test, in case we wanted to do some conditional HTML formatting. Or we could use the information to make the data more clear for the user.

It's also possible to loop through Objects, too:

data

```
<script>
export default {
  name: 'hello',
  data () {
    return {
      profile: {
        username: "jdoe",
        firstname: "Jane",
        lastname: "Doe",
        email: "jdoe@example.com"
      }
  }
}
</script>
```

template

```
<ul>
  <li v-for="(value, key) in profile">{{ key }}: {{ value }}</li>
</ul>
```

rendered template

```
<ul>
  <li>username: jdoe</li>
  <li>firstname: Jane</li>
  <li>lastname: Doe</li>
```

```
  <li>email: jdoe@example.com</li>
</ul>
```

In this example, rather than using the `item` and the `index` naming, we are using the `value` and `key` naming. This allows us to go through all of the properties in the Object, which are referenced using named keys, and output those along with the actual data stored in each property. This is often a better way to output the data in an Object. (Note: It is possible to write `v-for="value in profile"`, too, but that would only give us the property data to work with and not the property keys.)

These basic tools open the door to a lot of possibility. We can format and present data efficiently, consistently, and we can spend a great deal of attention on the details of forming our HTML structures.

More Complex Looping

Of course, loops can be nested. This is often the case when we are presenting information from some sort of data API. A common example might be a system where we are showing search results with the tags that have been applied to them:

data

```
<script>
export default {
  name: 'hello',
  data () {
    return {
      results: [
        { title: "Computers are Fun",
          tags: ["computers", "programming"]
        },
        { title: "Bugs are Cool",
          tags: ["bugs", "nature", "activities"]
        },
        { title: "The Sun is Bright",
          tags: ["sun", "astronomy", "climate"]
        }
      ]
  }
}
</script>
```

template

```
<ul>
  <li v-for="result in results">
    {{ result.title }}<br>
    Tags: <span v-for="tag in result.tags">{{ tag }}</span>
  </li>
</ul>
```

rendered template

```
<ul>
  <li>
    Computers are Fun<br>
    Tags: <span>computers</span><span>programming</span>
  </li>
  <li>
    Bugs are Cool<br>
    Tags: <span>bugs</span><span>nature</span><span>activities</span>
```

```
  </li>
  <li>
    The Sun is Bright<br>
    Tags: <span>sun</span><span>astronomy</span><span>climate</span>
  </li>
</ul>
```

In this example we can see that the Array of results contains Objects. Each of those Objects has a property called `title` that is a String, and a property called `tags` that is an Array. In order to output all the results and all the tags for each result, we must use a nested loop.

We have attached the first loop to the `<li>` element so that it contains all the information about each result. In order to display the `result.tags` data we have attached another loop to the `<span>` elements because those are inline elements that will be easier to style into our desired presentation.

This is just one example of the power of nested loops. Loops are very common in any template usage, so practicing and becoming capable with them is a crucial skill for developers to build.

Using key in Looping

If data supplied in a `v-for` loop changes state, vue.js will use an "in-place patch" strategy to update it in the view. This means that the order should match the order of the items in the view of the elements in the data. This works well if the view output doesn't rely on child components but there is something you can do to ensure your view will match your data regardless of where it is in the view or how it got placed there. You can supply a `key` attribute along side the `v-for` directive.

You may even see a lint error if you don't supply a `key` with your `v-for` directive. If your data has a unique key per item you can use that, but you can also use the index as a key.

For example in the template code provided above I can add an index variable to the list of variables returned by the for loop and then use it as the key.

```
<ul>
  <li v-for="(result,index) in results" :key="index">
    {{ result.title }}<br>
    Tags: <span v-for="tag in result.tags">{{ tag }}</span>
  </li>
</ul>
```

Computed Values

So far we have supplied essentially static data to our components and templates. The data defined in the template context can be changed through user interaction or other actions in our application. But we do not have a way to provide proper, reactive data that is based on a computation.

There are often situations where we want to compute a value based on the other data in the system: Calculating subtotal, tax, and total, providing additional annotation to information based on data values, or combining multiple data points into a more descriptive label, to name a few possible cases. The computed data in each of these examples would be valuable to users, so it's useful to calculate the information on the fly.

It's also possible that we need to reveal some other information that should be calculated at the time the user accesses the application, such as the date/time a function was initiated, the viewport size, or the scroll position of the window. In these cases, it is worthwhile to compute the data and use the computed value to display the information to the user.

Using Computed Values

To create a computed value, we will add a `computed` property to our component object. Here is an example imagining a weather component.

```
<script>
export default {
  name: 'weather',
  data () {
    return {
      forecast: {
        high: "65",
        low: "48",
        rainChance: "67"
    }
  },
  computed: {
    umbrellaTip: function(){
      if (this.forecast.rainChance >= 70) {
        return "Bring an umbrella for sure!";
      } else if ((this.forecast.rainChance > 40) && (this.forecast.rainChance < 70)){
        return "Probably a good idea to bring an umbrella, just in case.";
      } else {
        return "No umbrella needed today.";
      }
    }
  }
}
</script>
```

In this example, the data includes a `forecast` object, which has some weather information in it. The information includes a property called `rainChance`, which is the percentage chance of rain (67% in this example).

We can compute the `umbrellaTip` using the `this.forecast.rainChance` value. We provide three levels of `umbrellaTip`, and we can output this value to the template using the normal data reference syntax:

```
<p>{{ umbrellaTip }}</p>
```

This line in the template would output the following HTML:

```
<p>Probably a good idea to bring an umbrella, just in case.</p>
```

If the value of `forecast.rainChance` were to change, the value of `umbrellaTip` would automatically be updated because the `forecast` object is part of the data used in this template, and it is reactive. We could define multiple computed properties, and each would involve defining a function to calculate the computed value.

Here is another example where we use computed values to calculate the tax and total for a customer:

logic

```
<script>
export default {
  name: 'checkout',
  data () {
    return {
      subtotal: 45,
      taxRate: 0.04
    },
  computed: {
    tax: function(){
      return (this.subtotal * this.taxRate).toFixed(2);
    },
    total: function(){
      return this.subtotal + this.tax;
    },
    timestamp: function(){
      return Date.now();
    }
  }
}
</script>
```

template

```
<ul>
  <li>Time: {{ timestamp }}</li>
  <li>Subtotal: ${{ subtotal }}</li>
  <li>Tax: ${{ tax }}</li>
  <li>Total: ${{ total }}</li>
</ul>
```

rendered template

```
<ul>
  <li>Time: 1506979771129</li>
  <li>Subtotal: $45</li>
  <li>Tax: $1.80</li>
  <li>Total: $46.80</li>
</ul>
```

In this example, we have several computed values. The first is the `tax` value, which calculates the amount of tax using `this.subtotal` and `this.taxRate`. The second computed value is `total`, which adds the `this.tax` amount to the `this.subtotal`. Finally, we have the `timestamp`, which is based on the built-in JavaScript function `Date.now()`.

Since `tax` and `total` are based on reactive values, when the `subtotal` and/or `taxRate` values change the computed `tax` and `total` values would also be updated. However, since the `timestamp` value is based on the non-reactive `Date.now()` value, the value of `timestamp` would not be updated when data is changed.

Computed Strategies

Computed values are powerful tools we can use to help our templates be more easily readable and to give us exactly the output we want. Here are some common use cases for when computed values might be a good idea:

- Annotating data with additional information (such as an "umbrella tip" or other advisory)
- Formatting data to match specific HTML or textual patterns (such as adding a "°F" to the temperature)
- Combining data points into more useful labels (such as showing names in a "Last Name, First Name" display)

There are probably lots of other use cases we will discover as we work through the challenges of building applications. We will continue to learn more ways to affect the data we display to the user in upcoming sections, too. But for now, computed values gives us a way to do a lot of interesting data manipulation.

Quiz: Working with Templates

Please enjoy this self-check quiz to help you identify key concepts, points, and techniques discussed in this section.

Visit Quiz Online

The quiz on this page has been removed from your PDF or ebook format. You may take the quiz by visiting this book online.

Project: Templating Data

For this project, we will practice with some of the techniques used to output data in the template. We will use the Templates and Data Starter Repository to provide the data and basic application structure for this project.

As usual, the project repository defines a set of requirements that we must fulfill in order to complete this project. It's recommended to read through all of the directions before we begin working, and it might be preferable to work on the project before continuing to read through this section. If you become stuck, a full walkthrough of the project continues below.

As with all of the projects, we should begin by forking the starter repo to our own GitHub account, and then cloning it to our workspace.

Review the Directions and Data

The `README.md` file for the project outlines the requirements for completion and provides some ideas for how to push the project further if we crave additional challenge. Before we get too deep in working on this challenge, it's worthwhile to review the JSON object in the `src/apiresults.js` file. This JSON object is a captured result from The Movie Database API. It contains the top 20 movies from the 20th Century, sorted in order of popularity. We can see from the results that there are about 147,445 movies from the 20th Century in the TMDb system, but we are only seeing the top 20 most popular.

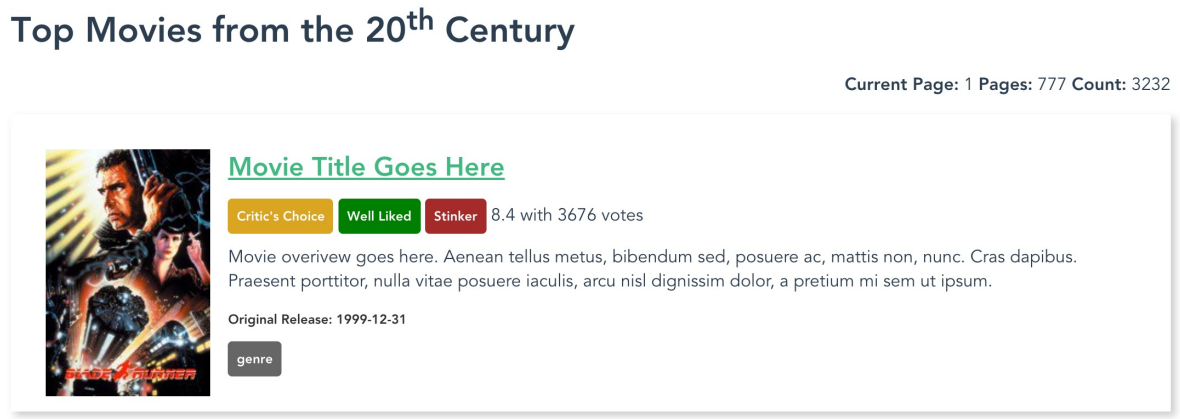
In the data we will notice some "metadata" about the search itself: How many total results, what page we're looking at, and how many total pages there are. We can also see an array called `results` that contains objects representing each of the 20 movies in the list. These objects all have the same data points:


```
{
  "vote_count":3676,
  "id":78,
  "video":false,
  "vote_average":7.9,
  "title":"Blade Runner",
  "popularity":105.797529,
  "poster_path":"/p64TtbZGCElxQHpAMWmDHkWJlH2.jpg",
  "original_language":"en",
  "original_title":"Blade Runner",
  "backdrop_path":"/k36huckDH0v3LP1zo7maFt3mJC0.jpg",
  "adult":false,
  "overview":"In the smog-choked dystopian Los Angeles of 2019, blade runner Rick Deckard is called out of reti
rement to terminate a quartet of replicants who have escaped to Earth seeking their creator for a way to extend
 their short life spans.",
  "release_date":"1982-06-25",
  "genres":[
    "Science Fiction",
    "Drama",
    "Thriller"
  ]
}
```


When we loop through each of the results in our template, each of these data objects will become available to our template. We can rely on each object having these same data points, and we can write our template accordingly. We will need to refer back to these property names so we can be sure to spell and capitalize everything properly.

Working the Project

First, be sure we have run `npm install` in the project repository to install the application dependencies. When that finishes, we can run `npm run dev` to run the development server. We should see the beginning template provided in the repository:



Starter template

To begin working the project, we should open up the `src/components/Results.vue` file. This file contains the template that we need to alter in order to display our work. When we first open the file, we can see that the template is full of static information, and it only shows one movie result. Our goal is to enhance this template to make it dynamic so it can show the full set of data we have in the `src/apiresults.js` file. This data is revealed to the template context. We will access this data in our template in order to fill in the proper information.

Search Metadata

Most search interfaces provide us with "metadata" about the search itself: How many items were found, how many pages there are, and what page we are currently viewing. This information is present at the top of the JSON object returned by TMDb. We must output the data into the proper area of the template:

```
<p class="search-meta">
  <span class="current-page"><b>Current Page:</b> 1</span>
  <span class="total-pages"><b>Pages:</b> 777</span>
  <span class="total-results"><b>Count:</b> 3232</span>
</p>
```

The relevant values in the root of the JSON object are:

```
"page":1,
"total_results":147445,
"total_pages":7373,
```

So we can add those values to our template by using basic data interpolation and their property names:

```
<p class="search-meta">
  <span class="current-page"><b>Current Page:</b> {{ page }}</span>
  <span class="total-pages"><b>Pages:</b> {{ total_pages }}</span>
  <span class="total-results"><b>Count:</b> {{ total_results }}</span>
</p>
```

This is the basic way we interpolate data into our templates. Since these properties all exist at the root (main level) of the JSON object, we can just use their names to reference them.

Movie Items

The meat of the page is really the movie items themselves. The HTML structure for the movie items has been provided for you: These will be formed by list items (`<li>` elements) within an unordered list. This is a convenient way to format these elements, and we can add all sorts of HTML elements within each `<li>` in order to achieve our desired effect.

The overall structure of each movie item is this:

```
<li class="movie-item">
  <img src="https://image.tmdb.org/t/p/w150_and_h225_bestv2/p64TtbZGCElxQHpAMWmDHkWJlH2.jpg" alt="Title of Movi
e Poster" class="poster-image">
  <h2 class="title"><a href="https://www.themoviedb.org/movie/78">Movie Title Goes Here</a></h2>
  <div class="ratings">
    <span class="rating-category critics-choice">Critic's Choice</span>
    <span class="rating-category well-liked">Well Liked</span>
    <span class="rating-category stinker">Stinker</span>
    <span class="vote-average">8.4</span> with <span class="vote-count">3676</span> votes
  </div>
  <p class="overview">
    Movie overivew goes here. Aenean tellus metus, bibendum sed, posuere ac, mattis non, nunc. Cras dapibus. Pr
aesent porttitor, nulla vitae posuere iaculis, arcu nisl dignissim dolor, a pretium mi sem ut ipsum.
  </p>
  <p class="release-date">Original Release: 1999-12-31</p>
  <ul class="genre-list">
    <li>genre</li>
  </ul>
</li>
```

Loop Through the Movies

The first thing we need to do is set up the loop that will loop through all of the movie objects in the `results` array. To accomplish this, we will use the `v-for` directive. We will apply this directive to the `<li>` tag so the `<li>` and all the elements contained within it will be duplicated for each item in the array. Here is how that `v-for` directive is written:

```
<li class="movie-item" v-for="result in results">
```

On each iteration of the for loop, an item from the `results` array will be made available as `result`. This pattern of naming is very common in software development, but it's not the only way to handle it. We could, for example, choose to call each item `movie` and write: `v-for="movie in results"`. This would be fine from a stylistic point of view, but the important thing to remember is that we are choosing the name we will use to reference the item within the loop, and we must be consistent. For the purposes of this walkthrough, we will use `result` to reference each item, and we will write our `v-for` directive as shown above.

Poster Images

The next element we encounter in the template is the `<img>` tag. This element places an image on the page, and we want to fill in two attributes on the `<img>` tag with dynamic data. We will need to use `v-bind` to bind a data value to the attribute. This can look confusing at first, but both of these examples utilize basic JavaScript string concatenation that will become more natural with practice. Here is what the augmented `<img>` tag should look like:

```
<img v-bind:src="'https://image.tmdb.org/t/p/w150_and_h225_bestv2'+ result.poster_path" v-bind:alt="result.titl
e + 'Poster'" class="poster-image">
```

The first `v-bind` directive is applied to the `src` attribute. This attribute requires the `result.poster_path` value to be concatenated with the base URL for the images on TMDb. If this value is not calculated correctly the image will not load. Pay close attention to the way that single quotes are used inside the double quotes defining the attribute value. This is a bit of a weird-looking syntax, but it is commonly used with Vue.js directives.

The second `v-bind` directive is applied to the `alt` attribute of the image. This allows us to provide a dynamic value that can be used to describe the image for screen readers. It's crucial to remember that your dynamic data needs to be applied to accessibility features, too.

Title Link

The next element in the template is the title of the movie. Each title should link back to the TMDb page with more information about the film. To accomplish this, we use an `<a>` tag inside of an `<h1>` tag. The link must have `v-bind` applied to the `href` attribute in order to dynamically build the link to the movie information page. Here is what that looks like:

```
<h2 class="title"><a v-bind:href="'https://www.themoviedb.org/movie/'+result.id">{{ result.title }}</a></h2>
```

The `v-bind` directive is applied to the `href` attribute. We see a similar single-quote syntax to concatenate the base URL with the `result.id` value. This forms a link that we can click to visit the TMDb movie information page. Inside the link tags we have interpolated the `result.title` value. This interpolation is simple, and we can use the basic mustache syntax to make it happen.

Ratings Information

In order to show a more interesting form of ratings information, we want to provide some additional data processing. We can do this right in the template to display an extra badge that shows a general ranking for the movie based on the `result.vote_average`. If the average score for the movie is over 8, then we want to call it a "Critic's Choice." If it is between 7 and 8, we want to call it "Well Liked." And if it's below 7 we want to call it a "Stinker". It's a harsh system, but movies are a tough business. Feel free to break down those categories further in order to practice more with writing conditionals.

Here is what the finished conditional set looks like:

```
<div class="ratings">
  <span class="rating-category critics-choice" v-if="result.vote_average > 8">Critic's Choice</span>
  <span class="rating-category well-liked" v-else-if="(result.vote_average > 7) && (result.vote_average <= 8)">
Well Liked</span>
  <span class="rating-category stinker" v-else>Stinker</span>
  <span class="vote-average">{{ result.vote_average }}</span> with <span class="vote-count">{{ result.vote_coun
t }}</span> votes
</div>
`
```

For "Critic's Choice" we use the `v-if` directive with the conditional statement `result.vote_average > 8`. This is a straightforward conditional statement, and it's written basically the same as we would do it in JavaScript. If the conditional is true, then this `<span>` will be shown.

The next `<span>` has a `v-else-if` directive applied to it. This means that the `<span>` will be shown if the `v-if` above it evaluates to `false` and if the conditional statement `(result.vote_average > 7) && (result.vote_average <= 8)` evaluates to `true`. If the score of the movie is between 7 and 8, then this `<span>` will be shown.

Finally, there is a `<span>` with a `v-else` statement. This `<span>` will only be shown if the two previous conditionals have evaluated to `false`. The `v-else` allows us to catch the cases where our other conditionals have failed and provide a solid fallback display of information.

We finish out the reviews section with a simple interpolation of the data for `result.vote_average` and `result.vote_count`. This information is helpful to a user who is trying to learn about a movie, so it's worthwhile to output even after we have given our overall recommendation.

We could test these conditionals further by going into the data and altering the scores for some of the movies. In a real-world API request setting we would need to identify movies that fall into each area of the spectrum and then make requests that would pull those films into our display in order to test out our logic.

Movie Overview and Release Date

The next data values we must output are the `result.overview` and `result.release_date`. These are simple data interpolations that we can make. Things would be more complicated if HTML were allowed in `result.overview`, but since TMDb uses plain text in their overviews we don't need to worry about interpolating HTML content. We could enhance the formatting of the release date, too, but this would require the creation of a filter or method to process the data for display.

The simple interpolation looks like this:

```
<p class="overview">
  {{ result.overview }}
</p>
<p class="release-date">Original Release: {{ result.release_date }}</p>
```

Genre List

The last part of the movie information that we need to output is the genre list. This list is stored in the `result.genres` array, and we will use another `v-for` loop (within our original loop) to output this value. Here is what it looks like:

```
<ul class="genre-list">
  <li v-for="genre in result.genres">{{ genre }}</li>
</ul>
```

This time, we are looping through the array `result.genres`, and we are calling each item in the array `genre` as it moves through the loop. The loop is applied, once again, to an `<li>` element using the `v-for` directive. Because each item in this array is a simple string value, we can output `{{ genre }}` directly.

It is very common for data to have properties that contain arrays of items. Those items can be data of any time, from simple Strings, Numbers, and Booleans to more complex Arrays and Objects. It's crucial to become comfortable with looping through different data objects and accessing different types of data when we need it. There are many situations where we can have no idea how many of something might exist, and the only way to display everything is to use a loop.

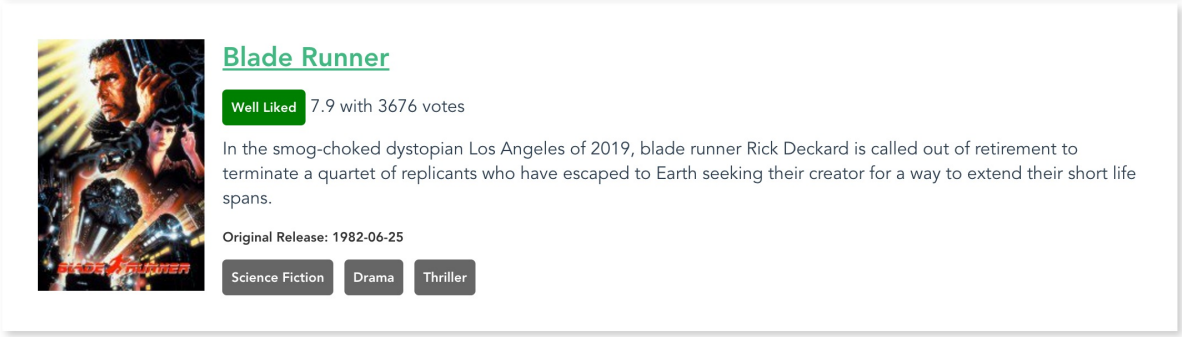
Wrapping it Up

The final result of all this work is the following finished template:

```
<template>
  <div class="results">
    <h1>Top Movies from the 20<sup>th</sup> Century</h1>
    <p class="search-meta">
      <span class="current-page"><b>Current Page:</b> {{page}}</span>
      <span class="total-pages"><b>Pages:</b> {{total_pages}}</span>
      <span class="total-results"><b>Count:</b> {{total_results}}</span>
    </p>

    <ul>
      <li class="movie-item" v-for="result in results">
        <img v-bind:src="'https://image.tmdb.org/t/p/w150_and_h225_bestv2'+ result.poster_path" v-bind:alt="res
ult.title + 'Poster'" class="poster-image">
        <h2 class="title"><a v-bind:href="'https://www.themoviedb.org/movie/'+result.id">{{ result.title }}</a>
```

```
</h2>
        <div class="ratings">
          <span class="rating-category critics-choice" v-if="result.vote_average > 8">Critic's Choice</span>
          <span class="rating-category well-liked" v-else-if="(result.vote_average > 7) && (result.vote_average
 <= 8)">Well Liked</span>
          <span class="rating-category stinker" v-else>Stinker</span>
          <span class="vote-average">{{ result.vote_average }}</span> with <span class="vote-count">{{ result.v
ote_count }}</span> votes
        </div>
        <p class="overview">
          {{ result.overview }}
        </p>
        <p class="release-date">Original Release: {{ result.release_date }}</p>
        <ul class="genre-list">
          <li v-for="genre in result.genres">{{ genre }}</li>
        </ul>
      </li>
    </ul>
  </div>
</template>
```

This template should result in a display that looks like this:



Completed movie item display

We should see dynamic data for each movie in our set of results: A unique poster image, the proper title, and all of the other unique data. The link to the TMDb movie info page should work properly, and users should be able to click and discover even more information about the movie.

Build and Deploy

Once we've finished our work, we can build and deploy the project. This project has been configured to build to the `docs/` directory, so we can follow the same pattern we used before:

1. Execute the `npm run build` command to build the files into the `docs/` directory.
2. Commit all of our code.
3. Push the code up to GitHub.
4. Go into the repository settings and set the GH Pages section to publish from the `docs/` directory.

The project should now be up and available to the public through GH Pages.

Stretch Goals

There are many more fun things we can do with this project. The `README.md` file lists several possible stretch goals:

- Figure out how to apply the "backdrop" image in some appealing way as a background for each movie
- Look up how to write a filter to format the Release Date value and make it look nicer

- Create a better simulation of pagination than the existing metadata at the top of the page
- Alter the base URL of the poster images to pull larger or smaller images (be sure to keep them in proper scale)
- Enhance some of the styles to give this project a nicer visual appeal
- Modify the template to make the ratings information be displayed as a percentage bar (or as a circle diagram, or something else) using a computed value

Much of the information we would need to accomplish these goals is presented in the previous sections of this book. We should feel empowered to explore the Vue.js Guide and other resources to fill in details and discover ideas for accomplishing these goals.

Good luck exploring and have fun pushing this as far as we can!

Handling User Input

In most websites or applications the system relies on input from the user. At a base level, we can classify the clicks and hovers a user makes as input: We often detect and respond to these actions in order to provide a responsive and engaging interface. However, sites and applications often need to gather some information from the user as structured data. To accomplish this, we generally use the many form-based tools that HTML makes available.

Within HTML we have a few key elements that allow us to gather input from the user. The `<form>` tag defines a form within an HTML document. Forms contain "fields", which are input elements the user can use to submit data. The `<input>` tag defines a wide variety of data input interfaces. The somewhat oddball `<textarea>` and `<select>` tags round out the set of default form interfaces supplied in standard HTML.

For the most part, we can accomplish the goal of collecting input from the user with these default form elements. When necessary, we can create custom input elements, or entirely custom interfaces for submitting data to the system. If our use cases get too exotic, we can always build our own mechanisms to handle those requirements.

It's important for web developers to become familiar with the way that forms work and to be able to leverage the abilities of forms to accomplish their goals. Forms are a fundamental part of the web, and even when they don't appear in the default style, they are behind almost all instances of user input.

Core Concept: Forms and User Input

In order to get the most out of the form abilities in Vue.js, it's crucial to understand what forms do and how they work in the context of a modern JavaScript application. In this section we will cover the basics of the default form elements in HTML and how they are used. We will also look briefly at how we detect user actions with events, especially events related to form submission and input changes.

How Forms Work

As a quick review of how we use forms in HTML, let's take a quick look at the fundamentals of how forms work. It's worthwhile to consult additional resources if this is not something we have a lot of experience with at this point. The Mozilla Developers Network has a great Introduction to Forms in their Learn Web Development series.

Forms are defined with the `<form>` element. Any HTML between the opening and closing `<form>` tags is part of the form itself, and whatever fields are defined between those tags are included as part of the data payload when the form is submitted.

Form Attributes

On each `<form>` element we can specify certain attributes. The following attributes are the most commonly used:

- `action` — This attribute specifies a URL that the form is submitted to. When the user submits the form, the browser will pass the form data to the specified `action` URL and the user will be directed to that URL at the same time. This has the effect of moving the user to a different page where the data is usually processed and a response is generated. In JavaScript applications, we often do not use the `action` attribute because the form will be processed by the JavaScript without redirecting the user to another URL.
- `method` — The form's `method` specifies how the data will be packaged when the form is submitted. The two possible values for `method` are `post` and `get`. Values sent with the `post` method are packaged as part of the HTTP request, and they are not directly visible to the user. Values sent with the `get` method are encoded into the URL as query string variables, and they are directly visible to the user.
- `enctype` — The `enctype` determines how the data will be encoded; it is short for "encoding type." The default value if unspecified is `application/x-www-form-urlencoded`. If we are submitting files with our form (using the `input` type of `file`), we must change the `enctype` to `multipart/form-data` so that the file data can be properly packaged.

There are other form attributes available, but these are the most commonly seen. A common form declaration might look like this:

```
<form action="/login/" method="post">
    <label>Username <input type="text" name="username" tabindex="0"></label>
    <label>Password <input type="password" name="password" tabindex="0"></label>
    <input type="submit" value="Login">
</form>
```

In the example above we can see a sample login form. The `action` property is set to a URL that will handle a login, and the method is `post` because we don't want to expose the username or password.

Form Layout

In order to help with readability and styling of forms, there are a couple of HTML elements that are meant for organizing fields in a form. These tags can be used to create visually separated sets of fields in the HTML page, and they can be styled to achieve whatever visual presentation is desired.

The `<fieldset>` element defines an area in the form. It is meant to be wrapped around a group of fields, and it can be used to style or disable/enable those fields in bulk. Fieldsets are often augmented by the `<legend>` element. This element defines a caption that is shown as part of the fieldset and is meant to help users understand the purpose of the grouping. The legend serves a purpose similar to labels on individual fields.

Here is an example of a fieldset:

```
<form action="/profile/update/" method="post">
    <fieldset>
        <legend>Personal Information</legend>
        <p><label>Salutation <input type="text" name="salutation" tabindex="0"></label></p>
        <p><label>First Name <input type="text" name="firstname" tabindex="0"></label></p>
        <p><label>Family Name <input type="text" name="familyname" tabindex="0"></label></p>
    </fieldset>
    <fieldset>
        <legend>Address</legend>
        <p><label>Street 1 <input type="text" name="street1" tabindex="0"></label></p>
        <p><label>Street 2 <input type="text" name="street2" tabindex="0"></label></p>
        <p><label>City <input type="text" name="city" tabindex="0"></label></p>
        <p><label>State <input type="text" name="state" tabindex="0"></label></p>
        <p><label>Postal Code <input type="text" name="postcode" tabindex="0"></label></p>
    </fieldset>
</form>
```

The HTML above produces the following form (with default HTML styles):

Personal Information

Salutation

First Name

Family Name

Address

Street 1

Street 2

City

State

Postal Code

As we can see, the fieldsets provide some visual definition for the form, and the legends allow the user to quickly understand what is being requested in each section of the form. We often deal with complex forms that benefit from this sort of structuring, and we can apply any styles we wish to these elements to make them match our visual design.

Labels

As we see in the examples above, `<label>` elements are used to wrap the names of form fields. Labels serve multiple purposes in an interface, and they should never be omitted. Labels can be used to make it easier to click into a form field (clicking the label activates a form field). This feature is often very helpful, especially when used in

conjunction with checkboxes in forms (which are often fiddly to click on). Labels are also used by screen readers to identify the purpose of form fields. This is why labels should never be omitted from a form: Without labels, we restrict accessibility of our forms.

There are two ways to apply labels. In the examples above, the `<label>` tag wraps the input that it is paired with:

```
<label>Username <input type="text" id="username" name="username" tabindex="0"></label>
```

The other method for labeling a form field is to use the `for` attribute on the `<label>` tag. The `for` attribute should reference the `id` of the field the label is meant to be related to. Here is an example:

```
<label for="username">Username</label> <input type="text" id="username" name="username" tabindex="0">
```

By using the `for` attribute, we get the same effect as wrapping the `<input>` with the `<label>` tags. Users can still click the label to activate the field, and screen readers can relate the label and field properly using the `for` attribute. This is also useful if we wish to achieve a visual design that does not have labels (for example, when we use the `placeholder` attribute to indicate what data should go into a form field). With the second structure we can visually hide the label but still make it available to screen readers and other assistive technology.

We can see that the end result of both approaches is the same:

Username

Username

Accessibility Tip: `tabindex="0"`

Whenever we want to make sure users can navigate our sites and apps via keyboard, we can add the attribute `tabindex` to our HTML elements. This is usually added to form fields, buttons, links, and other interactive elements. When we are coding applications in JavaScript we might use elements in unorthodox ways, which makes it more important for us to pay attention to adding `tabindex="0"` to our tags.

The `tabindex="0"` attribute tells the browser that this field should be included in the list of things that the user can access using the `tab` key on their keyboard. The value `"0"` tells the browser that this field should be accessed in order of its appearance in the HTML. It is generally better to leave the ordering of elements up to the browser unless we are very familiar with how `tabindex` works.

Many users who wish to interact with our sites are reliant on accessibility tools and keyboard navigation. We can serve these users much better by making sure to enable tabbing to any field, button, link, or interactive element in our pages.

The `<input>` Element

The real workhorse of form field elements in HTML is the `<input>` element. This is the element that handles most of the data input from users. The input element is relatively simple to write, but it allows for a lot of types. It's useful to know about the available types of input in order to build much more useful interfaces for our users. At this point browsers and HTML support many helpful interfaces for data input.

The basic `<input>` element is written with a few common attributes:

```
<label for="username">Username</label> <input type="text" id="username" name="username" placeholder="you@domain
.com">
```

This HTML would be displayed like this:

Username you@domain.com

We can see the most common attributes for a form field in this example. The `type` attribute determines which input interface is shown. Since this example specifies `"text"` the one-line text box is displayed. The `name` attribute specifies the name that will be used for this field in the form data payload. The form data contains values for every form field submitted, and they are referenced according to their respective names. The `name` attribute also defines the name that must be used when setting the `for` attribute of the corresponding `<label>` tag. Finally, the `placeholder` attribute defines the text that will be shown inside the field. In this case, the placeholder text attempts to let users know they should type in their email address as their username.

Common `<input>` Attributes

There are many attributes that can be used with the `<input>` element to accomplish whatever goals we have, but there are a few attributes that we use often. These attributes are worth memorizing:

- `type` — This attribute determines the type of input interface that will be shown. It should always be set on every input.
- `value` — This attribute contains the initial value of the field. If the user does not alter the value of the field, this value is used when the form is submitted.
- `required` — The required attribute makes an input field required. The form will not submit until it is populated with data.
- `disabled` — Sometimes we want to show an input as disabled until some event occurs. This attribute prevents the user from modifying or entering information in an input.
- `autocomplete` — This attribute allows developers to control whether or not fields should allow the browser to autocomplete them with saved information from the user (e.g. name, address, email, etc.). Developers can also label which type of data should be used to autocomplete the field, allowing for a better user experience.
- `spellcheck` — Indicates to the browser if it should do spelling and grammar checks on the content of an input.
- `checked` — Used with checkboxes, this attribute indicates the box is checked.
- `maxlength` & `minlength` — These two attributes allow the developer to specify maximum and minimum lengths for the data submitted (usually used with `text` fields).

Types of Inputs

The `type` attribute on an `<input>` tag determines the input interface the browser presents to the user. This can be varied according to what kind of information we expect the user to submit. It's useful to specify the proper type of input interface, especially for users on a variety of devices. Although there's not much difference between using a keyboard to enter an email and a phone number on a laptop, on a mobile device it's much more useful to present the user with either an alphabetic keyboard or a numeric keyboard according to the input type.

The following types of inputs are useful to know about:

- `button` — A push button with no default behavior.
- `checkbox` — A check box allowing single values to be selected/deselected.
- `color` — A control for specifying a color. A color picker's UI has no required features other than * `accepting simple colors as text (more info).
- `date` — A control for entering a date (year, month, and day, with no time).
- `datetime-local` — A control for entering a date and time, with no time zone.
- `email` — A field for editing an e-mail address.

- `file` — A control that lets the user select a file. Use the accept attribute to define the types of files that the control can select.
- `hidden` — A control that is not displayed but whose value is submitted to the server.
- `image` — A graphical submit button. You must use the src attribute to define the source of the image and the alt attribute to define alternative text. You can use the height and width attributes to define the size of the image in pixels.
- `month` — A control for entering a month and year, with no time zone.
- `number` — A control for entering a number.
- `password` — A single-line text field whose value is obscured. Use the maxlength attribute to specify the * `maximum length of the value that can be entered.
- `radio` — A radio button, allowing a single value to be selected out of multiple choices.
- `range` — A control for entering a number whose exact value is not important.
- `reset` — A button that resets the contents of the form to default values.
- `search` — A single-line text field for entering search strings. Line-breaks are automatically removed from the input value.
- `submit` — A button that submits the form.
- `tel` — A control for entering a telephone number.
- `text` — A single-line text field. Line-breaks are automatically removed from the input value.
- `time` — A control for entering a time value with no time zone.
- `url` — A field for entering a URL.
- `week` — A control for entering a date consisting of a week-year number and a week number with no time zone.

By using the appropriate type of input, we can make sure the user sees the correct interface. Here is a quick example:

```
<p><label>Color <input type="color"></label></p>
<p><label>Telephone Number <input type="tel"></label></p>
<p><label>Month <input type="month"></label></p>
<p><label>Range <input type="range"></label></p>
```

This is what those fields look like on the page. Click them to see the interfaces they bring up (especially if you are viewing this on a mobile device).

Color

Telephone Number

Month

Range

Exceptional Fields

There are a couple of exceptional fields we use often in web development. These input fields are not created with an `<input>` element, but they share many of the same attributes and basic usage. We still use labels with them, although usually with a `for` attribute, and they still get folded into the form data alongside the other inputs.

`<textarea>`

The `<textarea>` tag is seen almost anywhere we can write any significant amount of text. This field shows up anywhere we type a message, write a review, create a post, etc. The `<textarea>` tag is a little strange because it does not use a `value` attribute to store its starting data. Rather, it wraps its value between the opening and closing `<textarea>` tags. Here's an example:

```
<label for="quote">Quote:</label>
<textarea name="mlkquote" id="quote" cols="80" rows="4">
Darkness cannot drive out darkness; only light can do that. Hate cannot drive out hate; only love can do that.
-- Dr. Martin Luther King, Jr.
</textarea>
```

And here's the rendered field:

Quote: Darkness cannot drive out darkness; only light can do that. Hate cannot drive out hate; only love can do that. -- Dr. Martin Luther King, Jr.

In this example we see the quirky way that `<textarea>` wraps the value in its tags. We also see that it does not use `width` and `height` like most fields. It uses `cols` (the number of vertical columns) and `rows` (the number of horizontal rows) to determine the height. For a long time now browsers have supported resizing textareas, which is a feature that can be controlled via CSS.

`<select>`

Another input type that does not stem from an `<input>` tag is the `<select>` input. We see select inputs as either "drop down lists" or "multi-select lists" on websites, and it is a very common interface. In order to create a full select input, we must define some `<option>` elements. In applications and websites we often populate these options from our system data. Here is the structure of a typical `<select>` input:

```
<label for="petType">Type of Pet:</label>
<select id="petType" name="petType">
    <option value="dog">Dog</option>
    <option value="cat">Cat</option>
    <option value="pig">Pig</option>
</select>
```

The code above would produce the following:

Type of Pet: Dog

This is a typical "drop-down list" style of selection. We could turn it into a mutli-select interface by adding the `multiple` attribute to the `<select>` tag like this:

```
<label for="petType">Type of Pet:</label>
<select id="petType" name="petType" multiple>
    <option value="dog">Dog</option>
    <option value="cat">Cat</option>
    <option value="pig">Pig</option>
</select>
```

The code above would produce the following:

Type of Pet: Dog Cat Pig

Select inputs can use many of the same input attributes such as `required`. Option elements are denoted with the `selected` attribute (similar to how checkbox inputs use the `checked` attribute). We can specify a default value for a select list by using the `selected` attribute:

```
<label for="petType">Type of Pet:</label>
<select id="petType" name="petType">
```

```
    <option value="dog">Dog</option>
    <option value="cat" selected>Cat</option>
    <option value="pig">Pig</option>
</select>
```

The code above would produce the following:

Type of Pet: Cat

Form and Input Events

Although we've reviewed the basics of form creation, presenting the form to the user is only half the challenge. We must also respond to events prompted by the user entering data into the form. When working with forms we care most about the form submission event (`submit`), the field `change` event, and the field `input` event. Let's take a closer look at each of these events.

`submit`

The `submit` event is fired whenever a form is submitted. This event is handled by the browser by requesting the URL specified in the form's `action` attribute. The browser packages the form data into a payload that is included in the request. The structure of the data payload depends on the `method` attribute specified on the `<form>` tag.

In JavaScript applications we generally intercept the form submission, and we often never specify another URL in the `action` attribute. Most frameworks, including Vue.js, provide tools to easily prevent the default behavior of the browser and handle the form data with our custom logic.

`input`

The `input` event is signaled when the user modifies any information in a form field. This would apply to every letter the user types into a `text` input, or any selection the user makes in a `select` input. Sometimes this behavior is very useful, such as when a system autocompletes search results or email addresses based on the text a user is typing. Other times, this event happens too often, and it can lead to strange results.

`change`

The `change` event is emitted when the user commits a change to a field. This is interpreted differently for different input type and situations. In the case of selecting an option from a list, it works similar to the `input` event: The `change` event will fire after the user makes a selection. However, in the case of the `text` input, the `change` event is only fired after the `text` input field loses focus. This prevents the event from being fired for each character the user types. The `change` event can be a more useful way to handle some interface situations, so it's worthwhile to try varying between `input` and `change` events to note the differences.

Using Forms in Vue.js

In Vue.js we can use forms to gather data from the user, and the form fields and events can easily be connected to our component logic. To accomplish this, we create bindings between the form fields and our component data. The `v-model` directive is how we assign these relationships. By binding fields to data values, we can do a lot directly in templates, or we can create event listeners and use component methods to perform actions when the form is submitted or when another event occurs.

Basic Input Data Binding with `v-model`

The `v-model` directive associates a form input element with a variable in our component's data. This variable can be referenced within the component logic (in computed values, methods, or filters), or it can be referenced within the template for display to the user. The `v-model` directive creates a two-way binding between the value in the component logic and the value in the template, just like other variables we define in the component's `data()` function.

We can create an association like the one in this example component:

```
<template>
  <div class="forms">
    <h2>{{ message }}</h2>
    <label>Message: <input type="text" v-model="message"></label>
    <p>Computed reversed message: {{ reversedMessage }}</p>
  </div>
</template>

<script>
export default {
  name: 'FormsPractice',
  data () {
    return {
      message: 'Type in the input to change the message.'
    }
  },
  computed: {
    reversedMessage: function () {
      return this.message.split('').reverse().join('')
    }
  }
}
</script>
```

We can see here that we have a simple Vue.js component called `FormsPractice`. This component uses a data object that includes the `message` property. The `message` property is used in the template, where it is interpolated between the `<h1>` tags. The `message` property is defined in the component logic as part of the object returned by the `data()` function. There is also a computed value defined called `reversedMessage`, which also uses the `message` value.

All of these values are properly bound so that when the content of the text field is altered those changes are reflected in the HTML so the user can see them. Here is an example:

Animation of form field model binding

Since the default binding is updated whenever the form field is altered, we see immediate reaction to the user's input in the HTML. (If immediate reflection of the changed value is not desirable, we can use the `.lazy` modifier below to update the bound values only on the `change` event instead of the `input` event. See below for further explanation.)

Generating Options and Binding Groups of Inputs

There are a few kinds of form field inputs that rely upon or provide grouped information. Options for selects, checkbox groups, and radio buttons are often generated from an Array of data. Multiple selects and checkbox groups can also return an array of data for processing. Luckily, these input fields are well-connected in the Vue.js framework, and we can work with their data like any other data in the system.

We can use the `v-bind` directive to populate a form field with saved data. In many of the examples below, we use the `v-bind` directive to attach values to `<option>` and checkbox fields that are generated by looping through a data Array. As with any other HTML element attribute, we can use `v-bind` to create a binding between some value or computation and the attribute itself. This is a common case that we encounter regularly in developing forms for our applications, so it's worth noting that we can easily use `v-bind` with the `value` attribute.

Here is an example of a simple `<select>` field setup:

```
<template>
  <div class="forms">
    <h2>Selected: {{ petSelection }}</h2>
    <label for="petChooser">Pick a pet:</label>
    <select v-model="petSelection">
      <option disabled value="">Please select one</option>
      <option v-for="pet in petList" v-bind:value="pet">{{ pet }}</option>
    </select>
  </div>
</template>

<script>
export default {
  name: 'FormsPractice',
  data () {
    return {
      petList: [
        'dog',
        'cat',
        'pig'
      ],
      petSelection: 'pig'
    }
  }
}
</script>
```

In this example, we have a component that defines a `petList` Array and a String called `petSelection`. The template displays the current selection as text, and the `<select>` element is bound to the `petSelection` value. The `<option>` elements are generated using a `v-for` loop. (Note that the `v-bind:value` directive can access the item in each iteration of the loop.) The result is an interface that works like this:

Bound select input

The selection list is automatically initialized with the current value of the `pet` value specified in the `v-model` attribute. When the value is updated, the output is correspondingly updated. Similarly, we can look at an example of a checkbox group to get an idea of how that works.

```
<template>
  <div class="forms">
    <h2>Selected: {{ petSelection }}</h2>
    <label v-for="pet in petList">
      {{ pet }}
      <input type="checkbox" v-model="petSelection" v-bind:value="pet">
    </label>

  </div>
</template>

<script>
export default {
  name: 'FormsPractice',
  data () {
    return {
      petList: [
        'dog',
        'cat',
        'pig'
      ],
      petSelection: ['pig']
    }
  }
}
</script>
```

In this slightly modified version of the previous example, we have altered the `petSelection` value to be an Array. We have initialized its value with one item: `['pig']`. In the template, rather than looping the `<input>` tag, we have applied the `v-for` loop to the `<label>` tag so we can get proper wrapping of our checkbox inputs. The `v-model` attribute for each checkbox input is set to the same value, so all the values the user clicks will be added to the `petSelection` Array. Finally, the value of each item in the `petList` Array is bound to the `value` attribute using the `v-bind:value="pet"` directive.

Here is what it looks like in action:

Checkbox group example

Notice that the value the `petSelection` Array was initialized with is pre-checked when the input is displayed to the user. The Vue.js binding system keeps the checkboxes updated according to the value of the context variable each input is bound to in the template.

> ### Include a Dummy Choice!
>
> Due to the way some devices handle events on `
>
> This technique insures that we will always receive a data update when the user makes a selection. Without the dummy choice, it's possible that the user would want to select the first item in the list, and the device or browser might not always trigger a data sync throughout our Vue.js application.

Model Modifiers

Sometimes we need to modify the data, or how we collect the data, in a predictable way. Vue.js provides three modifiers we can use with the `v-model` directive to change the way data is bound. These allow us to provide a better user experience in many situations.

`.lazy`

The `.lazy` modifier allows us to alter the way data is updated throughout the system upon user input. Normally, a bound input field will update data on the `input` event fired by the field. This can be beneficial, but sometimes it is not desirable. When we would prefer to fire the event on the `change` event (as opposed to the `input` event), we can use the `.lazy` modifier.

Here is what this would look like on a text input:

```
<input type="text" v-model.lazy="username">
```

`.number`

The `.number` modifier insures the value is typecast as a Number when it enters the component logic. By default, any information coming out of a text input field will be typed as a String in our JavaScript logic. If we use the `.number` modifier, we can be sure our calculations will work as planned. Here is an example:

```
<input type="number" v-model.number="myNumber">
```

Although this example uses the input type `number`, **all** HTML inputs are typed as Strings when they are brought into the Vue.js component logic. If we wish for a value to be cast as a Number, we must use the `.number` modifier.

`.trim`

We can use the `.trim` modifier to remove extra whitespace from the beginning and end of a user's data. This is often used on login forms when we wish to discount any leading or trailing spaces that might result from a user's copy/paste. We can apply this modifier just like the others:

```
<input type="text" v-model.trim="username">
```

Handling Events in Vue.js

Events are a key part of any web-based application. The user interacts with a page by scrolling, clicking, hovering, touching, dragging, submitting forms, and more. All of these interactions can be captured. Responding to these events is a primary method we have for building reactive interfaces that help the user accomplish their goals.

As a quick recap about what events are: Many components of HTML will emit (signal, or "fire") "events." Events are meant to provide a signal that something interesting has happened, and much of what we do in our coding is to create situations where responding to these built-in events becomes useful to the user. For example, when a button is clicked, a "click" event is fired. This event can be detected in our code and we can respond by doing something the user expects.

Events are "handled" by event listeners (which we will often just call "listeners"). Listeners are specific watchers we have defined to detect an event and then execute some code in response to the event. Events are happening all the time when a user interacts with a website, but if no listeners have been defined then only the default event "handlers" will be invoked.

For example, a web browser will automatically submit a form when the form emits the `submit` event. A form could be submitted by a user pressing `enter` on their keyboard, or by clicking a button to submit the form. Regardless, the browser has a default event listener watching for that signal from any HTML forms, and it will execute code to respond to an event.

As developers, we can also detect events and we can choose to augment or replace the default response to any event. Vue.js provides us with a range of tools for working with events. These tools make it much easier to define, manage, and execute event handling in Vue.js projects.

Defining Event Listeners

The core directive used to define event listeners is the `v-on` directive. This directive takes an argument, which is the name of the event for which to listen. Because each HTML element can emit a different set of events, it's best to refer to a list of relevant events (such as the Web Events page at MDN) to know we are using the proper event label. But much of the time we use relatively standard events in our projects (e.g. `submit`, `click`, `keydown`, `keyup`, etc.).

Here is an example of a simple `v-on:click` listener that increments the value of a `counter` variable in the template context:

```
<h2>Counter</h2>
<button v-on:click="counter++">Add 1</button> {{ counter }}
```

All of the logic that responds to this event is in the template here. As long as `counter` is initialized to `0` in the component's data, no other logic is required in the component to make this work. When the template loaded in a browser, it looks like this:

Simple click event handler

This is easy enough to set up, but usually we want to do something more with our event handlers. Typically, rather than executing a simple line of code when an event is detected, we execute a component method to handle the event. Component methods can be used in many ways, and they are straightforward to add to our components.

Component Methods

Component methods are functions that can be executed in various ways within a Vue.js component. Much of the time, these methods will be executed in response to some event that has been triggered in the system. The `v-on` directive can define a method call that will be executed in response to an event signal. Let's look at an example:

```
<template>
  <div class="events">
    <h2>Message: {{ message }}</h2>
    <p><button v-on:click="setMessage('Danger! Danger!')">Show Danger Message.</button></p>
    <p><button v-on:click="setMessage('All Clear!')">Show All Clear Message.</button></p>
  </div>
</template>

<script>
export default {
  name: 'EventDemos',
  data () {
    return {
      message: 'Component loaded successfully.'
    }
  },
  methods: {
    setMessage: function (text) {
      this.message = text
    }
  }
}
</script>
```

In this example, there is only one variable revealed to the template: `message`. The message is initialized to a load statement, but the user can click a button to change the message. Each button has a `click` event listener defined using the `v-on` directive. When the `click` event is detected on one of the buttons it will execute the `setMessage` method. This method is a function that expects to receive some text to change the value of the `message` variable. In this example, the text is set by hand, but this text could also be provided by referencing another variable or data point.

Here is what this code looks like when displayed to the user:

Event handler using component method

We can see that when the user clicks the buttons the message is changed instantly. This demonstrates the fundamental ability to use methods to respond to events. With this tool, we can make all sorts of things happen in our interfaces.

Preventing Default Event Handling

Although we can now respond to events, there are situations when we will find ourselves fighting against the default event handling provided by the web browser. Vue.js provides us with several modifiers we can use alongside the `v-on` directive to prevent the default actions from taking place. We should keep the following modifiers in mind as we work through trickier event handling situations.

`.prevent`

The `.prevent` modifier will prevent the default event handlers from executing when an event is triggered. This is most commonly used to prevent form submissions. If no `action` property is supplied on a `<form>` element, the browser will still refresh the page. That refresh could be enough to really interrupt our well-planned user experience. Luckily, it's easy to prevent the default form submit event from happening thanks to `.prevent`.

```
<template>
  <div class="forms">
    <h2>Message: {{ message }}</h2>
    <form v-on:submit.prevent="handleMyForm">
      <p><label>New Message: <input v-model="newMessage" type="text"></label></p>
      <p><input type="submit"></p>
    </form>

  </div>
</template>

<script>
export default {
  name: 'FormsPractice',
  data () {
    return {
      message: 'Submit the form to change this text.',
      newMessage: ''
    }
  },
  methods: {
    handleMyForm: function () {
      this.message = this.newMessage
    }
  }
}
</script>
```

Rather than directly binding the `message` value to the form input, this example uses a component method to handle the form submission. The method is called `handleMyForm` and it changes the value of the `message` variable. Note the use of `.prevent` on the `<form>` element's event listener: This is used to prevent the browser from refreshing to submit the form.

Here is what this example looks like to the user:

Using .prevent on a form

`.stop`

In some situations there could be multiple event handlers that could be triggered when we perform a single action. It's common for HTML elements and their parent or child elements to each have events defined, and in those cases the event "bubbles" up the DOM from child to parent to grandparent and beyond. This could lead to multiple event handlers being executed in quick succession when a single event is triggered.

In order to stop the bubbling of events through the DOM, JavaScript has a `stopPropagation` method that we use. In Vue.js, we can simply apply the `.stop` modifier to a `v-on` directive. This will stop the bubbling of the event through the DOM and prevent any other event handlers from firing. Here is an example of how this looks:

```
<a v-on:click.stop="handleMyClick">Click me</a>
```

Note: This is different from `.prevent`, which only prevents the *default* event handlers from firing. The `.stop` modifier will stop all other events handlers from being executed. It is sometimes desirable to both stop the event from propagating through the DOM and to prevent the default event handlers from executing. Luckily, directive modifiers can be chained in Vue.js:

```
<a v-on:click.stop.prevent="handleMyClick">Click me</a>
```

It's important to keep in mind that chained modifiers are possible with any modifier, and that they are executed in the order they are chained. The order of chaining modifiers can dramatically alter their effect, so we must pay close attention to our ordering of modifiers.

`.once`

It is possible that we have an event we only wish to trigger once. For example, submitting a payment form should only be done once, but it's common for users to double-click the submit button or click submit a second time after they wait for a moment. We might use the `.once` modifier to prevent that behavior:

```
<form v-on:submit.prevent.once="handleMyForm">
```

We can see that we can also chain the `.once` modifier with the `.prevent` modifier to fully handle the form with our custom component method.

There are additional modifiers available, but these are the most commonly used. Refer to the Vue.js Events Guide for more details about modifiers.

Detecting Keyboard Input

Keyboard events are helpful for building an interface that is both efficient for users prefer keyboard navigation and accessible to users who are forced to use keyboard navigation. Adding hotkeys, keyboard shortcuts, and general navigational support via keyboard can be a great enhancement to any user experience. Detecting keyboard and device events is straightforward with Vue.js.

The events associated with keyboards are:

- `keydown` — When the key is first pressed down by the user.
- `keypress` — If the key is not a modifier (`alt`, `opt`, `ctrl`, `cmd`, etc.), this event is fired.
- `keyup` — When the key is released by the user.

Users are accustomed to these events being used in different ways. The `keydown` event is somewhat "instrumental": We expect to hear a sound when we press the key down on the piano, for example. This is also how we expect game controls to work much of the time. The `keypress` event is often used to filter out modifier keys, which is helpful for

page shortcuts, and this is the way we expect keyboard combinations to work. The `keyup` is the event that we most associate with single-key keyboard commands: A command executes when we release the key, not when we first press it. It's worthwhile to experiment with these different events to get a better feel for how keyboard interfaces work.

Responding to Specific Keys

We often do not want to just randomly respond to every keyboard event. Usually, we want to set specific events for specific keys or key combinations. Keyboard events include a `keyCode` that can be used to identify the key being pressed. We can find the `keyCode` for any key using a tool like keycode.info.

Luckily, Vue.js provides quite a few aliases for these codes that make it easier to define an event listener for a specific key. For example, if we wanted our users to submit a form by pressing the `enter` key, we could use this directive:

```
<form v-on:keyup.enter="handleMyForm" v-on:submit.prevent>
```

In this example, the user would be able to either click a `submit` button or press the `enter` key to submit the form. In both cases, the form will be handled by the `handleMyForm` component method. (Also note that we can have multiple event listeners defined on a single element.)

Here is the full list of `keyCode` modifier aliases provided by Vue.js:

- .enter
- .tab
- .delete (captures both “Delete” and “Backspace” keys)
- .esc
- .space
- .up
- .down
- .left
- .right

If the key we need to respond to is not in this list of modifier aliases, we can still refer to it by the numeric code:

```
<a v-on:keyup.78="createNewItem">New</a>
```

In this example, we have mapped the key `78` (which corresponds to the `n` key) to a component method that will create a new item. We can pair this reference to a `keyCode` with a modifier key using one of the modifiers listed here:

- .ctrl
- .alt
- .shift
- .meta

We often pair keyboard shortcuts with modifier keys such as `ctrl` or `alt`. It has become increasingly common to let users do more significant actions by pressing `shift + enter`, too. We could accomplish a feature like that with a simple `v-on` directive:

```
<form v-on:keypress.shift.enter="handleMyForm">
```

In this example we have used the combination of `shift` and `enter` to handle the form submission. We could apply other keyboard shortcuts in more conventional ways, too. Here is a new item shortcut that uses the `ctrl + n` keyboard combination:

```
<a v-on:keypress.ctrl.78="createNewItem">New</a>
```

With all of the tools Vue.js gives us for handling keyboard input, it's incredibly easy to add keyboard navigation and robust keyboard features to our websites and applications. This can help us present better, more accessible experiences to our users.

`v-on` Shorthand

If we get sick of writing `v-on:click`, we can alter our templates to use the shorthand version: `@click`. Here is an example of what one of our previous templates would look like using the shorthand.

```
<template>
  <div class="events">
    <h2>Message: {{ message }}</h2>
    <p><button @click="setMessage('Danger! Danger!')">Show Danger Message.</button></p>
    <p><button @click="setMessage('All Clear!')">Show All Clear Message.</button></p>
  </div>
</template>
```

Quiz: Handling User Input

Please enjoy this self-check quiz to help you identify key concepts, points, and techniques discussed in this section.

Visit Quiz Online

The quiz on this page has been removed from your PDF or ebook format. You may take the quiz by visiting this book online.

Responding to User Input Part One

For this section's project, we will begin work on a small sequence of forms that we will finish out in the next section. The starter repository for this project is located here. As usual, we will begin by forking the repository on GitHub into our own account, and then cloning the repository to wherever we are doing our development. Once we have the project cloned into our development environment, we are ready to get started.

Review the Requirements

As with all of our projects, we begin by reviewing the `README.md` file in the root of the repository. This file will tell us the requirements we must fulfill in order to successfully complete this project. For this first part of the project, we will focus on the requirements outlined for the `Home.vue` file. We will finish the remaining requirements in the next section of the book (Routing and URLs).

In its entirety, this project asks us to create a couple of forms and move the user through multiple locations in the application to complete those forms. For this first part of the project, we will focus primarily on creating and validating the sign up form. When the user provides valid data, we will display a success message. If the user provides invalid data, we will display an error message.

The Basic Requirements for this project are broken into requirements for each of the files. Stretch goals are provided that apply to all of the work in this project. We are encouraged to push this experiment as far as possible, so adding or altering form fields, adjusting validation logic, and providing more textured error messages are all fair game for stretching this project in this section. (But remember, to fully complete the project we must complete the second part in the Routing and URLs section.)

Introduce the "views" directory

You'll notice that our Component files are located in a views directory. This is because we will be using these components represent routable views and we'll be using a router in the second half of this project. It is a convention in Vue.js to put Components referenced by the router into a `views` directory instead of the `componentents` directory. Not all components are routable and this will become more clear in future projects.

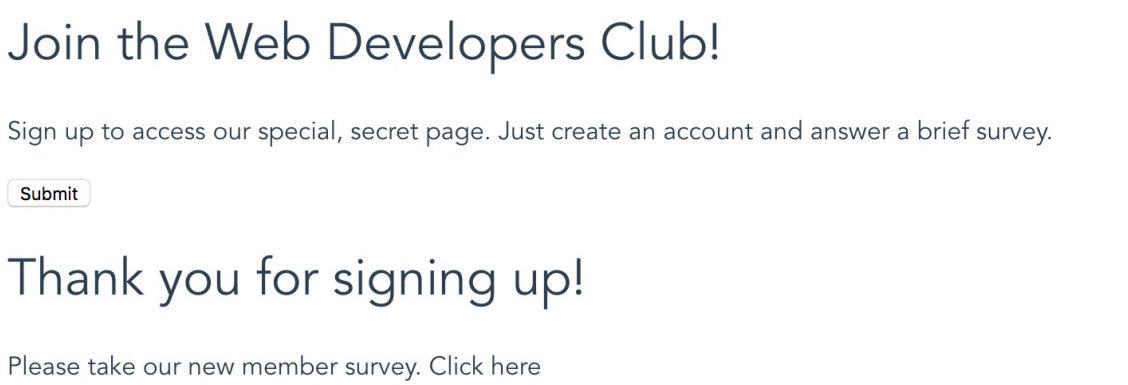
Here are the Basic Requirements relevant to this first part of the project:

In the `src/views/Home.vue` file

- Create a form for the user to sign up to the site
- Use `v-show` to toggle between the sign up form and the success text
- Use the `submit` event to trigger the `validateForm` method on the component
- Validate the form according to the rules in the `Home.vue` file
- If the form is valid, show the success message; if not, show a warning message on the form

Working the Project

To begin working on the file, it is necessary to install our dependencies (by running `npm install` inside the project repository), and then we will want to run our development server (by running `npm run dev` inside the project repository). Once we have the project running, we will see this in the browser:



Home view

Work on this part of the project will happen in the `src/views/Home.vue` file. This file contains the `Home` component, which controls the first page of our project website. In the initial state, the view contains no form input fields and it shows the success message right away. We must fill in the details to make this a functioning sign up form.

Show/Hide Form and Thank-You Message

We begin with a set of data defined in the component's `data` object. One of those data values is `showForm`, which is set to `true` by default. The first `TODO` in the file tells us to modulate the display of the form container (`<div class="form-container">`). If we glance down a little further, we can see that we also need to modulate the display of the `<div class="success-message">` element, which contains the success message the user should see after successfully submitting the form. We can tackle both of these `TODOs` at once.

First, we can add a `v-show` directive to the `<div class="form-container">` :

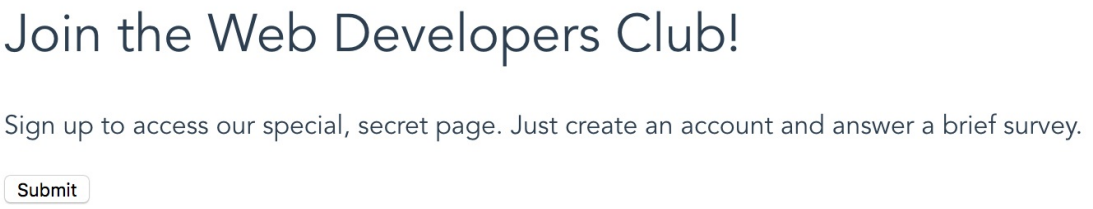
```
<div class="form-container" v-show="showForm">
```

If `showForm` is `true` (which it is at the beginning), this form container will be shown. Remember that `v-show` accepts a conditional statement, so we can use conditional operators with the `v-show` directive, too. We will do that to hide the success message until the right time.

In order to handle the show/hide of the success message, we can add another `v-show` directive to that element:

```
<div class="success-message" v-show="!showForm">
```

In this case, we want to show the success message when the form is NOT showing. Essentially, the success message is the opposite of the form container. We use the `!showForm` syntax to specify that when `showForm` is `false`, this element should be shown. Once we implement these two `v-show` directives, we should see that the success message is hidden by default:



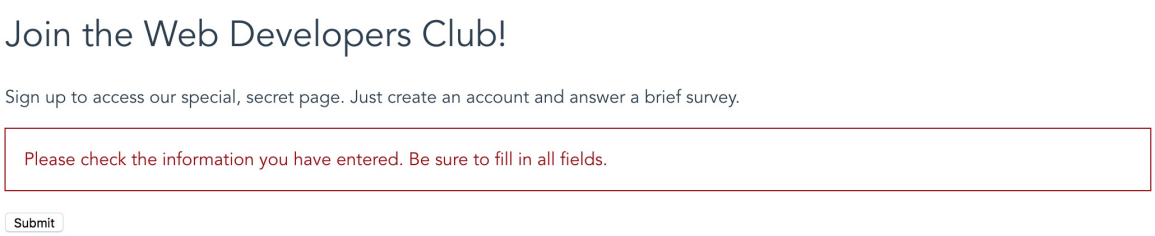
Hiding the success message

Create Error Message

The next `TODO` we come across asks us to make an error message and use `v-show` to make it appear when the form has an error. First, we must add the element that we will modulate. If we glance at the styles defined in the `Home` component, we can see there is a class defined to style an error element: `.error`. We will use that class and add the following to the component template:

```
<p class="error" v-show="showError">Please check the information you have entered. Be sure to fill in all field
s.</p>
```

By default, this warning will not show up. In order to complete our work here, we must alter the `showError` value in the component to be `true`. If we just adjust the value where it is defined in the `data` object we can see the error message:



Error message displayed

Now we can reset the initial value of `showError` to be `false` (we don't want to show the error when the user first arrives on the site), and we can alter the value in the `validateForm` method accordingly.

Form Submit Event Handler

Next, we want to set up the event handler for the form submission event. To do this, we will modify the `<form>` tag and add a `v-on` directive. This will be very similar to the examples we saw earlier. Here is what the `<form>` tag looks like after we've added the handler:

```
<form v-on:submit.prevent="validateForm">
```

Notice that we have used the `.prevent` modifier with the `v-on` directive. This modifier will prevent the default event handlers from firing in the browser when the form is submitted. If we fail to add this, the page would refresh each time the form is submitted and we would never be able to process any data. By using `.prevent` we tell the browser to stop processing the event, which allows us to handle the event with our `validateForm` method, which we will modify later on. For now, we can move on to populating our form with input fields.

Add Input Fields

The next set of `TODOs` ask us to create inputs for each field in our form. These fields must be properly bound to a value in the component `data` object, and they should use appropriate labels. This is what things look like after we've made the edits:

```
<p>
    <label for="username">Username
        <input type="text" id="username" v-model="username">
    </label>
</p>
<p>
    <label for="email">Email
        <input type="email" id="email" v-model="email">
    </label>
</p>
<p>
    <label for="password">Password
        <input type="password" id="password" v-model="password">
    </label>
</p>
<p>
    <label for="passwordVerify">Verify Password
        <input type="password" id="passwordVerify" v-model="passwordVerify">
    </label>
</p>
```

Notice that each input in this form is has a specific value in the `type` attribute. The username is a `text` input. The email is an `email` input. And the password fields are `password` inputs. Properly labelling the `type` of input fields is crucial for providing a good user experience and for collecting valid data.

Each input field uses the `v-model` directive to set up the data binding between the inputs and the component logic. These `v-model` attributes must match up with the values specified as part of the `data` object. Once we've got this set up, we are ready to move on to writing our validation.

Validating the Form

Whenever a form is submitted by a user, it's essential to perform some sort of validation process. This is done to make sure the information that has been submitted meets our needs and that no required fields have been left blank. There are many ways to validate data, but for the purposes of this project we are just looking for a simple validation that makes sure there is information in every field and that the values of `password` and `passwordVerify` are equal. (Feel free to create more complex validation logic in pursuit of the stretch goals below.)

We already have a `validateForm` method defined in our component logic. In order to perform real validation, we must add code inside this method to access all of the input field values, analyze them, and then respond accordingly. This sounds complex, but really we are talking about some form of conditional. (Again, feel free to explore different approaches here, but what follows is one way of accomplishing this goal.)

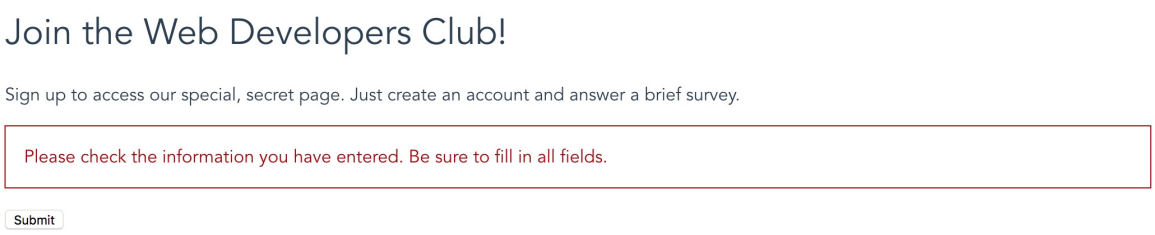
```
methods: {
    validateForm: function () {
      if ((this.username != '') &&
          (this.email != '') &&
          (this.password === this.passwordVerify)){
        // Form data is valid, so turn off the form to show the success message.
        this.showForm = false;
      } else {
        // Form data is NOT valid, so show the error message.
        this.showError = true;
      }
    }
```

```
}
```

The validation rules we must enforce are:

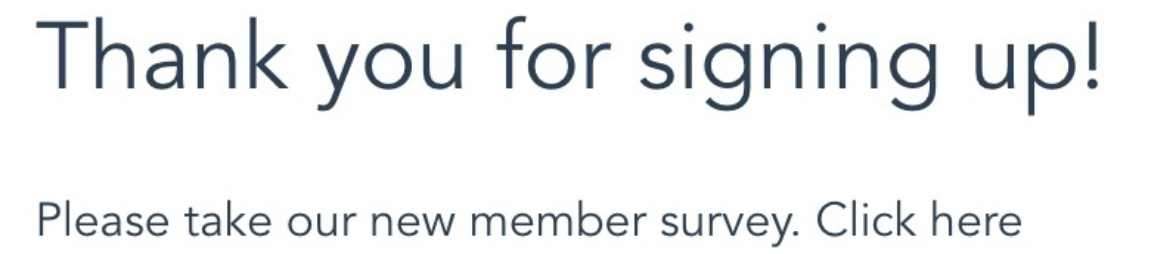
- `username` is not blank
- `email` is not blank
- `password` and `passwordVerify` are equal

The example above uses a simple series of conditional statements to check that all three conditions are met. If there is a problem with any of these fields, the validation method will cause the error message to show by setting `showError` to `true`. Here is what that looks like in the browser:



Error message

If all of the conditions pass, then the form is hidden by setting `showForm` to `false`. This also has the effect of showing the success message, which looks like this:



Success message

Currently, the success message contains a `TODO` asking us to add a `<router-link>` tag to link to another page in our application. We will continue this project in the next seciton and add that tag. For now, though, we have finished this project. We should be able to fill in the form with valid data and see the success message. When we submit invalid data (omit a field, or submit mismatched passwords), then we will see the error message.

Wrapping it Up

Once we have made all of our changes to the `src/views/Home.vue` file, it should look something like this:

```
<template>
  <div class="home">
    <div class="form-container" v-show="showForm">
      <h1>Join the Web Developers Club!</h1>
      <p>Sign up to access our special, secret page. Just create an account and answer a brief survey.</p>
      <p class="error" v-show="showError">Please check the information you have entered. Be sure to fill in all
 fields.</p>
      <form v-on:submit.prevent="validateForm">
        <p><label for="username">Username <input type="text" id="username" v-model="username"></label></p>
        <p><label for="email">Email <input type="email" id="email" v-model="email"></label></p>
        <p><label for="password">Password <input type="password" id="password" v-model="password"></label></p>
```

```
        <p><label for="passwordVerify">Verify Password <input type="password" id="passwordVerify" v-model="pass
wordVerify"></label></p>

        <p><input type="submit" value="Submit"></p>
      </form>
    </div>
    <div class="success-message" v-show="!showForm">
      <h1>Thank you for signing up!</h1>
      <p>Please take our new member survey. Click here</p><!-- TODO: Link "Click here" to the survey page. -->
    </div>
  </div>
</template>

<script>
export default {
  name: 'Home',
  data () {
    return {
      username: '',
      email: '',
      password: '',
      passwordVerify: '',
      showForm: true,
      showError: false
    }
  },
  methods: {
    validateForm: function () {
      if ((this.username != '') &&
          (this.email != '') &&
          (this.password === this.passwordVerify)){
        // Form data is valid, so turn off the form to show the success message.
        this.showForm = false;
      } else {
        // Form data is NOT valid, so show the error message.
        this.showError = true;
      }
    }
  }
}
</script>

<style scoped>
h1, h2 {
  font-weight: normal;
}
.error {
  border: 1px solid #aa0000;
  padding: 1rem;
  color: #aa0000;
}

ul {
  list-style-type: none;
  padding: 0;
}

li {
  display: inline-block;
  margin: 0 10px;
}

a {
  color: #42b983;
}
</style>
```

Build and Deploy

Once we've finished our work, we can build and deploy the project. This project has been configured to build to the `docs/` directory, so we can follow the same pattern we used before:

1. Execute the `npm run build` command to build the files into the `docs/` directory.
2. Commit all of our code.
3. Push the code up to GitHub.
4. Go into the repository settings and set the GH Pages section to publish from the `docs/` directory.

The project should now be up and available to the public through GH Pages.

Stretch Goals

There are many more fun things we can do with this project. The `README.md` file lists several possible stretch goals:

- Enhance the sign up form to collect additional info, different info, or to use different input types to collect the data
- Enhance the validation to be more specific (e.g. verify no numbers in the name, or there is an `@` and a `.` in the email address, etc.)
- Enhance error messages to be more specific (e.g. build a message that mentions which field is causing the problem)

There are many other ways we could push this forward. Feel free to explore and experiment with forms and methods to handle user input. Keep pushing, and have fun!

Routing and URLs

Within any website or web-based application, we use URLs to indicate different "locations" in our sites. Of course, these aren't really "locations" so much as they are "states" or "views" into our data. Still, the metaphor of the web is that things are "located" somewhere and we "go" there. This is the dominant metaphor we use when setting up URIs for our single page applications.

There are many situations where we want our users to share content, or we wish to direct a user to a specific feature of our sites. Users still bookmark pages so they can return to specific content or functionality. To make all of this possible, we must have some URLs that indicate different parts of our web-based software. When we go to `/profile/` we expect to see our Profile page. When we go to `/posts/` we expect to see some posts.

Throughout this section we will learn to use the `vue-router` module to add URLs to our Vue.js applications and navigate our user through our entire website. Users will be able to bookmark specific parts of our site and share links to specific content.

Core Concept: Routing and URLs

The concept of routing to control what the user sees in our web applications is common and pervasive. Whether we're using a backend framework like Django or Rails, or a frontend framework like Angular or Vue.js, we are probably implementing some form of routing. This notion of creating "routes" is a physical metaphor for moving users through our applications. We want to provide pathways to locations in our applications. Those locations are denoted by URLs. Given the nature of the Web, URLs are beneficial to us in many ways.

Working with routing in an application can be simple or complex. In this section we will talk about routing in more general terms, then we will look at how to implement routes in our applications in the following sections. Later in the book we will explore more complex approaches to routing afforded to us by virtue of using Vue.js, but for now we will focus on the fundamentals.

What URLs Do

The URL is one of the fundamental building blocks of the web. It is a "universal resource locator" that allows us to reference the network location of any file, directory, or server. We can bookmark URLs, share them with people via email or messaging, and rely on the consistency of the information we find at a URL. URLs often explain to us what they are all about:

```
http://www.imdb.com/title/tt4846340/
```

From this URL we can tell a lot about the site: It's not using any encryption when sending data. It is located on the `www.imdb.com` server. We can look at the path and see that this is the URL for a specific `title` with the ID `tt4846340`. This is a common way to form URLs with a path that includes the content type and a content item ID.

If we look at URLs from our favorite websites, we will probably see that they use similar labeling methods. The idea of using an ID or some other representative label (often called a "slug" in web development terminology) is extremely common. A URL like this tells a savvy user what they can expect to see on that page.

URLs provide a way for us to identify content online. They provide a way for us to save that location (because they are simple text that can be saved in myriad ways). They provide us a way to share content. And they give us a better understanding of what we are looking at on any given website.

Routing Views

When we talk about routing the user, we will also talk about the "view" that each route leads to. This is akin to a "page" in traditional websites, but since we are working with a single page application framework we do not use traditional pageviews. Instead, our views are more representative of the "state" of the application: Where is the user right now? What are they doing here?

Defining routes and the views they associate with looks different in different technologies, but they typically include a few key details:

- The URL pattern that the application is trying to match
- The view that should be rendered when the URL is matched
- Additional information needed to render or use the view (this is probably optional and used in specific cases)

In a backend technology like Django, the routing setup (which they call `urlpatterns`) looks like this:

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^/$', views.home),
    url(r'^login/$', views.login),
    url(r'^register/$', views.register),
    url(r'^user/(?P<username>\w+)/$', views.user_profile),
]
```

In a Vue.js application, a routing setup looks like this:

```
const router = new VueRouter({
  routes: [
    { path: '/', component: Home },
    { path: '/login', component: Login },
    { path: '/register', component: Register },
    { path: '/user/:username', component: User }
  ]
})
```

We can compare these two route definitions and see that they essentially define the same navigational structure: There is a home view, a login view, an a registration view. Those URLs are defined so that they are not looking for any additional information. Then there is a user profile view that requires a route parameter: the user's username. The username is required so the profile view can make use of the logged-in user's profile data. It is common to use route parameters to allow a view to fetch the proper data. We see these when reading news articles, viewing videos, or interacting with almost any piece of content online.

This kind of navigational structure is mirrored on millions of sites online. Each site, depending on what technology powers it, probably uses a similar system for routing requests. In a large site, the route configuration can become very large, so there are sometimes ways of breaking up route definitions or organizing them to be more efficient.

URLs in Modern JavaScript Applications

In modern JavaScript applications, there is a bit of trickiness around managing URLs. The concept of the URL was designed to point to a specific piece of content online. But in most dynamic web applications (including single page applications written in JS), the URLs do not map directly to content on the server. Instead, the URL is actually telling the web application how to process the request.

In the examples above, the `/login` URL does not correspond to a directory named `login` on the server. The `/login` URL corresponds to a route definition that says, "When this route is hit, render the login view." The web application then processes that view, renders HTML custom for that user, and then shows that HTML to the user in the browser. Even in a Python or Ruby server-side web application, the URLs do not map to files on the server.

Hash Mode

To make this work for single page JS applications, we use two major techniques. The first technique, which is the most common and requires no significant server support, is the "hash" approach. This writes all URLs using a hash mark:

```
http://example.com/#/login
```

This works in all browsers because the `#` symbol is seen as an internal page reference. This is the same syntax we would traditionally use to reference a specific element in the page by ID:

```
<a href="#map">Jump to the Map</a>

... lots of HTML content ...

<h2 id="map">Map</h2>
... a map ...
```

In the example above, we see how the hash is traditionally used. To link directly to the map on this page, we could use the link: `http://example.com/#map`.

When working with a single page JS application, we abandon this technique for internal page links. Instead, we use the hash to denote a location in the application itself. This is a tradeoff we're willing to make, and we can still scroll users to internal page content using other approaches.

History Mode

The alternative approach is to utilize the HTML5 History API, which allows us to create "normal" links:

```
http://example.com/login
```

This approach works fine in all modern browsers, and it produces a somewhat "cleaner" look to the URL (no hash mark or extra punctuation). In order to use this approach in a single page application, we must be deploying on a server that will ignore the paths after the domain and send all traffic to the same `index.html` file. We can configure any server software to allow for this, so it's a matter of setting up Apache or nginx or whatever other server we're using to, itself, properly route requests. Here is an example Apache configuration from the Vue-Router documentation:

```
<IfModule mod_rewrite.c>
  RewriteEngine On
  RewriteBase /
  RewriteRule ^index\.html$ - [L]
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteCond %{REQUEST_FILENAME} !-d
  RewriteRule . /index.html [L]
</IfModule>
```

Although we won't dive into the details of configuring the server to support history mode in this book, we can appreciate that in the example above the Apache configuration has been set up so that all requests will be sent to `/index.html`. In the case of a single page application, this is the correct way to handle the requests.

It's worthwhile to mention that this is also basically what server-side applications do when they are serving dynamic HTML content. The server is configured to hand off all user requests to the application itself, which then constructs the proper response based on analysis of the URL. In this example taken from the Django Project website, we can see that the same basic idea is at play:

```
WSGIScriptAlias / /path/to/mysite.com/mysite/wsgi.py
WSGIPythonHome /path/to/venv
WSGIPythonPath /path/to/mysite.com

<Directory /path/to/mysite.com/mysite>
<Files wsgi.py>
Require all granted
</Files>
</Directory>
```

In the case of a Django app, the WSGI standard is used. Each Django app contains a `wsgi.py` file that handles the requests that come in. When deploying a Django application, we configure Apache to hand off all requests to the `wsgi.py` file.

Again, these examples are intended to illustrate the similarities of how we handle these problems across different technologies. Although the specifics vary, the core concepts are largely the same. In the next sections we will look more closely at how to set up routing in our Vue.js applications.

Setting Up `vue-router`

Vue Router (`vue-router`) is the tool that we will use to route our users within our applications. This is a core piece of the Vue.js framework, and there is a great documentation site available to learn about it. We will cover the fundamental details in this section, but feel free to review the additional documentation.

Getting `vue-router` into our applications is easy when using the Vue CLI because it can be included as part of the project skeleton. If we wanted to add `vue-router` to an existing project, we could follow the guide provided in the Vue Router documentation.

Bootstrapping a Project with `vue-router`

When we bootstrap a project, we can choose to use `vue-router` . This results in a setup that includes a routes definition and the integration of the routes in several places in the project.

First, in the `src/main.js` we can see that the router is included in the imports and setup of the app itself:

```
import Vue from 'vue'
import App from './App'
import router from './router'

Vue.config.productionTip = false

new Vue({
  el: '#app',
  router,
  template: '<App/>',
  components: { App }
})
```

If we look at the file in `src/router.js` we can see the route definitions:

```
import Vue from 'vue'
import Router from 'vue-router'
import FormsPractice from '@/components/FormsPractice'

Vue.use(Router)

export default new Router({
  routes: [
    {
      path: '',
      name: 'forms',
      component: FormsPractice
    }
  ]
})
```

In the `App` component, found in `src/App.vue` , we can see that the template now includes a `<router-view>` element:

```
<template>
  <div id="app">
    <h1>Practicing with Forms</h1>
    <router-view/>
  </div>
</template>
```

The `<router-view>` element is the location in the App template where the router's view will be injected into the application. It replaces the use of the component tag in the template like we saw used in the previous Vue.js projects in this book. This will inject whatever component we have defined in the routes into the HTML at that location. Of course, the next question is: How do we define routes?

Defining Routes

We define routes in the `routes.js` file as part of the `routes` Array. The convention is to use lowercase in paths and route names, and Uppercase the first letter of the component name. Path This definition looks something like this:

```
import Vue from 'vue'
import Router from 'vue-router'
import Home from '@/components/Home'
import Login from '@/components/Login'
import Register from '@/components/Register'

Vue.use(Router)

export default new Router({
  routes: [
    {
      path: '/',
      name: 'home',
      component: Home
    },
    {
      path: '/login',
      name: 'login',
      component: Login
    },
    {
      path: '/register',
      name: 'register',
      component: Register
    }

  ]
})
```

In the example configuration above, we see three different route objects defined: `Home`, `Login`, and `Register`. These three route objects have the same set of properties: `path`, `name`, and `component`. These properties are commonly used, and often the routing for our apps need not get any more complex. Let's take a look at what these properties mean.

The `path` property is the desired URL for the route. These URLs should begin with a forward slash (`/`), and they can include any URL safe characters. This the URL the user will see in their browser's location bar, and it should make sense based on the content of the view.

The `name` property is a unique name for this route in the Vue.js application. The name can be used to reference the route in templates, so instead of linking to `/some-view-location` we can link to `ViewName`. The advantage here is that we can alter the URLs used in the app by modifying the `routes.js` file without modifying every template where we link to those routes. This is a very helpful feature of `vue-router` and is a common feature among URL routing systems in various frameworks. (We will look more at building navigational linkage in our templates later.)

Finally, the `component` property indicates which component should be injected into the `<router-view>` element in the application template. This component might utilize several other components as part of its template, and there are even ways to get more complex with loading multiple components in each route, but for most cases this feature works perfectly as it is.

In addition to these fundamentals of route definition, there are a few other more advanced features we often need to use in our route definition.

Dynamic URLs

Sometimes it is necessary to create a URL using data known in the system. A common use case for this is showing a user profile at a URL that looks like this:

```
http://example.com/user/username
```

When linking to user profiles in a system, it's necessary to read that `username` value out of the URL and process the user profile view accordingly. We can define a route that will give us access to the `username` value like this:

```
const router = new VueRouter({
  routes: [
    {
      path: '/user/:username',
      name: 'userProfile',
      component: User
    }
  ]
})
```

In this case, the `:username` portion of the URL defines a "route parameter", which is what we call these variables that are inserted into the URL of a route definition. We use route parameters to modulate the content of the view we show to our users. Route parameters can be referenced in our component logic like this:

```
<template>
  <div class="component">
    <h2>{{ welcome }}</h2>
    <p>{{ $route.params.username }}</p>
  </div>
</template>

<script>
  export default {
    data () {
      return {
        salutation: 'Ahoy'
      }
    },
    computed: {
      welcome: function () {
        return this.salutation + ', ' + this.$route.params.username
      }
    }
  }
</script>
```

In this example we can see that we are using the `$route.params` object in both the component template and logic. In the template, we can directly reference `$route.params` to use in creating the interface. In the component logic, we can refer to the route parameters in the same way we refer to other properties defined as part of the component's `data` object: `this.$route.params`. Any value we assign in the URL of the route definition will be revealed as a property in `this.$route.params`.

In the example above, we can see that there is a computed value called `welcome` that is being generated. This `welcome` value uses `this.salutation` and `this.$route.params.username` to create a welcome message that is interpolated into our template. If the URL is changed, the page will properly reflect the change in the

`this.$route.params.username` value.

Note: If we are using more complex components (as we do later in this book) that define a `created` function or do other initialization work in the component, we may need to re-initialize the component when the `$route.params` object is changed. More information about making that happen can be found on the Dynamic Matching documentation as part of the Vue Router docs.

We could also add multiple values to a route definition. Here is a helpful table showing how route parameters get used in various URLs:

Route Definition	Actual URL	Route Parameters
`/:username`	`/shawnr`	`{ 'username': 'shawnr' }`
`/:username`	`/jdoe`	`{ 'username': 'jdoe' }`
`/:username/posts/:postid`	`/shawnr/posts/12345`	`{ 'username': 'shawnr', 'postid': 12345 }`
`/:username/posts/:postid`	`/jdoe/posts/9876`	`{ 'username': 'jdoe', 'postid': 9876 }`

Route parameters are powerful tools for coordinating information between different views in our applications, and we will explore many use cases for them throughout this book.

Nested URLs

Sometimes we wish to define URLs as part of a "section" of our site. For example, in many sites we would have some kind of user profile page located at `/username`. However, we might also have the following URLs:

- `/user/username/profile` — User profile page
- `/user/username/profile/edit` — Edit user profile page
- `/user/username/settings` — Site settings page for a specific user
- `/user/username/posts` — Listing of all posts the user has made
- `/user/username/post/12345` — View for a single post made by the user

We see these kinds of related URLs all the time online, and they make sense: We like to group pages with similar functionality under similar URLs. It helps us as developers, and our users, understand our websites and applications better.

The route definition that would create some of these URLs would look like this:

```
{
  path: '/user/:username',
  component: User,
  children: [
    {
      // UserProfile will be rendered inside User's <router-view>
      // when /user/:username/profile is matched
      path: 'profile',
      name: 'userProfile',
      component: UserProfile
    },
    {
      // UserPosts will be rendered inside User's <router-view>
      // when /user/:username/posts is matched
      path: 'posts',
      name: 'userPosts',
      component: UserPosts
    }
  ]
}
```

In this case, we have a general `/user/:username` URL that loads the `User` component. The `User` component will inject an appropriate *nested* route according to the `path` values we have defined. We define nested routes using the `children` property on a route definition. This property accepts an array of additional route definitions.

The `<router-view>` element in the `User` component template looks like this:

```
<template>
  <div class="user">
    <h2>{{ welcome }}</h2>
    <router-view></router-view>
  </div>
</template>
```

The `UserProfile` and `UserPosts` components will be rendered inside the `<router-view>` tag within the `User` template. Each of these components can access the `$route.params.username` value in their templates or component logic. Note that the URLs of nested routes defined in the `children` property do not use absolute paths—they do *not* begin with a forward slash ('/'). Rather, they are understood to continue from the `path` definition on the parent route.

Moving On

Now that we've got the basics of how routes work, let's move on and discuss creating new components as well as linking our URLs together using `<router-link>` tags. There are more features in `vue-router` that we have not covered here. Some of those will be touched on later in this book, but it's always worthwhile to visit the official Vue Router documentation to learn more.

Creating New Locations in the Application

Creating new locations in a Vue.js app is as simple as adding a new route definition. However, we also will usually want to create a new component to handle the unique content that should be displayed at the new location. This is not terribly difficult to accomplish, but it can be made easier if we have some handy reference material.

In this section we will look at the basic steps involved in creating a new component and route within our Vue.js application. (There are plenty of reasons to create new components that don't necessarily map to a specific route in the application. The first part of these directions should generally work for that, too.)

Defining a New Component

The first step in creating a new component is to create a new `.vue` component in the either the `src/components/` directory or the `src/views` directory. If your component will be used as a router component then it should go in the `views` directory.

All of our components follow the same pattern, so we can use a little snippet of code as a way to bootstrap the new components.

Here is a basic Vue component skeleton. There is no logic here, and it is meant to be edited to perform the specific function we need. But this provides us with a basic template to fill up with our custom stuff.

```
<template>
  <div class="component">
    <h2>{{ message }}</h2>
  </div>
</template>

<script>
  export default {
    data () {
      return {
        message: 'This component works.'
      }
    }
  }
</script>

<style scoped>
  .component {
    background: #e8e8e8;
    min-height: 400px;
  }
</style>
```

To get a new component going, we can simply copy the blank template above into a file and save it into the `src/components/` or `source/views` directory in our project. It is important to maintain our naming pattern and give the file a `.vue` extension so it can be properly understood by our build system and framework.

Once we have bootstrapped the basic component (even in this primitive form), we can define the route that will serve our new component.

> **Code Snippets**

> Many code editors allow developers to create "snippets" of code that can be re-used in different places. Developers love to create and save small chunks of code that can help them get started quicker on building a new component, or to help them stick with a proven pattern or method. Saving snippets of code can be a huge timesaver when writing complex software, especially when we work with the same frameworks and libraries over and over again.
>
> Most code editors will support a "snippets" feature. It's worthwhile to explore the features of our chosen text editors and see how they handle snippets of code. If the editor doesn't support storing random small code snippets, we can build up a set of files in a snippets directory and manage our boilerplate code that way. However we choose to manage this information, storing and accessing these quick snippets is a great way to work.

Setting up the New Route

To set up a new route that uses the component we just created, we can add a new route definition object to our routes Array. It's important to note that we must import the component at the top of the `routes.js` file so that the Vue.js framework can properly identify, understand, and link the modules we've defined. Here is what that would look like in our route definitions file:

```
import Vue from 'vue'
import Router from 'vue-router'
import Home from '@/views/Home'
import NewComponent from '@/views/NewComponent'

Vue.use(Router)

export default new Router({
  routes: [
    {
      path: '/',
      name: 'home',
      component: Home
    },
    {
      path: '/newcomponent',
      name: 'newComponent',
      component: NewComponent
    }
  ]
})
```

Looking at this example, we can see that there is a set of import statements at the top of the `routes.js` file. These list every module that is being used within this file: `Vue`, `Router`, `Home`, and `NewComponent`. Each import statement names the module it is importing and then indicates where the module will be found using the `from` clause. In the case of `Vue` and `Router`, these modules are made available for us by virtue of our dependency management system (stewarded by NPM and Webpack). (**Note:** We will explore more about dependencies in future sections.)

If you look in your `package.json` file you should see that the `vue-router` is set up as a dependency to be installed when you run `npm install`.

If it's not installed, you can install is as an application dependency with the following command.

```
npm install vue-router --save
```

In the case of our custom code, we must actually indicate what file contains the module we wish to import. This file path uses a couple of shorthand references to reduce the characters we need to type. The path is a string, so it needs to be in quotes. Each path begins with the `@` symbol, which is a shortcut unique to the way a Vue.js project is created when using the Vue CLI. The `@` shortcut refers to the `/src/` directory, and it is defined in the Webpack configuration. It is the same as writing `/src/` at the beginning of the path. Finally, we have the path and filename. Filenames do not need to include the `.vue` extension because it is inferred.

So the import for `import NewComponent from '@/views/NewComponent'` refers to the file found at `/src/views/NewComponent.vue` and will import the module from that file with the name `NewComponent`. We use the `NewComponent` name when referencing the component in the route definition.

Once we have this route definition set up, we could visit the path in our development browser and we should see the new content displayed. The only thing left to do is to create links within our application to bring users to this new location.

Building Navigation

Building navigation in a Vue.js application is made easy by the tools `vue-router` gives us. We can use the custom `<router-link>` tag to insert links in templates by name or by path. We can programmatically move users to different parts of the application within the component logic. And `vue-router` even handles properly denoting when a particular link is "active" based on the current path. Let's dive in and learn how to use all these features to our advantage.

Router Links

We create links to other routes by using the `<router-link>` tag in our templates. This tag is automatically converted into an `<a>` tag when it is processed by Vue.js, but it has the advantage of coming with extra functionality to make all of our route links work flawlessly. Here is an example of how we could create a navigation in our templates using `<router-link>`:

```
<ul class="nav">
    <li>
        <router-link v-bind:to="{name: 'home'}">Home</router-link>
    </li>
    <li>
        <router-link v-bind:to="{name: 'newComponent'}">New Component</router-link>
    </li>
</ul>
```

When we use the `<router-link>` element to create links, we also use a `to` attribute instead of a typical `href` attribute. The `to` attribute accepts *either* the `name` or the `path` property of the route. In the example above we have used the `name` property. But we could achieve the exact same result (with slightly less typing) using the `path` property:

```
<ul class="nav">
    <li>
        <router-link to="/">Home</router-link>
    </li>
    <li>
        <router-link to="/newcomponent">New Component</router-link>
    </li>
</ul>
```

Use Named Routes

To many new web developers, the idea of using named routes seems a little strange. After all, why add another thing to remember when we could just refer to the route by the path, which is required by Vue Router? This is a reasonable question, but the value of using named routes outweighs the difficulty of defining and remembering the name.

Named URLs pay off in a few ways: First, they allow us to alter the paths in our application without editing our templates. We can alter the path in the route definition, and we do not need to touch any other files if we have only used named routes in our links. Second, paths can become long and difficult to remember. It can be especially difficult to remember the order of route parameters,

and on a large project this only becomes more difficult. Third, named routes can be named in a sensible way so it's easy to remember which route handle what, and that can allow us to use abbreviated paths when suitable (such as when building a complex document retrieval system).

Passing Data as Route Parameters

When creating links, we often need to supply some information for the route parameters. Imagine that we are building the user pages we defined in the previous section:

```
routes: [
    {
      path: '/user/:username',
      name: 'userProfile',
      component: User
    }
]
```

In order to link to the `UserProfile` page, we need to provide a `username` parameter. We would write this link in the template like this:

```
<router-link v-bind:to="{ name: 'userProfile', params: {username: 'jdoe'} }">Profile</router-link>
```

We must use the `v-bind` directive to supply a JavaScript object to the `to` attribute. The JS object we create has a property called `name`. We could alternatively use the `path` property if we wished to define this link with a `path` instead of a `name`. We also supply a `params` property that contains an object with all the parameters this route requires. In this case, that is just a `username` value, which we have set to `'jdoe'`. The result of this tag in our template would be a link that points to `/user/jdoe`.

It's more likely that rather than providing the username as a static string, we would want to use some value known to the component to create this link. If we imagine that we have a `user` object defined in our component data, the `<router-link>` tag would look more like this:

```
<router-link v-bind:to="{ name: 'userProfile', params: {username: user.username} }">Profile</router-link>
```

All that we've changed here is to refer to `user.username` instead of `'jdoe'`. This is a common way to use dynamic route parameters with the `<router-link>` tag.

Using `router-link-active`

One common design pattern for frontend code is to label the link to the current page with an `active` class. This allows us to write styles for our navigation system that indicate which page the user is currently viewing. Many sites use convoluted systems to coordinate the current path in the location of the browser to the navigation links on the pages. Luckily, `vue-router` takes care of this for us when we use the `<router-link>` tag to define our internal site links.

When we view a page in our Vue.js app, we can look at the links we've defined with `<router-link>` and see that the currently active link has a `router-link-active` class appended to it. If we style that class in our `App` component code, then we can see our currently active links indicated as we navigate around our Vue.js website. This is a handy feature and saves us yet another small development headache.

Programmatic Navigation

Although we mostly will move users around with links that the user clicks, sometimes it is necessary to move users from one location to another programmatically. For example, when a user fills out a complex form, we might verify we have all the correct data before moving them off to the confirmation page. Or we might build a "wizard" style interface that walks users through the different steps of a process. In that case, it might be useful to push users to the next step automatically. To accomplish this, we use the `router.push()` method supplied by `vue-router`.

Just like with the `<router-link>` tag, we can use `router.push()` with either a simple route `name` or `path`, or we can use it with a more complex object that defines a route `name` or `path` and a `params` data object. Here is the simple way to use `router.push()`:

```
<script>
  export default {
    data () {
      return {
        validated: false
      }
    },
    methods: {
      validate: function () {
        if (this.validated) {
            router.push('confirm');
        }
      }
    }
  }
</script>
```

In this example, we can imagine that we have some form being presented to the user. When it is filled in with data the `validate` method would be executed. If `validated` is `true`, then the user would be moved to the `confirm` route. Programatic routing allows us to execute some logic before moving the user to a new view.

```
<script>
  export default {
    data () {
      return {
        validated: false,
        orderID: 123
      }
    },
    methods: {
      validate: function () {
        if (this.validated) {
            router.push({ name: 'confirm', params: {orderID: this.orderID} });
        }
      }
    }
  }
</script>
```

In this second example, the same scenario applies, but this time we want to supply the `orderID` value to the `confirm` route. To do so, we construct an object very much like we did with the `<router-link>` tag, and we define our route parameters in a `params` object. Now, when the form is validated all of this data will be properly assembled into a route reference.

All of these features combine to allow us to build a huge variety of navigational elements in our sites and applications. We can use whichever approach works to fulfill our requirements, and we can integrate this navigation at every level of our application.

Quiz: Routing and URLs

Please enjoy this self-check quiz to help you identify key concepts, points, and techniques discussed in this section.

Visit Quiz Online

The quiz on this page has been removed from your PDF or ebook format. You may take the quiz by visiting this book online.

Project: Responding to User Input Part Two

In the previous section, we worked on the first part of the Multi-View Application project. For this final part of the project, we should continue working on the repository we cloned in the last section. We will work a bit more with forms and explore how to set up simple routes in our application to provide bookmarkable locations for our users.

It's assumed that we've already forked, cloned, and set up our workspace. We should be able to continue from where we left off last time.

Review the Requirements

In this second part of the project, we will set up another form and also some new routes in our application. The new routes will allow us to experiment with using `vue-router` to move the user to different locations within our site. The goal is to allow the user to click to the new member survey page, and then to move the user to the secret page once they have completed the survey successfully.

To do this, we will need to edit four files: Three components and the router definitions. There are `TODO` notes in each of the files to help us find what we need to edit, and we should be able to use the examples that came previously in this section to complete the work.

Here are the Basic Requirements outlined in the project `README`:

In the `src/views/Survey.vue` file

- Complete the survey form by filling in the TODO notes
- Use `v-for` loops in the template to create options for the checkbox groups
- Create a validation method to handle the rules outlined in the component comments
- Use a `$router.push` statement to move the user to the Secret page

Create the `src/views/Secret.vue` file

- Create a basic component from scratch called `Secret`
- The content of the `Secret` page should be something you come up with: A favorite tip about web development, a funny joke, a humorous image, etc.
- Provide links back to the other two pages using `<router-link>` tags in the template

In the `src/router.js` file

- Import the Survey component properly
- Import the Secret component properly
- Add the Survey component at the `/survey` path
- Add the Secret component at the `/secret` path

Please note: We will not be tackling these requirements exactly in order, but we will complete each of these.

Working the Project

As we begin work we will need to have the following files open:

- `src/views/Home.vue`
- `src/views/Survey.vue`
- `src/views/Secret.vue`
- `src/router.js`

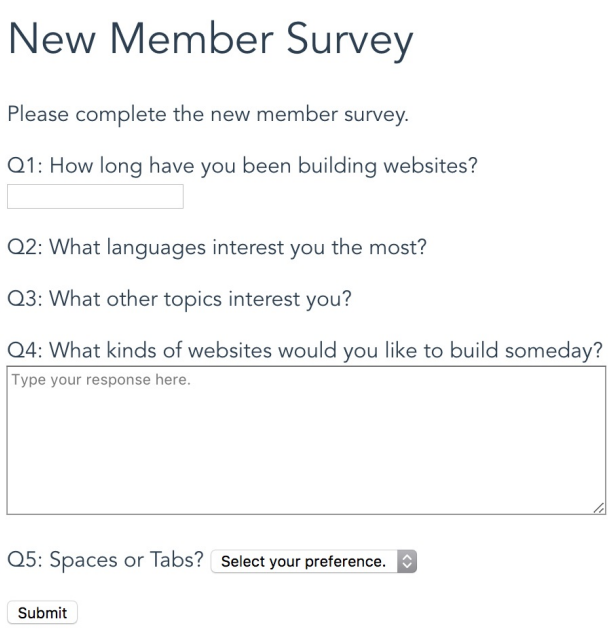
We will move between these files to complete all of the work for this project.

Adding the Survey View to the Routes Array

First of all, we should add the Survey view to the routes definition so we can preview our work as we complete the form and logic in that component. This is done by editing the `src/router/index.js` file. We find `TODO` notes in this file helping us know where we need to edit. First, we must import the `Survey` component, and then we must set up a route definition. Here is what that looks like:

```
import Vue from 'vue'
import Router from 'vue-router'
import Home from '@/views/Home'
import Survey from '@/views/Survey'
// TODO: Import the Secret component

Vue.use(Router)

export default new Router({
  routes: [
    {
      path: '/',
      name: 'home',
      component: Home
    },
    {
        path: '/survey',
        name: 'survey',
        component: Survey
    }
    // Add the Secret route definition
  ]
})
```

As we can see, there are still some `TODO` notes left in here, but we will handle those later. For now, we have added the `import` statement for the `Survey` component and the route definition. The route definition specifies the URL for the view, so we can load up that URL to preview our site:



Starting Survey View

The image above shows what the starting Survey view looks like. It can be accessed at `http://localhost:8080/#/survey`. Right now we must manually type in that URL, but we will eventually add a link in the success message on the `Home` component. Notice that although we see some form fields here, we do not actually have choices for the questions. Let's finish out the form in the `Survey` component so we can see the completed page.

Creating the Form in `Survey.vue`

This form is partially there for us. We have to add the `v-on` directive to the `<form>` tag so we can properly handle the `submit` event. We will also need to use the `.prevent` modifier, just like we did with the sign up form on the `Home` component.

We also need to set up an error message the same way we did on the `Home` component. We can see that the component `data` object already contains a `showError` value, so we can use the same method to make this error message appear and hide:

```
<p class="error" v-show="showError">Please check the information you have entered. Be sure to fill in all field
s.</p>
```

The next `TODO` asks us to set the model on the `q1` input. This is a pretty simple addition:

```
<input type="text" id="q1" v-model="q1">
```

We will use the same kind of `v-model` directive just like we did with the `Home` component. The next question present us with a somewhat different challenge: We must create a loop on the `<input>` element and use it to populate the form with a set of checkboxes based on the information stored in the `languageOptions` array:

```
<p>Q2: What languages interest you the most?<br>
  <label v-for="language in languageOptions">
    <input type="checkbox" v-model="q2" v-bind:value="language.value">
    {{ language.text }}
  </label>
</p>
```

In this example we see that we have used `v-for` to set up the loop on the `<label>` tag. This will duplicate the label and all the children elements it contains for each item in the `languageOptions` Array, which is defined as part of the component's `data` object. The `languageOptions` Array contains objects that have two properties: `value` and `text`. These two properties are used in the appropriate places to output the right information into the template.

Since all of these checkboxes are meant to be part of the same question (and we want to record which checkboxes are checked in the `q2` Array), we give each input field the same `v-model` name. However, we want to alter the `value` attribute according to the information in the Array. This is accomplished using the `v-bind` directive on the `value` attribute. We finish out by simply printing the `text` property for use as the label on our checkbox.

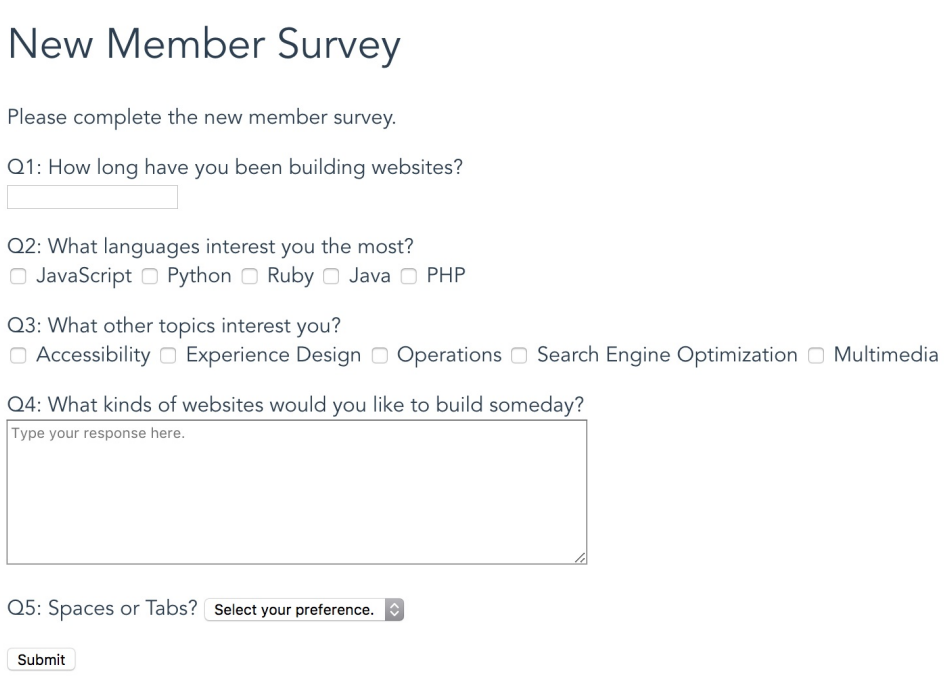
Question 3 uses the same checkbox group setup that Question 2 used, so we will essentially create the same template structure, using the `topicOptions` Array instead of the `languageOptions` Array:

```
<p>Q3: What other topics interest you?<br>
  <label v-for="topic in topicOptions">
    <input type="checkbox" v-model="q3" v-bind:value="topic.value">
    {{ topic.text }}
  </label>
</p>
```

We can continue through the `TODO` notes adding the `v-model` directives to their respective input fields until the form is complete. Once we're done, we should have a form that looks like this:

```
<form v-on:submit.prevent="validateForm">
  <p class="error" v-show="showError">Please check the information you have entered. Be sure to fill in all fie
lds.</p>
  <p><label for="q1">Q1: How long have you been building websites?<br><input type="text" id="q1" v-model="q1"></
label></p>
  <p>Q2: What languages interest you the most?<br>
    <label v-for="language in languageOptions">
      <input type="checkbox" v-model="q2" v-bind:value="language.value">
      {{ language.text }}
    </label>
  </p>
  <p>Q3: What other topics interest you?<br>
    <label v-for="topic in topicOptions">
      <input type="checkbox" v-model="q3" v-bind:value="topic.value">
      {{ topic.text }}
    </label>
  </p>
  <p>
    <label for="q4">Q4: What kinds of websites would you like to build someday?<br>
      <textarea cols="70" rows="8" id="q4" placeholder="Type your response here." v-model="q4"></textarea>
    </label>
  </p>
  <p>
    <label for="q5">Q5: Spaces or Tabs?
      <select id="q5" v-model="q5">
        <option value="">Select your preference.</option>
        <option value="spaces">Spaces</option>
        <option value="tabs">Tabs</option>
      </select>
    </label>
  </p>
  <p><input type="submit" value="Submit"></p>
</form>
```

Now we can see our completed form in action:



Completed Survey Form

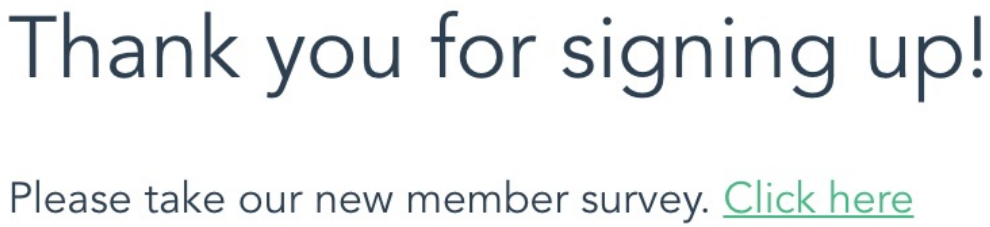
We have all of our choices available and everything should be all connected in the template. Let's take a moment to add that link on the `Home` component to get users to this view.

Adding the Link to the Survey from the Home View

Adding the link for a user to click is very easy. Since we have used name routes, we can reference the name of the route. We only need to open the `src/views/Home.vue` file and add a single `<router-link>` tag inside the success message:

```
<p>Please take our new member survey. <router-link to="/survey">Click here</router-link></p>
```

Once we have this, we can return to the home of our project (`http://localhost:8080/#/`) and fill in the sign up form. When we have successfully submitted valid data, we should see the link show up. Clicking the link should take us back to the Survey view. Here is what it looks like:



Survey Link on Home View

Creating the Secret View

The next thing we need to do is create a brand new component to run the Secret view. This is a view that should convey some helpful tip about web development (or, really, you can put anything you want on this page).

To begin, create a new file in `src/views/` called `Secret.vue` . We can use the boilerplate component code included earlier in the Routing and URLs section to start us off:

```
<template>
  <div class="component">
    <h2>{{ message }}</h2>
  </div>
</template>

<script>
  export default {
    data () {
      return {
        message: 'This component works.'
      }
    }
  }
</script>

<style scoped>
  .component {
    background: #e8e8e8;
    min-height: 400px;
  }
</style>
```

We can modify the message to say whatever we would like. We can also alter the template in whatever way we prefer. And, of course, we can change the styles to fit our needs. Once we have created the basic component we can save this file and add the Secret view to our routes definition so we can preview our work.

Adding the Secret View to the Routes Array

Adding the Secret view to the `routes` Array is pretty much the same as adding the Survey view previously. Once we are done, we should be able to visit `http://localhost:8080/#/Secret` to see our new view:

Secret View

Obviously we will each have a different secret tip, but feel free to use this one if you are stumped. Now we only need to finish the validation method for the `Survey` component, and then we will be completely finished with this project.

Finishing the Validation Method for the Survey

The last bit of work we must complete is to finish the validation method for the `Survey` component. This is mostly like the previous `validateForm` method we wrote in the `Home` component, except the rules are a little different.

This time out, we want to verify the following:

- `q1` is not blank
- `q2` is an Array with the length greater than zero
- `q3` is an Array with the length greater than zero
- `q4` is not blank
- `q5` is not blank

We can, once again, validate the data using a conditional with a set of conditional statements:

```
methods: {
  validateForm: function () {
    if ((this.q1 != '') &&
        (this.q2.length > 0) &&
        (this.q3.length > 0) &&
        (this.q4 != '') &&
        (this.q5 != '')) {
      // Form is valid
      this.$router.push('secret');
    } else {
      this.showError = true;
    }
  }
}
```

When the form is valid, the validation method will execute `this.$router.push('secret');`, which moves the user to the Secret view. This is the programmatic way of moving the user to different locations in the application (as opposed to just asking the user to click, like we did on the Home view).

Now that we've added this code, we can test the Survey view and see that when we fill out the form successfully it moves us to the Secret view. Huzzah! We are done!

Wrapping Up

Several files have been modified to complete this project. Here are the complete files so we can check our work against them.

`src/components/Home.vue`

Full file contents:

```
<template>
  <div class="home">
    <div class="form-container" v-show="showForm">
      <h1>Join the Web Developers Club!</h1>
      <p>Sign up to access our special, secret page. Just create an account and answer a brief survey.</p>

      <p class="error" v-show="showError">Please check the information you have entered. Be sure to fill in all
 fields.</p>

      <form v-on:submit.prevent="validateForm">

        <p><label for="username">Username <input type="text" id="username" v-model="username"></label></p>
        <p><label for="email">Email <input type="email" id="email" v-model="email"></label></p>
        <p><label for="password">Password <input type="password" id="password" v-model="password"></label></p>
        <p><label for="passwordVerify">Verify Password <input type="password" id="passwordVerify" v-model="pass
wordVerify"></label></p>

        <p><input type="submit" value="Submit"></p>
      </form>
    </div>
    <div class="success-message" v-show="!showForm">
      <h1>Thank you for signing up!</h1>
      <p>Please take our new member survey. <router-link to="/survey">Click here</router-link></p>
    </div>
  </div>
</template>

<script>
export default {
  name: 'Home',
  data () {
    return {
      username: '',
      email: '',
      password: '',
      passwordVerify: '',
      showForm: true,
      showError: false
    }
  },
  methods: {
    validateForm: function () {
      if ((this.username != '') &&
          (this.email != '') &&
          (this.password === this.passwordVerify)){
        // Form data is valid, so turn off the form to show the success message.
        this.showForm = false;
      } else {
        // Form data is NOT valid, so show the error message.
        this.showError = true;
      }
    }
  }
}
</script>

<style scoped>
h1, h2 {
  font-weight: normal;
}
```

```
.error {
  border: 1px solid #aa0000;
  padding: 1rem;
  color: #aa0000;
}

ul {
  list-style-type: none;
  padding: 0;
}

li {
  display: inline-block;
  margin: 0 10px;
}

a {
  color: #42b983;
}
</style>
```

src/components/Survey.vue

Full file contents:

```
<template>
  <div class="survey">
      <h1>New Member Survey</h1>
      <p>Please complete the new member survey.</p>
      <form v-on:submit.prevent="validateForm">
        <p class="error" v-show="showError">Please check the information you have entered. Be sure to fill in a
ll fields.</p>

        <p><label for="q1">Q1: How long have you been building websites?<br><input type="text" id="q1" v-model=
"q1"></label></p>

        <p>Q2: What languages interest you the most?<br>
          <label v-for="language in languageOptions">
            <input type="checkbox" v-model="q2" v-bind:value="language.value">
            {{ language.text }}
          </label>
        </p>

        <p>Q3: What other topics interest you?<br>
          <label v-for="topic in topicOptions">
            <input type="checkbox" v-model="q3" v-bind:value="topic.value">
            {{ topic.text }}
          </label>
        </p>
        <p>
          <label for="q4">Q4: What kinds of websites would you like to build someday?<br>
            <textarea cols="70" rows="8" id="q4" placeholder="Type your response here." v-model="q4"></textarea>

          </label>
        </p>
        <p>
          <label for="q5">Q5: Spaces or Tabs?
            <select id="q5" v-model="q5">
              <option value="">Select your preference.</option>
              <option value="spaces">Spaces</option>
              <option value="tabs">Tabs</option>
            </select>
          </label>
        </p>
        <p><input type="submit" value="Submit"></p>
      </form>
  </div>
```

```
</template>

<script>
export default {
  name: 'Survey',
  data () {
    return {
      showError: false,
      q1: '',
      q2: [],
      q3: [],
      q4: '',
      q5: '',
      languageOptions: [
        {
            text: 'JavaScript',
            value: 'js'
        },
        {
            text: 'Python',
            value: 'py'
        },
        {
            text: 'Ruby',
            value: 'ruby'
        },
        {
            text: 'Java',
            value: 'java'
        },
        {
            text: 'PHP',
            value: 'php'
        }
      ],
      topicOptions: [
        {
            text: 'Accessibility',
            value: 'axe'
        },
        {
            text: 'Experience Design',
            value: 'ux'
        },
        {
            text: 'Operations',
            value: 'ops'
        },
        {
            text: 'Search Engine Optimization',
            value: 'seo'
        },
        {
            text: 'Multimedia',
            value: 'media'
        }
      ]
    }
  },
  methods: {
    validateForm: function () {
      if ((this.q1 != '') &&
          (this.q2.length > 0) &&
          (this.q3.length > 0) &&
          (this.q4 != '') &&
          (this.q5 != '')) {
        // Form is valid
        this.$router.push('secret');
      } else {
```

```
        this.showError = true;
      }
    }
  }
}
</script>

<!-- Add "scoped" attribute to limit CSS to this component only -->
<style scoped>
.error {
  border: 1px solid #aa0000;
  padding: 1rem;
  color: #aa0000;
}
h1, h2 {
  font-weight: normal;
}

ul {
  list-style-type: none;
  padding: 0;
}


a {
  color: #42b983;
}
</style>
```

src/components/Secret.vue

Full file contents:

```
<template>
    <div>
        <h2>{{ message }}</h2>
    </div>
</template>

<script>
export default {
    data () {
        return {
            message: 'Always look in the devtools.'
        }
    }
}
</script>

<style scoped>

</style>
```

router/index.js

Full file contents:

```
import Vue from 'vue'
import Router from 'vue-router'
import Home from '@/views/Home'
import Survey from '@/views/Survey'
import Secret from '@/views/Secret'
```

```
Vue.use(Router)

export default new Router({
  routes: [
    {
      path: '/',
      name: 'home',
      component: Home
    },
    {
        path: '/survey',
        name: 'survey',
        component: Survey
    },
    {
        path: '/secret',
        name: 'secret',
        component: Secret
    }
  ]
})
```

Build and Deploy

Once we've finished our work, we can build and deploy the project. This project has been configured to build to the `docs/` directory, so we can follow the same pattern we used before:

1. Execute the `npm run build` command to build the files into the `docs/` directory.
2. Commit all of our code.
3. Push the code up to GitHub.
4. Go into the repository settings and set the GH Pages section to publish from the `docs/` directory.

The project should now be up and available to the public through GH Pages.

Stretch Goals

There are many more fun things we can do with this project. The `README.md` file lists several possible stretch goals:

- Add a navigation element to the `src/App.vue` template, and be sure to use the `router-link-active` class to style the current page link
- Add more pages of content to the site to practice creating new components from scratch and adding them to the routes Array

And it's still worthwhile to pursue the stretch goals from the previous portion of this project:

- Enhance the sign up form to collect additional info, different info, or to use different input types to collect the data
- Enhance the validation to be more specific (e.g. verify no numbers in the name, or there is an `@` and a `.` in the email address, etc.)
- Enhance error messages to be more specific (e.g. build a message that mentions which field is causing the problem)

There are many other ways we could push this forward. Feel free to explore and experiment with forms and methods to handle user input. Keep pushing, and have fun!

Using API Data

On the web today, there are many computers out there waiting for other computers to come request some information. We tend to think only of the websites that are made for human eyes, but there is a whole ecosystem of services and sites meant to be used by our applications. We call these machine-oriented endpoints "Application Programming Interfaces" (APIs), and they are often terribly difficult for new web developers to grasp. The problem is partly that most APIs are used behind the scenes, and until very recently nobody ever talked much about APIs in polite company.

Another problem with understanding APIs is that the concept of an "API" applies to many different situations. We hear developers talk about the JavaScript API, the Vue.js API, the jQuery API, or myriad other specific modules, libraries, and frameworks. Why do we use the same words to describe both the web-based services that can supply data and functionality to our apps and the special words and symbols we use to invoke specific behaviors in programming languages and libraries?

The answer is: Interface. APIs are the way we communicate with the machine. We use a very controlled vocabulary to ask a machine to do something for us, and then it performs that task according to predefined routines. This is what an API is for: Communicating to a software system. Because of this, it doesn't really matter whether we're communicating with a system that is living in our web browser or that's living on a remote server across the ocean.

A World of Services

We live in a world of web services. Our sites rely on services to handle authentication, to provide storage, to manage infrastructure, and much more. We integrate the features of other sites with ours so we can allow our users to share content, post messages, or invite friends. And even when we are building functionality strictly for ourselves, we want to leverage services to help us keep our components separate, autonomous, and easier to maintain or enhance.

As front end web developers, we become much more powerful when we can leverage third-party APIs to provide data and services we could not build entirely in the browser. We can retrieve almost any information and use almost any kind of function to make our sites better for our users if we only implement the proper API integrations. This is not necessarily an easy task, but it is possible.

And sometimes it's easy, too.

Core Concept: Using Third-Party Data APIs

The topic of Third-Party Data APIs is really too broad to be covered thoroughly here. It is a topic full of nuance and importance, especially for developers creating Data APIs. When designing an API, it is as complex as designing a framework or code library: You are trying not only to solve a problem, but to solve it in a way that is understandable and accessible to everyone.

Luckily, we do not need to design an API for our purposes here. We need only to "consume" an API: That is, we will make requests to an API service and we will receive responses from that service that contain data we can use in our Javascript application. This process requires us to get to know the shape of some data returned from a chosen service, but it does not require us to understand the finer points of how APIs are put together.

Just so we aren't diving in completely blind, we will review some of the basics of how APIs work so that we can more easily understand the directions we see in our chosen API's documentation.

Websites for machines

If you compare human beings and computers, you will notice many differences. In fact, there is not much in common between what a human hopes to get from the world and what a computer hopes to get from a world. (Mostly because computers don't "hope".)

When trying to understand the concept of Data APIs that run on the web, it can be tough to get a mental image of what's really happening. The explanation that has always helped me is to imagine building a website for machines.

When building a website for humans, we care about things like information architecture and clear presentation of content that utilizes visual design to reinforce hierarchy, relationships, and meaning. Machines (computers) also need a solid information architecture, but the way we present data for machines is very different from how we present data for users.

To give machines what they need, APIs respond with information that is structured so it can be interpreted by whatever software is making the request. Typically, this means JSON or XML formatting, which are discussed below. Both of these formats lack the visual component that is essential to delivering information for humans. Machines are terrible at interpreting graphic design, but they are very good at keeping track of intricate hierarchies as long as the structure is consistent and syntactically correct.

When we use APIs we are asking our software to do some web browsing for us, and the information it retrieves makes our own application much more valuable to the humans using it.

REST

There are many kinds of APIs, but the dominant architecture on the web today is called "REST". You may see references to "RESTful APIs" or "REST-based APIs"—these are all ways of talking about an API that uses REST architecture.

REST stands for "Representational State Transfer", and although it sounds quite complex, it's something that you encounter every day. REST is the way the web works, and the concept was, in fact, defined by Roy Fielding (co-founder of the Apache web server project) as he worked on defining the HTTP protocol. REST feels natural on the web because REST is the way the web works.

Requests

REST is an architectural principle based on the client-server model. REST assumes that there is a client (web browser, Javascript app, mobile app, etc.) that wishes to consume data from a server (web server, API service, etc.).

When we make a request to a web server or API service, we are essentially sending our own file of information. The request is structured in a way similar to any HTML response we get. Requests have a "method", "headers", and a "message body". Each of these parts of the request is taken apart and inspected by the web server (or API service) in order to understand how to respond.

Methods

Methods are the "verbs" of the web. RESTful services use the base methods defined in the HTTP specification to handle data. These methods are:

- **GET**—Retrieves information for the client without altering any data on the server. This is commonly known as the "read" method.
- **POST**—Creates an entirely new record or data object on the server based on information sent by the client. This is often called the "create" method, and it is the most common method for HTML forms to use in order to send data back to the server.
- **PUT**—Updates the specified record with the data contained in the request. This is known as the "update" method.
- **DELETE**—Deletes the record specified by the client on the server. Most people don't feel the need to rename the "delete" method.

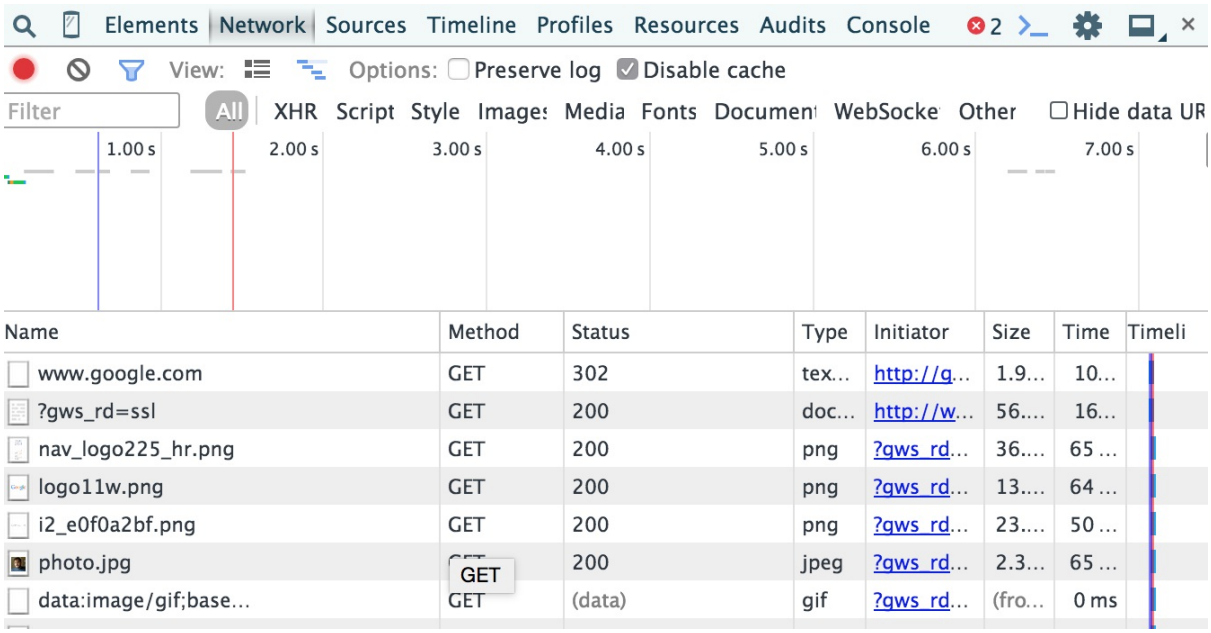
In working with any data content, we often use the CRUD acronym to describe basic functions of handling data: Create, Read, Update, Delete. The basic methods of HTTP mirror this concept very well, making them efficient at handling our data management needs.

Headers

In any request, the headers are of importance. Headers contain all sorts of data used by the server to determine the proper response. Each header has a meaning and an impact on the request, but for the most part we never pay much attention to the headers being sent each time we access a website. Here is a snippet of the headers sent when requesting `google.com`:

```
GET / HTTP/1.1
Host: google.com
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/43.0.
2357.134 Safari/537.36
X-Chrome-UMA-Enabled: 1
X-Client-Data: CIm2yQEIgfjbJAQiptskBCMG2yQEI7YjKAQifl8oB
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
```

You can easily see the request is using the GET method, and you can probably figure out quite a few more details about the request (language, browser, etc.). You can view the headers used in any request made by your web browser by looking at the `Network` tab in the Chrome developer tools:



Click any of those requests and you'll see a summary of the request and response including all headers and data. (Please note: Being able to look at what your browser is doing like this can be very helpful when trying to figure out why your API-dependent app is not working properly.)

Sometimes when working with data APIs you will need to set a header. For example, authorization to use an API may involve you sending your API token as an HTTP header. This header is sometimes specified like so:

```
Authorization: Token 9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b
```

Most libraries you use to consume data APIs will provide a convenient way to set a header to a specified value. Many frameworks are specifically designed to minimize how much manual management of headers you must perform. When you set configuration options and other setup details you may actually be specifying that the framework or library will include or exclude specific headers. It's worthwhile to be cognizant of what headers are being used in your API requests.

Message Body

The request contains a message body, which contains any data the client is sending to the server. In the case of POST and PUT methods, data is sent to the server for interpretation and then handling (usually it is being saved into the database).

When HTML forms are sent using the GET method, the data from the form fields is serialized into "query string parameters". These are attached with a question mark (`?`), like so:

```
http://domain.com/catalog/search?q=slippers&brand=fuzzybunny&gender=male&size=14
```

In the example above you can see several query string parameters have been defined: `q` , `brand` , `gender` , and `size` . It is very common to use query string parameters to represent things like search filters. On many ecommerce websites you will notice that the filters you choose in the catalog are easily identifiable in the query string parameters for the catalog page you are viewing.

The URL above could have come from an HTML form with fields called `q` , `brand` , `gender` , and `size` that was sent using the GET method. This is handy in cases where we want to be able to copy/paste a URL and have our friend see the same results (in this case, a selection of very large fuzzy bunny slippers).

In other situations, it would not be proper to send the data contained in an HTML form as part of the URL. The URL of a request can never be encrypted, since it must be used to route all of the pieces of the request from client to server. Therefore, any data sent using the GET method is going to be visible to all the network nodes those bits pass through. In the case of sensitive information (such as passwords or credit card numbers), it's much better to use POST and PUT so that the message body content can be properly encrypted via HTTPS and much less visible to any nefarious agents.

State(less)

REST is "stateless". (So is the web.) This means that each request to a web server (or a REST API service) must define all of the parameters to receive the desired response. In RESTful services, all of these parameters are packed into the URL and form data. Imagine the number of times you've seen a URL that looks like this:

```
http://domain.com/profile/username/edit/
```

Presumably, if `domain.com` follows best practices, that URL would allow us to edit the profile for the user with the specified `username`. The server responding to this request would be able to fetch the data for the specified `username` and serve us back a web page designed to allow us to edit that information. No extra information is required to receive our desired response, and no information from this request will be used in any other response the server sends.

When we edit the information on the page and click submit, we are making a brand new request to the web server. This request defines all of the information the server requires in order to respond to our request. This time, when we send data to the server we include "form data", which is the data collected using HTML `form` elements (`input`, `select`, etc.). This form data provides the information we want the server to put into the database for us.

Data

REST APIs respond with data that can be contained in an HTTP response. This generally means text. The text can return a URL reference to a media file, of course, but then you must actually use that URL properly in your HTML to allow the user to see that media. You might insert an image tag, or a video tag, or whatever other HTML structure is needed to display the media.

The text responses returned by REST APIs tend to either be formatted as JSON (Javascript Object Notation) responses or XML (eXtensible Markup Language) responses. The trend today is toward JSON since it is more easily consumed by Javascript applications, and for our purposes we will expect JSON responses.

Here is a sample API response from OpenWeatherMaps.org for the query `api.openweathermap.org/data/2.5/weather?q=seattle,wa,us&units=imperial&id=4717560d`:

```
{
  ...

  "weather": [
    {
      "id": 701,
      "main": "Mist",
      "description": "mist",
      "icon": "50d"
    }
  ],

  ...

  "main": {
    "temp": 61.05,
    "pressure": 1018,
    "humidity": 63,
```

```
    "temp_min": 57.2,
    "temp_max": 64.99
  },
  "wind": {
    "speed": 4.54,
    "deg": 220
  },
  "clouds": {
    "all": 90
  },

  ...

  "id": 5809844,
  "name": "Seattle",
}
```

In this example, I have removed a few pieces of the response to make it a little easier to read. (The missing pieces are indicated by the `...` lines.)

As you read through this example response, it's pretty easy to see some interesting data fields. We can see that this data object contains several "root level" or "base" attributes: `weather`, `main`, `wind`, `clouds`, `id`, `name`. This is the "current weather" data for Seattle. We are given the ID, which is a more reliable way of not getting confused with the other cities called "Seattle". We also have a weather summary (at the moment "Mist"), current temperature (61.05F), cloud coverage (90%), and more. With this information, we can build a solid weather report.

When this data object is received by our webapp, we will parse the information into a JavaScript object. We can call that object whatever we want (perhaps `weatherReport`?) and we can pipe that into our view templates like this:

```
<h1>Weather report for {{weatherReport.name}}</h1>
<p>Current conditions: {{weatherReport.weather.main}}, {{weatherReport.main.temp}}F</p>
```

And then our users would see this rendered in our webapp:

Weather report for Seattle

Current conditions: Mist, 61.05F

This is just a quick preview of how handy it is to use a JavaScript toolkit to interact with data APIs that deliver JSON responses. We will explore much more about how to make this all work in our webapp over the next few pages of this chapter.

Exploring APIs

Whenever we begin using a new data API, it's good to spend a little time exploring the API documentation and trying requests using an API browser tool like Postman.

Postman allows us to construct HTTP requests to any API. We can set necessary headers, query parameters, form data, and more. Results are clearly displayed so we can truly begin to understand the shape of the data that comes back from the API. Each API will deliver a different type of data object, and all the attributes contained in these responses become available to us inside our application once we set up our API calls correctly.

That means we can use an API browser tool to get a clear view of what information exists in the data coming back from your API service. Since the API data objects are translated directly into JSON objects in our application, we can easily see how to access the specific information we require for your purposes.

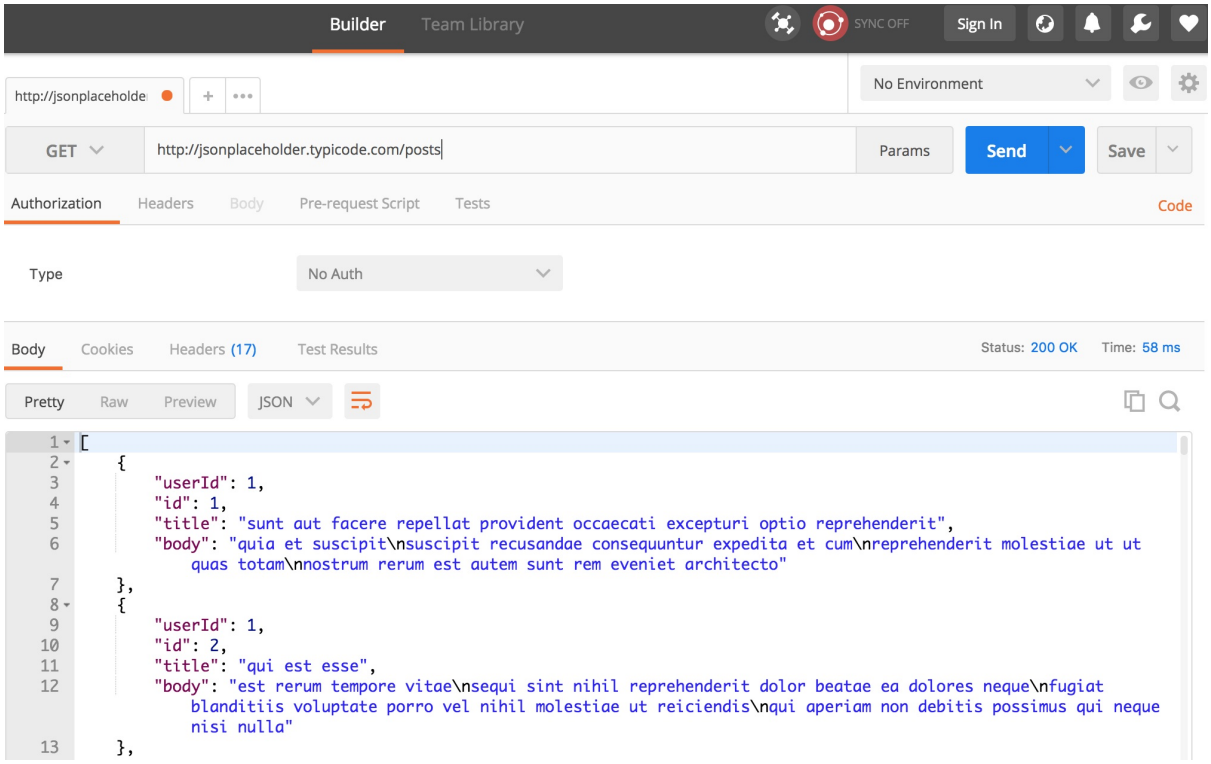
These tools also help us form proper API requests. If our request is not properly formed, then we will not get the data we need back from the API service. Luckily, modern JavaScript modules make it easy to create basic requests, but it can still be tricky to work out how to form our request in the first place. A good API browser tool will help us formulate correct requests and will allow you to know exactly what you are getting back. You can even use it to help test edge cases and learn how the API responds if, for example, you send some data value it does not expect.

The Postman Documentation site has a lot of useful information, tutorials, and guides for using Postman to explore API services. We should read up on how to use this tool, or find a similar tool you prefer and use that one. It is very essential to have a way to view the data coming into your app, and to test your app's API queries, apart from your custom Javascript code.

Exploring an API With Postman

We can plug in API URIs and explore them in Postman. Let's practice with a common placeholder resource for API content, JSON Placeholder API. We can see from the documentation on their homepage that they offer a few different API endpoints to experiment with. We will experiment with the `/posts/` endpoint in these examples.

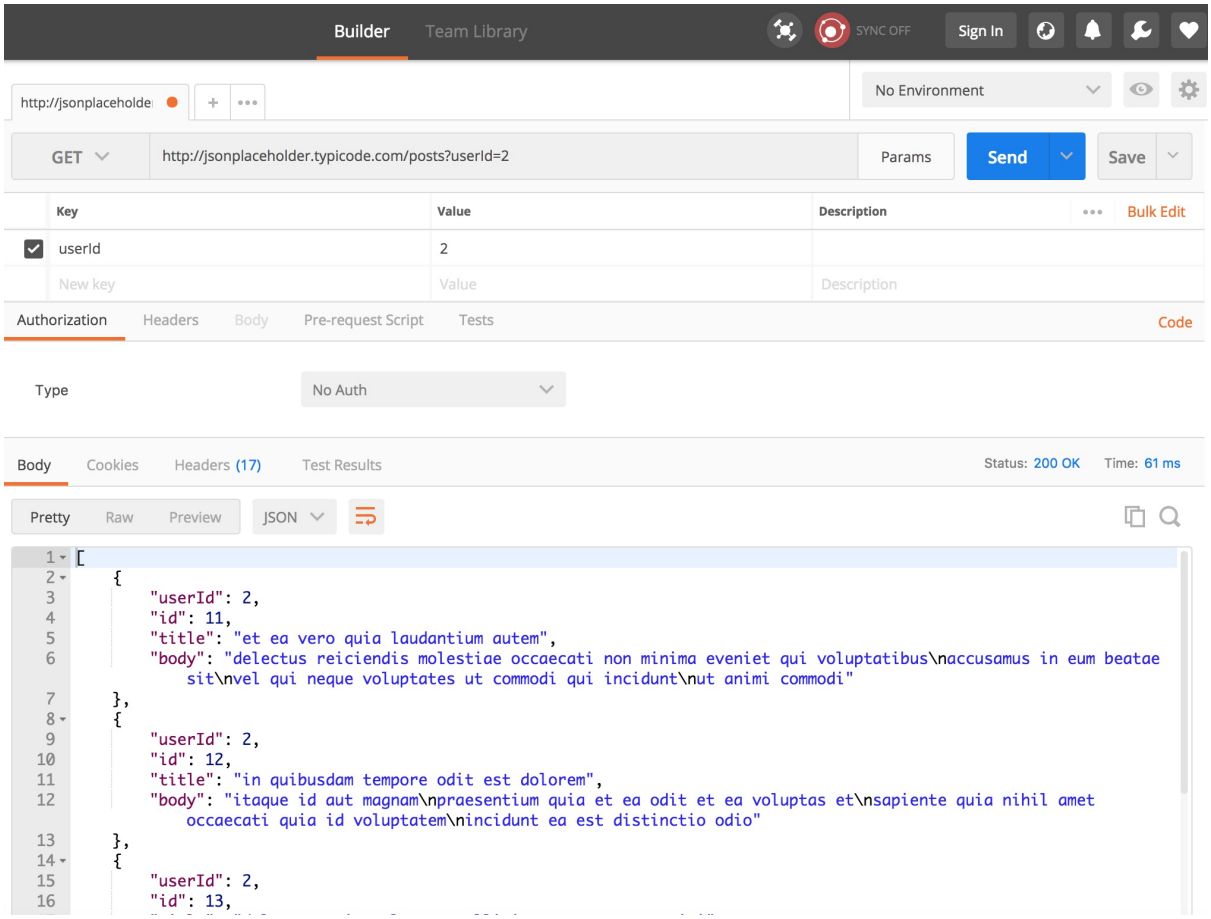
First, let's set up a query to `http://jsonplaceholder.typicode.com/posts`. We should make this a `GET` query, and we do not need to supply any additional parameters or authentication. Paste the API URL into the location box and click Send. We should see a result that looks something like this:



Postman Query

We can see in this image that we have received a response that is an array of JavaScript objects representing the "posts". These posts are described using JSON notation, so we should recognize the structure of the information. We can see that each post object has `userId`, `id`, `title`, and `body` properties.

We could alter our query by clicking the "Params" button in Postman to reveal a way to set key/value pairs that will be translated into the query string of the API URL. If we click the Params button we can set the `userId` value to `2`. This will perform an API request that will only fetch the posts that have been authored by the user with the `userId` of `2`. Here is what that looks like:



Postman Query for `userId=2`

As we can see, the results have changed so that they all have the same `userId` value. This is a common way for APIs to filter results according to our request. Other APIs will reveal different information and will accept different parameters. It's worthwhile to test out any API we plan to work with using a tool like Postman in order to get a feeling for the way the API works. Although most APIs adhere to the rules of RESTful API design, each API is still a unique system with its own quirks and features.

Using other APIs

You can use the patterns described in this book to use any other REST API that serves JSON results. There are many, many APIs that fit that description. Some APIs require you to do more arduous authentication or to have your app approved before you will be granted developer privileges. Other APIs have steep charges for using them, and although you may be able to develop against them reasonably it would be prohibitively expensive to release a website using that API. These considerations and more should inform your decision as you look for APIs.

The Suggested API List in Appendix B of this book is a great place to find other APIs worth trying out.

Requesting Data from an API

In order to make requests to an API and then receive and process the data from the API responses, we will use a JavaScript module designed for this purpose. The current most popular choice when working with Vue.js is a module called Axios, which is a Node.js module that can be used by virtually any JavaScript application. It is possible to use an alternative, including the formerly-preferred module, Vue Resource. Most of the content of the examples that follow will apply to any API request module, although the specifics of the implementation will vary.

Install Axios

In order to use Axios in our application, we must first install it. Since we are already using a well-tooled environment, we can run the `npm install` command like this:

```
npm install axios --save
```

This command installs Axios for use in our application, and it also saves Axios as a dependency in the `package.json` file that tracks all of the modules we use in our project. Once we've run this command, we can look in our `package.json` file and see that it is listed. (Look in the `dependencies` object.)

```
"dependencies": {
    "axios": "^0.16.2",
    "vue": "^2.5.2",
    "vue-router": "^3.0.1"
},
```

Use Axios in a Component

Now that we've made Axios available to our application, we can use it inside a component. The first step to make an API request within a component is to properly import Axios. This import statement should go at the top of the `<script>` section in our component:

```
import axios from axios
```

Now that we have imported `axios`, we can reference that within our component logic. We will use `axios` to make a request to an API server and then receive the response, which we will then make available as a part of the `data` object. Before we can actually perform the data request, we should initialize the `data` object properties that will be used in the template:

```
data () {
  return {
    posts: [],
    errors: []
  }
}
```

We have initialized two values: `posts` and `errors`. Both of these are set to empty arrays. We will populate these values according to the results of our API requests. We can reference these two values in our templates in order to display the information returned by our API request.

Creating the Request

In order to actually perform an API request, we must write some code to do so. It's very common to have a component that requires data from an API when it is first loaded. In these cases, we want to load the data as soon as the component is loaded, so we can use the `created` property of the Vue component. We will see an example shortly of using the `created` property to load data when a component is loaded, but it's worthwhile to remember that this same technique can be applied to writing component methods. In those cases, the data could be fetched whenever the user performs a specific action. This is a common way to implement API requests to refresh data or submit data customized for the user's needs.

For our first example, let's look at an API request made when the component is first loaded:

```
<script>
import axios from 'axios';

export default {
  name: 'HelloWorld',
  data () {
    return {
      posts: [],
      errors: []
    }
  },
  created () {
    axios.get(`http://jsonplaceholder.typicode.com/posts`)
    .then(response => {
      this.posts = response.data
    })
    .catch(e => {
      this.errors.push(e)
    })
  }
}
</script>
```

Notice that `created ()` is defined in a very similar way to the `data ()` function. Within the `created ()` function, we only make one JavaScript statement: a somewhat complex command using `axios`. Although this doesn't look much like the code we've been used to writing/reading, it's actually the most simple way to write an API request. Once we take a closer look, we should see that this is not as complex as it first appears.

The first line of the command is:

```
axios.get(`http://jsonplaceholder.typicode.com/posts`)
```

This invokes `axios` and supplies the URL for the API request. This command results in the creation of a "promise", which is a JavaScript object that works as a placeholder for some information or function that will take a moment to be resolved. (We can read more about promises here.) Because the request to the API cannot be resolved instantly (it takes some time to receive the data from the server, just like when we are downloading other information from a web server), we need this promise object to handle the receipt and transformation of the API data into something we can use in the application.

The next clause of the `axios` command defines what happens after the promise is resolved:

```
.then(response => {
  this.posts = response.data
})
```

This is a `then` clause, and it will be executed when the promise is completed, regardless of what the result of the API call was. The way we use Axios in our applications, we want to use the JavaScript "arrow function" to define the function that will be executed when the `then` clause is invoked (which will be when the API response is received).

Using the arrow function syntax, the argument for this function is named `response`. This argument will be passed to the code block defined between the curly braces. (The arrow is a reminder that the `response` value will be "injected" into the code that follows.) The `response` value is an object that represents the API request we made with `axios`. The actual information received from the server is stored in `response.data`. So the `then` clause executes an arrow function that sets `this.posts` equal to `response.data`. Our template has access to `this.posts`, so it can successfully parse that data and we can output the information using a loop.

The final clause in this request is the `catch` clause:

```
.catch(e => {
  this.errors.push(e)
})
```

The `catch` clause executed when there is an error in the request. It also uses the arrow function syntax to define the function that will be executed when an error is detected. The error is captured as the value `e`, and passed into the arrow function. We only have one line of code here, which will add this error to the array of `errors` we defined in the component's `data` object. Again, our templates will have access to these values, allowing us to present this information to our users.

```
<ul v-if="posts.length > 0">
  <li v-for="post of posts">
    <p><strong>{{post.title}}</strong></p>
    <p>{{post.body}}</p>
  </li>
</ul>

<ul v-if="errors.length > 0">
  <li v-for="error of errors">
    {{error.message}}
  </li>
</ul>
```

We can see from this example code how the `posts` and `errors` values can be used in our templates to output useful information for our users. In this case, all of this code would combine to show the list of sample results provided by the JSON Placeholder API.

Requests with Parameters

Axios actually lets us do quite a few different things to make our API requests more complex and functional. One case that comes up often is using user-submitted information to modify a request. We often want to have a `refreshResults` or a `search` method on our components that can be invoked to provide the user with better information.

Let's take a look at an example request made to the Datamuse API, which can return all sorts of word data. In this case, we will use a query that returns synonyms for words or phrases:

```
<template>
  <div class="hello">
    <h1>Example Data Request</h1>
    <p>Find synonyms for <input v-model="phrase"> <button v-on:click="findSynonyms">Search</button></p>
    <ul v-if="synonyms.length > 0">
      <li v-for="synonym of synonyms">
        <p><strong>{{synonym.word}}</strong></p>
        <p>{{synonym.score}}</p>
      </li>
```

```
    </ul>

    <ul v-if="errors.length > 0">
      <li v-for="error of errors">
        {{error.message}}
      </li>
    </ul>
  </div>
</template>

<script>
import axios from 'axios';

export default {
  name: 'HelloWorld',
  data () {
    return {
      synonyms: [],
      errors: [],
      phrase: 'logic'
    }
  },
  methods: {
    findSynonyms: function(){
      axios.get('https://api.datamuse.com/words', {
        params: {
          ml: this.phrase
        }
      })
      .then(response => {
        this.synonyms = response.data;
      })
      .catch(error => {
        this.errors.push(error);
      });
    }
  }
}
</script>
```

In this example, we can see the template and the logic for a component that allows the user to type in a word or phrase and then receive synonyms back. If we look at the template, we see that it generally matches what we saw in the previous example. The new template includes a text input field and a button to allow the user to type in their word or phrase and then trigger the `findSynonyms` method to execute by clicking the "Search" button.

If we look at `findSynonyms`, we see that it executes an axios call similar to the previous example. The big difference is that this call includes a little extra configuration object that defines an extra "parameter" for the API request. The parameter is called `ml` and it is set equal to the value the user typed into the `phrase` input.

The result of this configuration is that the API request has the parameters appended to it as query string parameters. If a user does a search for the word "logic", then the API request URL is:

```
https://api.datamuse.com/words/?ml=logic
```

If there user were to type in a different word, it would be appended to the URL instead of "logic". This allows the user to get the specific results they desire from our interface.

The same technique for appending parameters to a request can be used to append API keys, other static parameters that make results more useful for users, and much more. We can also use expanded configurations of the `axios` call to change the request method or add other elements like authorization headers to the API request.

Advanced JavaScript Concepts

The concepts of **arrow functions** and **promises** are difficult to grasp at first. In this book we have only explained them far enough to use these concepts in relation to the API calls we're making. However, there are many ways that arrow functions and promises can be used to enhance our code. Both of these features were added to JavaScript to address weaknesses that have bothered developers for ages. In the early stages of our development careers it might be difficult to grasp just how valuable they are, but as we progress and grow as developers we will see these concepts used more often.

Quiz: Using API Data

Please enjoy this self-check quiz to help you identify key concepts, points, and techniques discussed in this section.

Visit Quiz Online

The quiz on this page has been removed from your PDF or ebook format. You may take the quiz by visiting this book online.

Project: Using API Data

In this project, we will use the Using APIs repository as our starting point. This project asks us to create a rhyming thesaurus using the Datamuse API. The Datamuse API is a wonderful tool that allows us to do all sorts of interesting look-ups to find suggested words. In this project, we will be looking for words that rhyme with one word (or phrase), but which also have some relation to another word (or phrase): a rhyming thesaurus. This is a tool that could be incredibly useful for poets, songwriters, battle rappers, and many others.

Please note: There are many features in the Datamuse API that can be used to create other types of applications ranging from avant garde algorithmic poetry to pragmatic, helpful writing tools. Feel free to explore the many possibilities of this free and open API.

Review the Requirements

This project requires us to make an API call when the user clicks a search button. We want to take some input from the user, which we will use to form our API request. First, we will walk through tasks outlined in the `Rhymesaurus.vue` file, which guides us toward creating the rhyming thesaurus interface. Then, we are asked to create a brand new Vue.js component that makes use of a different API call to the Datamuse API. Our goal is to link together the Rhymesaurus and our new tool within our site so users can access whichever they need.

Here are the Basic Requirements from the project's `README.md` file:

`/src/views/Rhymesaurus.vue`

`<template>`

- Set up an event handler to trigger the `findWords` method when the `form` is submitted
- Use a conditional to control the display of the `ul.results` element so it only displays when results are ready to be shown
- Use a loop to process all of the results into list items
  - Output the word in each list item
  - Output the score for each word in each list item
- Use a conditional to control the display of the `.no-results` message, which should only show when the user has attempted a search and no words have been found
- Use a conditional to control the display of the `ul.errors` element so it only displays when there are errors ready to be shown
- Use a loop to process all of the errors and display them for the user

`<script>`

- Import axios for use in the component logic
- Create a method called `findWords` (don't neglect to add the `methods` attribute to this component)
- Within the `findWords` method, create an `axios.get()` statement that will make a request to `https://api.datamuse.com/words`
  - Define the `params` for `ml` (which should map to the `phrase`) and `rel_rhy` (which should map to the `rhyme`)
  - Define a `then` clause that sets the component's `results` value to the value of `response.data`
  - Define a `catch` clause that will add any error to the `errors` array in the component

`/src/views/NewComponent.vue`

(Please do not actually name the new component `NewComponent.vue`. This name is only being used for reference here. Name the component according to whatever feature it provides.)

- Create a new component (use the boilerplate component code, or copy the component you just created)
- Refer to the Datamuse API documentation to determine a way to modify your code to provide an alternative way to find words (e.g. "sounds like", "related to", "adjectives that go with a word", "words that often follow a word", etc.).
- Implement a similar interface to perform this new search
  - Create the necessary template to allow the user to enter at least one search parameter
  - Create the necessary method to handle the form submission and API request
- Implement the proper template elements to output the results to the user
  - Show all relevant results returned by the API
  - Show all errors to the user
  - Show a message when no results have been found, so the user knows the system is working even though the data is not there

`/src/router.js`

- Add the new component to the import statements in the router definitions file
- Add the new route to the router definitions list (use a sensible URL and name for it)

both `/src/views/Rhymesaurus.vue` and `/src/views/NewComponent.vue`

- Add navigation elements to provide links between the two search pages
- Use proper `router-link` tags to create links
- Allow the user to easily switch between the two pages in the application.

Working the Project

For this project walkthrough, we will focus on the initial work to complete the rhyming thesaurus. After we finish that work, we will offer some guidance to putting together the new component, but that will primarily be left as a challenge to complete based on understanding from previous sections and projects. Refer back to how we previously created new components and set up new route definitions in the Responding to User Input project.

Set Up the API Request

Rather than jumping immediately into the template work, it's probably more streamlined for us to write the API request so we can test it as we build out our template features. The first `TODO` in the `<script>` area of the `Rhymesaurus.vue` component asks us to import `axios` for use in our `findWords` method. Assuming we have installed `axios` in the project with the standard `npm install --save axios` command, this import statement should work:

```
import axios from 'axios';
```

That makes the `axios` object available within our component logic, so we can then write the `findWords` method. To add the `findWords` method, we must create the `methods` property on the component object. Then, we can define `findWords` like so:

```
methods: {
  findWords: function(){
    axios.get('https://api.datamuse.com/words', {
      params: {
        ml: this.phrase,
        rel_rhy: this.rhyme
      }
    })
    .then(response => {
      this.results = response.data;
    })
```

```
      .catch(error => {
        this.errors.push(error);
      });
    }
  }
```

As we can see, `findWords` is a fairly simple method that uses `axios` to make a call to the Datamuse API. The request to the API is formed around the `ml` and `rel_rhy` parameters. These two parameters are explained in the Datamuse API Documentation (keep that link handy because we'll refer to it later on, when we build our new component). The `ml` parameter indicates "means like", which returns synonyms for words and phrases. The `rel_rhy` parameter attempts to find words that rhyme with the supplied words. By combining these parameters we can find words or phrases that are synonymous with `this.phrase` and which also rhyme with `this.rhyme`.

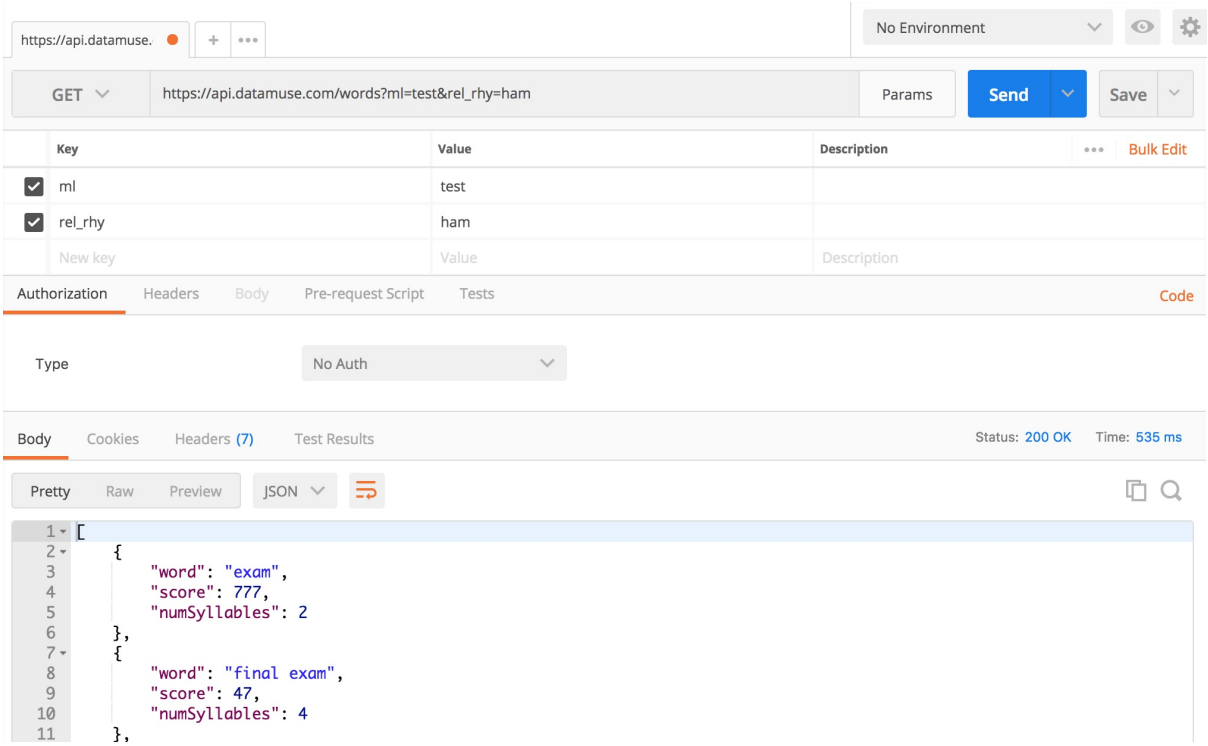
Notice that in addition to the URL for the API request, we have also supplied a configuration object in the `axios.get()` call. This configuration object only specifies `params` at the moment, but if we needed we could specify other types of configuration options (e.g. `headers`, `transformResponse` functions, etc.). It's critical that the `params` property names match the parameter names used by whatever API we are calling (in this case, the Datamuse API). Refer to the API documentation for the specific names an API uses and then use those in the `params` object when setting up `axios` requests.

This is an example of the kind of unorthodox tool we can build by using discrete features of an API. There are plenty of tools and books to look up rhymes for words (we call them "rhyming dictionaries"). And there are tools and books to look up synonymous words (we call that a "thesaurus"). But what we are creating here is a "rhyming thesaurus" (or "Rhymesaurus"), which is a much more unusual type of tool. It's also the kind of cross-lookup that would be difficult to replicate in print, making it especially suitable for building as a software application.

If we are using Postman (which we should be using) to test our API queries, we can try something like this:

```
https://api.datamuse.com/words?ml=test&rel_rhy=ham
```

This API request should return words that are synonymous with "test" and also rhyme with "ham". The list should include "exam" and variations on that word. If we test that query in Postman, we can see the results we hope for:



API Results in Postman

Now that we have verified our API results using Postman, and we have set up our API call using `axios`, we can continue to edit the template.

Rhymesaurus Form

In the template we primarily need to set up the input form and then the output of the results and any potential errors. We also want to let our user know if the API request completed successfully, but no words were found. (If we don't let our user know that no words were found, then they are almost guaranteed to perceive that case as a breakage in our software.)

To do this, we must first set up the event listener to handle the form submission event. This follows the same pattern we used in the previous project:

```
<form v-on:submit.prevent="findWords">
```

Again, we are using the `v-on` directive, and we specify the `submit` event. We use the `.prevent` modifier to prevent all of the default event handlers from executing. Finally, we specify the `findWords` method to handle the form submission event. Whenever the form is submitted, `findWords` will be executed.

Once we have set up the form submission handler, we must connect our form fields to our component data. As with the previous project, we will use the `v-model` directive to do that:

```
<p>Find rhymes for <input type="text" v-model="rhyme"> related to <input type="text" v-model="phrase"> <button type="submit">Search</button></p>
```

There is not much special about this implementation. We reference the `rhyme` and `phrase` values in the data object. It's important to remember that these values must be initialized in the content data object in order to be used in the template like this.

To Form or Not to Form

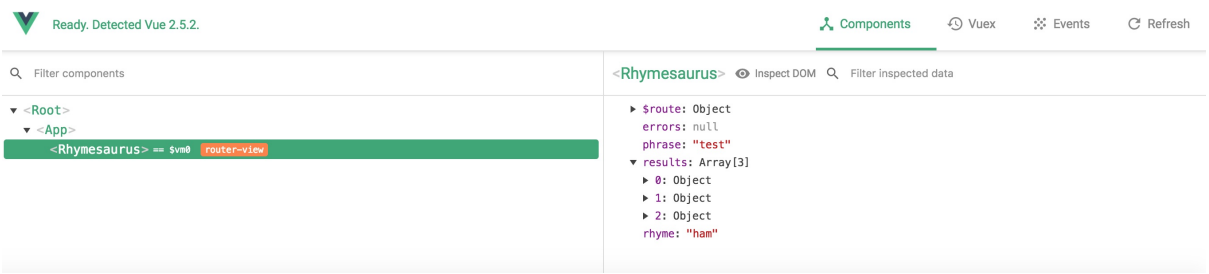
It is possible to use HTML input fields and buttons to simulate a form experience using only a `v-on:click` directive on the button. This can work, but it only works when the button is clicked. By using the form, and tracking the form submission event instead of the button click event, we leverage the default behavior of forms in the web browser to allow users to press the `Enter/Return` button to submit the form. This is a nice enhancement that works better for users who prefer keyboard navigation of a website (often either users with accessibility needs or power users who want greater efficiency). Where possible, it's usually better to use a fully defined form to handle user-submitted data.

Now that we have the form wired up to the proper component data values and the proper form submission handler, we can test our page in the browser. When we click search, we will not see any results (because we haven't written that part yet) but we will see the XHR request made by Axios in the devtools console:

```
XHR finished loading: GET "https://api.datamuse.com/words?ml=test&rel_rhy=ham".
```

XHR Request in Devtools Console

We can also see the information in the Vue Devtools tab. After we have made the request we can see that the `results` value is populated as an Array with 3 items:



Vue Devtools Results

Now we can set up the output of this information for the user to view.

Data Output in Template

To output the data, we will create a list of items. The list will use the `.results` class to label it and control styling. We will make a loop using the `v-for` directive on each of the list items. In the end, it will look like this:

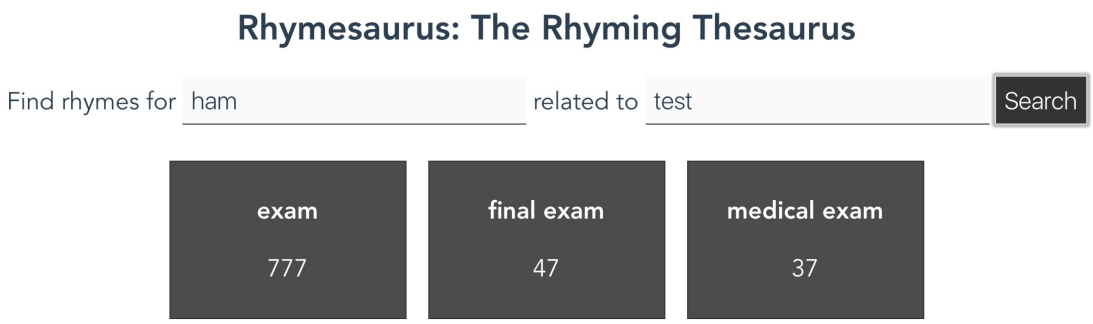
```
<ul v-if="results && results.length > 0" class="results">
  <li v-for="item of results">
    <p><strong>{{item.word}}</strong></p>
    <p>{{item.score}}</p>
  </li>
</ul>
```

We want to hide the list until we have results. We also want to hide the list if we have zero results returned from the API. We can notice that `results` is initialized to `null`, which means that until the user triggers a form submission event the results will not show. Once the `findWords` function is executed, `results` will be an Array. If it has any items in it (if `results.length` is greater than `0`), then the `.results` list will be shown.

On the `li` element, we have created a `v-for` loop. We are using the somewhat generic term `item` to label each item in the `results` array as they are processed through the loop iterations. Each `item` is an object with `word` and `score` values (we can see all of the properties on each item returned by the API in Postman when we inspect our API

call). We output the values of those properties for the user to view.

Once we have this template code in place, we can run a test and we should see the results.



Testing the Data Output

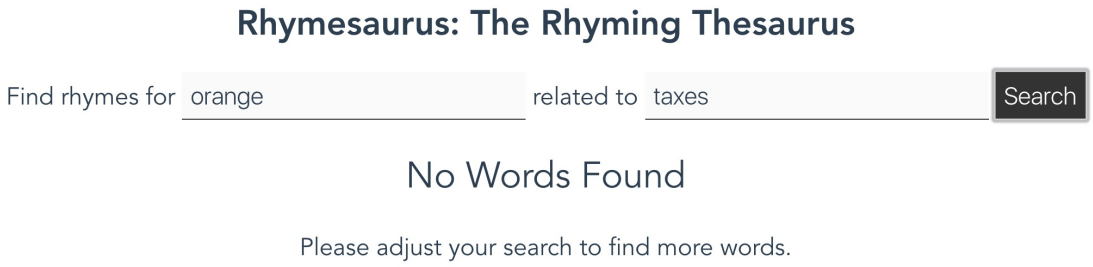
Handling No Data

Sometimes the user will search for a rhyme and synonym that does not exist. In these cases, the API will return zero results. If we do not let the user know that the API returned zero results, then they are likely to believe our software is not working correctly. It's crucial to provide this message so the user can have a much better experience and understand that they should adjust their search to get better results.

To accomplish this, we will use a `v-else-if` statement to control the display of a message that tells the user to adjust their search parameters. This is what it looks like:

```
<div v-else-if="results && results.length==0" class="no-results">
  <h2>No Words Found</h2>
  <p>Please adjust your search to find more words.</p>
</div>
```

Notice that in this conditional, we are once again checking to see that `results` exists. We do not want to show an error message right away when a user hits our page. We only want to show the error message if the `results` array has been created and if the length of the `results` array is zero. Once we've set this up, we can do a search that will not produce any results:



Orange and Taxes: No results

In this example, we search for words that rhyme with `orange` and are related to `taxes`. There are "No Words Found," which makes perfect sense.

Handling Errors

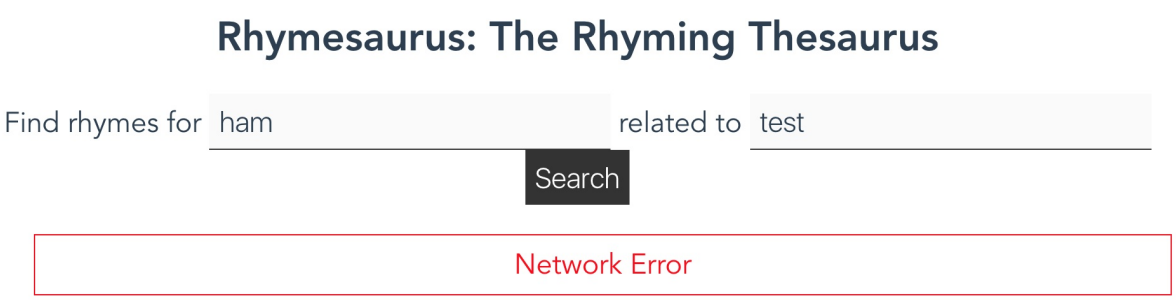
We handle the display of error conditions in much the same way that we handle the display of results and the "no words" message: We will create a conditional to see if the `error` value contains at least one error.

```
<ul v-if="errors.length > 0" class="errors">
  <li v-for="error of errors">
    {{error.message}}
  </li>
</ul>
```

The conditional here is almost identical to the conditional we used to control the display of the `.results` list. We can test this display by forcing an error in our API request. The easiest way to do that is to alter the first line of our `axios` command:

```
axios.get('https://api.dataTESTmuse.com/words', {
```

By inserting the word `TEST` into the domain name, we will trigger a domain error (because that domain does not exist). This comes across to the user like so:



Network Error

We could also try to adjust our URL to trigger other errors within the API (e.g. 404 errors, 500 errors, etc.). Each of these errors should be displayed to the user with a somewhat helpful label. Although the error message might not allow the user to solve the problem, it at least indicates that something has gone wrong beyond their control.

Creating the New Component

The remaining project requirements ask us to create a new component that will allow us to create a different kind of word-finding tool. This new component can essentially use the same setup as the `Rhymesaurus.vue`, and it would not be bad to start by simply copying the existing component file with a new name. To change the behavior, we only need to update the form, the data bindings, and the API call that is being formed in our `axios` call.

Once we have created the new component, we are asked to link these two pages together using a standard router definition and the `<router-link>` element. Building this little navigation should be very similar to setting up routes and links in the previous project. Refer back to those pages for additional instruction.

The toughest part of creating the new component is deciding on which features of the Datamuse API to combine. There are many possibilities, and we are encouraged to explore them to find something interesting to work with. If we're feeling uninspired, here are some ideas along with the URLs to make the API calls.

- `https://api.datamuse.com/words?rel_jjb=car` — Find adjectives for a given noun
- `https://api.datamuse.com/words?rel_jjb=car&rel_rhy=ham` — Find adjectives for a given noun that rhyme with another word
- `https://api.datamuse.com/words?rel_bgb=statue` — Find words that frequently precede the given word

- `https://api.datamuse.com/words?rel_bga=statue` — Find words that frequently follow the given word

There are many other lookups available on the Datamuse API that can be combined to create interesting tools. Try out queries in Postman to get a feel for what sort of results can be achieved.

Wrapping Up

The final `Rhymesaurus.vue` file looks like this when we're done (excluding the links to the new component):

```
<template>
  <div class="rhymesaurus">
    <form v-on:submit.prevent="findWords">
      <p>Find rhymes for <input type="text" v-model="rhyme"> related to <input type="text" v-model="phrase"> <b
utton type="submit">Search</button></p>
    </form>
    <ul v-if="results && results.length > 0" class="results">
      <li v-for="item of results">
        <p><strong>{{item.word}}</strong></p>
        <p>{{item.score}}</p>
      </li>
    </ul>
    <div v-else-if="results && results.length==0" class="no-results">
      <h2>No Words Found</h2>
      <p>Please adjust your search to find more words.</p>
    </div>

    <ul v-if="errors && errors.length > 0" class="errors">
      <li v-for="error of errors">
        {{error.message}}
      </li>
    </ul>
  </div>
</template>

<script>
import axios from 'axios';

export default {
  name: 'Rhymesaurus',
  data () {
    return {
      results: null,
      errors: [],
      phrase: '',
      rhyme: ''
    }
  },
  methods: {
    findWords: function(){
      axios.get('https://api.datamuse.com/words', {
        params: {
          ml: this.phrase,
          rel_rhy: this.rhyme
        }
      })
      .then(response => {
        this.results = response.data;
      })
      .catch(error => {
        this.errors.push(error);
      });
    }
  }
}
</script>
```

```
<!-- Add "scoped" attribute to limit CSS to this component only -->
<style scoped>
.rhymesaurus {
  font-size: 1.4rem;
}

input[type="text"]{
  border-top: none;
  border-left: none;
  border-right: none;
  border-bottom: 1px solid #333;
  width: 300px;
  font-size: 1.4rem;
  color: #2c3e50;
  font-weight: 300;
  background: rgba(0,0,0,0.02);
  padding: 0.5rem;
}
button{
  background: #333;
  padding: 0.5rem;
  font-weight: 300;
  color: #fff;
  border: none;
  cursor: pointer;
  font-size: 1.4rem;
}
h1, h2 {
  font-weight: normal;
}

ul.results {
  list-style-type: none;
  padding: 0;
}

.results li {
  display: inline-block;
  margin: 10px;
  border: solid 1px #333;
  padding: 0.5rem;
  width: 200px;
  min-height: 100px;
  color: #fff;
  background: rgba(0,0,0,0.7);
}
ul.errors {
  list-style-type: none;
}
.errors li {
  border: 1px solid red;
  color: red;
  padding: 0.5rem;
  margin: 10px 0;
}

a {
  color: #42b983;
}
</style>
```

The additional requirements ask us to create a new file in `/src/components/` for the new component and to add a new route to the `/src/router/index.js` file. These changes are up to us to work out according to our unique vision.

Build and Deploy

Once we've finished our work, we can build and deploy the project. This project has been configured to build to the `docs/` directory, so we can follow the same pattern we used before:

1. Execute the `npm run build` command to build the files into the `docs/` directory.
2. Commit all of our code.
3. Push the code up to GitHub.
4. Go into the repository settings and set the GH Pages section to publish from the `docs/` directory.

The project should now be up and available to the public through GH Pages.

Stretch Goals

There are many more fun things we can do with this project. The `README.md` file lists several possible stretch goals:

- Create a similar application using a different API. (Find suggested APIs for experimenting with here.)
- Add a second API to this application and mingle the results in some interesting way
- Use the "score" data from the API to modify the visual presentation of the words to indicate which ones are the best matches for the user's search parameters
- Use the data returned from the API to modify the appearance or interface in some other way

There are many other ways we could push this forward. Feel free to explore and experiment with the many other features of the Datamuse API and all the things we can build with it.

Application Architecture

As we become more savvy in building applications, it is possible for us to think in more advanced ways about how we are approaching the organization and management of the pieces of the application. This is something that always grows organically, no matter how experienced we become.

In any project, we begin with a rough idea of what we hope to achieve, and we often don't have much more than that to go on. As we attempt to build solutions to the problems we are working to solve, we often discover much more knowledge about the problem, how best to approach it, and what it will take to deliver working software. From these improved understandings comes ideas of how to change and improve our application.

Within the world of software design, "refactoring" is the process of making structural improvements to existing code without changing the function of the code (Wikipedia). The goal of refactoring code is to make the current features easier to maintain and easier to improve. It's important to note that the goal of refactoring is really not to directly improve the performance or quality of features in the software. Rather, refactoring should make it easier to implement performance or quality enhancements. Refactoring is often required before making those kinds of improvements to software due to the nature of writing and revising code.

Refactoring can be a necessary task when we see potential to improve our overall application architecture (our software design). As our experience with a project or problem space grows, we will evolve our approaches to building software. These improvements require us to change the way we write code. Sometimes we will reorganize information, sometimes we will use a different pattern to implement the same feature, and often we will use some combination of those techniques to make planned changes in the software possible.

Understanding what changes to make is a blend of adhering to some general principles of application architecture and software design, as well as bringing new knowledge of the specific problem domain and what the specific project requires. We can always find ways to improve our initial approaches in order to improve our ability to collaborate with others, grow new features, meet new requirements, and evolve alongside the web ecosystem.

Refactoring Goals

In this section we will explore techniques and methods for refactoring software in general, and Vue.js projects in detail. We will review some core concepts for software refactoring and improving the architecture of our applications. We will also look at the concepts and patterns Vue.js utilizes in order to better understand how to work within our chosen framework. Finally, we will practice with a complete Vue.js application that can be improved in several ways.

Core Concept: Separation of Concerns

Much of our refactoring work will boil down to upholding the "separation of concerns"—the belief that software should be organized into distinct components, each of which handles a single "concern". The goal is to create a "single source" or "authority" on any given feature or data point in our application. This allows us to more easily understand how the application works, and it allows us to isolate changes, which helps prevent us from introducing unexpected bugs in the system.

When we first begin building a project, we cannot be totally sure of what things we will build: We have an idea of the main features we need to build and how we want to build them. But along the way we discover all sorts of smaller elements we must build, configurations that must be available to multiple parts of the application, and data that must be properly stored and managed within the system.

The more time we spend building on a project, the more we realize opportunities to improve the structure of our application. We might realize that we can regroup components, re-label methods, functions, and variables, or re-organize our files so that everything just makes more sense and works better together. By making sure to separate our concerns, we create single locations for specific functionality or data to live within the system.

Imagine that we have a system that allows us to create blog posts. This might involve managing users of the system, the content of the posts, the media used in posts (such as images, videos, etc.), and much more. Each of these big areas would break apart into major components: users, blog posts, media. Within each of those major components, we can often break things down even further to create subdivisions until will have a nicely organized set of features and data that is organized in a sensible way.

For example, if we wanted to add a new piece of metadata to a user's profile, we would intuitively start looking at the user management components in our software. It would be unexpected to find a user's profile stored as part of the blog post management components. Similarly, it would be more difficult to maintain consistent data or functionality in the system if information about the user were duplicated within the blog post management components.

Upholding the separation of concerns in our applications earns us several benefits: We can more easily maintain our application because any changes can be isolated to the specific components they affect. We can more easily enhance our project because we know exactly which components are responsible for which functions and we can, again, minimize the opportunity to create unexpected errors in the system. We can also more easily bring on additional collaborators to work with us on projects because our application has a better, more straightforward organization.

Of course, we have been thinking about this topic in very abstract terms so far. Let's take a look at some specific things we can do to enhance the separation of concerns within our applications.

Re-Grouping Components or Data

When we begin developing a piece of software, we care most about getting things working. We will use one-off variables, write convoluted conditionals, line after line of logical statements, and generally bend over backwards to get things going. That is absolutely the correct approach when we begin a project and we should not hesitate to prioritize progress over elegance. However, at some point we accumulate a lot of messy code, and this is when refactoring can help.

One thing we often notice when looking at the code we wrote to get a project up and running is how much we repeat ourselves and how often we create data as separate objects that would be better connected. When we notice these tendencies in our code, that is a great indicator that we have an opportunity to re-group, better organize, and possibly create more meaningful abstractions in our code.

Consolidate Data

It's common to begin with a specific variable named for each data point we use in our code. We might use `username`, `userEmail`, `userFirstName`, and `userLastName` as individual variables when we first create our project. As we evolve our code, it often makes sense to define each of these as properties of a larger object:

```
let user = {
    'username': 'jdoe',
    'email': 'jdoe@example.com',
    'firstName': 'Jane',
    'lastName': 'Doe'
}
```

In this revised data structure it's easier to see all of the different properties of the `user` that are required in the system. When referring to data about the user, we can know that all the information is contained in this single object.

This process of moving data or functionality to more structured objects is often known as "abstraction". Abstraction allows us to create a "model" or "template" of how the data and functions for a given object or concept relate. In this case, we have "abstracted" all of the user data into the `user` object, which now provides a better way to group and show that these data points are all related. Anytime we have the `user` object, we can easily discover where to find all of the data about the user, and that data can be easily used as a package.

During the process of refactoring, we will often perform this consolidation and abstraction of data and features several times. When applied to the functionality of the system, abstraction often results in new or altered logical structures, such as functions, classes, and methods.

Convert Repetitive Logic into Functions/Methods

Sometimes we find ourselves writing the same lines of code in multiple places within our application. Even when this is only a few lines of code, it can add up dramatically over time. This is problematic because we spend time writing code that is really not needed, and we also create the potential to make more significant mistakes when we change something later.

As an example, imagine that we have a system that outputs dates in several locations. We might define a couple of constants at the beginning of our app to store the month and day names, but then we would use them wherever we need to output a date:

```
const months = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October'
, 'November', 'December'];
const weekdays = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday'];
```

Wherever we want to output a date, we might have these lines repeated:

```
let weekday = date.getDay();
let daynum = date.getDate();
let month = date.getMonth();
let year = date.getFullYear();

let fullDate = `${ weekdays[weekday] } ${ months[month] } ${ daynum }, ${ year }`;
```

This could work, and there's no shame in writing this when first starting on a project, but eventually we want to clean up the repetition and create a single, authoritative mechanism for formatting dates in our application. To this end, we could create a helpful function like this:

```
function formatDate(date){
```

```
    const months = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'Oct
ober', 'November', 'December'];
    const weekdays = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday'];

    let weekday = date.getDay();
    let daynum = date.getDate();
    let month = date.getMonth();
    let year = date.getFullYear();

    return `${ weekdays[weekday] } ${ months[month] } ${ daynum }, ${ year }`;
}
```

In this example, we have isolated all of the logic and data used to format a date within the `formatDate` function. This function will execute the same way each time we call it, so we can replace the several lines of formatting the date with a single line:

```
let fullDate = formatDate(date);
```

This saves us from repetitive typing, and it also gathers everything we would use to format dates. There is no chance that if we change the way we want our dates formatted we will miss one case that results in a bad experience for the user. This also opens the door for making more complex date formats in the future because we can focus on writing the best logic for date formatting in this single location within our system.

If we are working in an object oriented environment, we should also consider whether we have defined the proper class objects. Classes are templates for relating data and functionality together. We often create classes to model objects within the system, and we use the methods on the class to perform specific functions. If we're working with classes, then our refactoring will more likely lead to changes in which methods we specify than global functions. Nonetheless, the same concepts of grouping and re-organizing our logic still apply.

Breaking Apart Logic

Although we have mostly discussed combining logic or data objects to create self-contained data objects or functions, sometimes it is just as important to break apart logic or separate data objects. It becomes much more difficult to understand very long sequences of logical instructions. When reading a lengthy blocks of code, it is easy to lose track of all the names and relationships being created.

It is preferred to break sequences of logical instructions into their smallest pieces. It's easy to read a function or method with 5-15 lines of code. Beyond that, things become less clear and it's much more difficult to actually parse each line as a reader. However, we often make methods or functions that start out as high-level instructions and refer to more and more detailed methods or functions.

For example, it is possible to imagine writing a game where we need to initialize a game board for the players. If we imagine we have a `game` class defined, we might begin with a lengthy `setUpGame` method:

```
setUpGame: function(){
    // Clear existing board/content
    // Reset score
    // Gather player input for player names & tokens
    // Create a new board
    // Place pieces on board
    // Randomly select which player goes first
    // Initiate first move
}
```

In the example above, we see a list of comments representing each of the major tasks that needs to be completed. But for each of these tasks there might be several lines of code required to accomplish the goal. This method could easily balloon to 50 or more lines of code, which would make it difficult to read through and understand.

Rather than writing each line of code, it would be better to write a series of statements that call other methods to perform each specific task:

```
setUpGame: function(){
    // Clear existing board/content
    this.clearBoard();
    // Reset score
    this.resetScore();
    // Gather player input for player names & tokens
    this.setUpPlayers();
    // Create a new board
    this.createBoard();
    // Place pieces on board
    this.placePieces();
    // Randomly select which player goes first
    this.coinToss();
    // Initiate first move
    this.firstMove();
}
```

In this updated version, we can see that this method has become a list of other methods that must be invoked. In each one of these other methods, there can be some combination of specific instructions and calls to other methods. A developer might need to trace functionality through several other components to really get the full sense of the application.

It might seem difficult and unnecessary to separate the actual lines of code that do the work out of this method. However, there are some clear advantages. We've already discussed the fact that reading lengthy sequences of code is difficult. But there are other advantages, too.

When we move functionality out into discrete components, we can more easily modify the application. In this case, if we altered the way the score worked we would only need to update those methods directly responsible for managing scores. Likewise, if we altered the way boards are drawn (perhaps to allow user-generated game boards), then we would only need to modify lines of code in the `createBoard` method. This allows us to make those changes with minimal risk of inadvertently altering the way the `setUpGame` method works. If we find a bug in the coin toss, we do not need to wonder if it is caused by code surrounding it in the `setUpGame` method; instead, we can focus our investigation in the `coinToss` method.

We also create a modularity that allows the use of features in all the situations where we need to use them. For example, there may be other times when we need to perform a virtual coin toss. If we have separated that logic out into a standalone method, then we can use the coin toss feature whenever we wish. The ability to combine game logic in unique and interesting ways is a hallmark of what makes software games engaging.

Encapsulating Logic or Data

We might remember from previous experience with object oriented programming that "encapsulation" is often described as "information hiding." This seems like an odd concept: Why would we want to hide information from developers or users? But the reason this is done is to create a better "interface." All of our logic and data constitutes the interface of our system, and as developers we are often more concerned with the interface a system presents than the actual inner workings of the code.

There is an old analogy of driving an automobile: We can get into a car and learn to use the steering wheel, gas, brake, gear shift, etc. We do not need to be able to build a vehicle of our own, and we do not need to know exactly how our car works. Understanding the interface allows us to gain the value of using a car while the inner workings of the vehicle remain hidden from us.

The information hiding concept extends to managing how we access the deep functions of a system. Encapsulation allows us to control access to data properties or features that might not be "safe" to access directly. This often is used to allow developers to provide validation checks to make sure nothing is going to break the system.

In the vehicle analogy, this would be like systems to automatically slow us down or sound an alarm when an obstacle is detected. Although it feels like pushing the gas is directly engaging the vehicle to go forward, this input is actually being interpreted and applied through a buffer of features that are designed to prevent us from making a catastrophic mistake.

In the coding world, this is often much more mundane. We use "getters" and "setters" to allow a developer to access and save data with the benefit of data validation, clean-up, and coordination. We also use private methods and properties to make features work "behind the scenes" to provide a better developer and/or user experience.

Here is an example using a `Course` class that might exist in a learning management system:

```
class Course {
    constructor (subjectCode, number, name, description) {
        this.subjectCode = subjectCode;
        this.number = number;
        this.name = name;
        this.description = description;
    }
    get title () {
        return `${ this.subjectCode } ${ this.number }&mdash;${ this.name }`;
    }
}
```

Given this class definition for a `Course`, we could use it in this way:

```
let myCourse = new Course('WATS', 4000, 'Building JS Web Apps', 'Some longer description');
console.log( myCourse.title );
// Prints in console: WATS 4000: Building JS Web Apps
```

In this case, rather than asking developers to constantly string together the subject code, course number, and course name to create the full display title, this functionality is provided as a getter and the developer is able to use the `Course` object without risking as many errors. This is a somewhat mundane example, but additional logic can be provided on getters and setters to enforce data validation rules, process requirements, and other necessary logic.

Avoiding Complications

There is a difference between "complex" and "complicated." Many of the systems we deal with when creating software become _complex_ because they require the coordination of many data points or the use of very specific software tools or design patterns. We live in a complex world, so making software that is useful to us will necessarily be complex. Whenever we must deal with many details or components, the task will be complex.

Complication, however, arises from interactions with things that are non-essential (even if they are unavoidable). When recovering from a medical procedure, for example, we talk about "complications" as thing that will make it more difficult to achieve the goal of healing. Complications are things that prevent us from proceeding as we intend, and we generally hope to avoid them. If we find it difficult to explain the changes we've made then we might have gone too far in revising the structure of our application.

Concerning Separations

It's important to remember that we do not so much "create" abstractions or separations of concerns in our applications. Rather, we recognize when we have opportunities to enhance the separation of concerns in our software, or to enhance our system by introducing an abstraction. Any of the techniques mentioned above can be used to do harm to a project as well as doing good. The process of refactoring is not without its own risks. By thinking rationally about our projects and always keeping vigilant for ways to improve the organization of our logic and data we can move towards continuous enhancements.

Tips for Refactoring

In the previous section we discussed techniques for refactoring that are specific to enhancing the separation of concerns in our applications. Those are big concepts and worthy of the thought and effort it takes to implement them. But these are not the only ways to improve our code through refactoring.

Keep in mind that refactoring is not about achieving performance enhancements or new features. It is about leaving the code more understandable to developers. Refactoring is all about improving the structures of our code so it is easier to maintain and enhance. This is a process of revision, and there are many ways to improve code that do not require as much conceptual thought as untangling logic and data to achieve a good separation of concerns.

Improve Naming

Probably the most impactful improvement we can make on our code is to improve names where we have opportunities. We can improve the names of data objects, properties, functions, methods, classes, and anything else we have created and defined in our code. High quality names are beneficial to everyone who comes in contact with the work. They make it easier to understand what does what and how pieces go together.

A good metaphor can help guide understanding and even enhancement of a system, providing a shorthand for bigger concepts like relationships and hierarchies. But over time our software metaphors tend to change. Be sure to update naming when metaphors no longer apply. Allowing legacy naming patterns to stick around when they are no longer used creates confusion and leads to mistakes.

Balancing specific, clear names and all of the textual labels required to make software can be challenging. Updating names when concepts or business requirements change is important, and choosing high quality names is essential. Here are some tips for improving naming in our applications:

- Keep names short, but avoid shorthand names (better to be verbose than confused about which abbreviation is preferred)
- Name collections (arrays) with plural names to indicate the set of items
- Name data objects with nouns relevant to the properties they contain
- Avoid repeating parts of a name (e.g. `postTitle`, `postSummary`, `postBody`, `postAuthor`, etc.)
- Avoid overly specific names incorporating brand names, developer initials, etc.
- Use specific names that correspond to concepts/labels used elsewhere in the business
- Handle capitalization and formatting of names according to whatever style guide rules the codebase and do not deviate from those stylistic directives
- Single-letter variable names should never be referenced over more than two or three lines (preferably only in single-line statements) and should be well-scoped

There are many more considerations when naming things in our systems, but leaning on the advice above should at least help us move in the right direction with our renaming efforts.

Clean Up Files

Do not leave extra files in the repository. If we need to use things like scripts or generate data output files, they should either be kept in a specific, well-defined location in the project, or they should be removed after they have been used. No developers on the project should leave test data or scripts in unexpected locations.

Every file in the system demands attention from developers. If we work on projects with unused files strewn about, we lose focus on which files actually matter. We may find ourselves wasting time improving something that is not used, be distracted by outdated content, or otherwise be negatively impacted by vestigial files. These files often confuse new developers on the project because they increase the "noise" level in the repository.

Remove Commented Code and Failed Experiments

Along the same lines as the advice to clean up extraneous files, it's important to let go of commented-out code and failed experiments. Developers often cling to versions that were almost working, or previous attempts they made to make something work. Sometimes we refactor a component, but then we leave the old code commented-out in the file like some monument to the work that came before. Unfortunately, nobody will ever come to see these monuments, and they can only do harm to the project.

Just like with extra files in our repository, having old code lingering in files is dangerous in several ways. First, it creates the potential for confusion on the part of developers. They may become distracted by the contents of the old code, or they may misunderstand and interpret the old code as guidance for something they should be doing. It's even possible to simply goof on a text edit and un-comment lines of old code, thereby causing an error in the software.

If we have trouble letting go of old code in our projects, remember that we can always go back in the history of the project and review what used to be in those files. As long as we're using a version control system, we can easily see the old code whenever we need. This should make us feel fine about deleting all that commented-out code.

Improve Documentation

An improvement that many people overlook when refactoring is improving documentation. This is, technically, not altering the "structure" of the code, but it can definitely improve the efficiency of developers working on the project. Documentation applies to both longer instructional documents (e.g. "How to get the build deployed" or "How to work with our major feature") and to line comments in the code itself.

Keep line comments relevant to the current state of the code. When changing functionality, be sure to update the comments, too. Do not leave any erroneous or irrelevant comments in the codebase because bad comments create the same kinds of issues that are created by extra files and legacy code: They cause confusion and increase the chance of developers making mistakes.

It is possible to go overboard on comments. We do not want to write so many comments in our code that it becomes difficult to read and follow the logic. We also do not need to comment on details that are totally evident to a regular developer: We can assume any developer coming into a project understands the basics of the language and platform. We should focus our comments on the elements that make our software unique and which will aid the understanding of the "new" developer. (Don't forget that in six months when we return to this code we, ourselves, will be the "new" developer!)

Here are some tips for writing comments in our code:

- Do not leave old `TODO` comments laying around after completing the task (use the "find in files" or similar tool in an editor to help find comments across all files)
- Do not leave outdated comments that describe functionality that has changed or in some other way become erroneous
- Focus on describing "why" something is happening in the code; don't just describe "what" is happening
- Be consistent in style and formatting of comments
- Describe Classes and major functions/methods with a descriptive block comment at the beginning of the definition
- Use inline comments to describe variables or values without taking up too much space

- If very long documentation is required, move it to another location and reference that location in a shorter comment
- Comment on any unique functionality
- Comment on any implementation that is required due to the use of another tool (e.g. "This configuration is used by XYZ module to accomplish ABC goal.")
- Avoid putting personal information in comments (nobody needs to see your name, rank, serial number, or anything else in the comments)
- Comments are not an art project; consider other developers and the ease of reading comments while working on code

Spending a little extra time to improve comments in our code is something we should do each time we make an edit. If we pay attention to the tips above, we can provide a better experience for everyone working on the project.

Composing Components

Vue.js is organized around the Web Components Standard, which is an idea that has gained momentum over the past few years. The idea of web components has been popularized primarily by another single page application framework, React. React and Vue.js share a many common approaches and concepts, most of which stem from their mutual interest in web components.

The idea of web components is straightforward and powerful: Allow developers to create reusable, custom HTML elements that handle whatever tasks are needed in an application or site. By creating small, specific components, we can use several components together to achieve a more complex interface or experience. In the jargon of Vue.js, this process of combining components to build our interfaces is known as "composing components."

In Vue.js we write components that handle specific parts of our application. Most of these components should be small and dedicated completely to their role in the application: A "favorite" button, a search result, or the content of an article. Other components might handle larger pieces of the application, such as a component to control an entire "About Us" view, or a component that would build up a gallery of images or video.

Those larger components will most likely reference smaller components within their templates. This allows us to create more complex interfaces, but we can still keep our individual components as lean as possible. In each component, we want to keep our template, logic, and stylesheet as small as possible. Fewer lines makes our files more readable, more easily and immediately understandable, and eliminates many opportunities for error.

The process of determining how we should organize our applications is similar to the methods for refactoring that we discussed earlier: We want to create a solid separation of concerns, and that can be achieved by encapsulating functionality into specific components. We can also use our choices about what components we create to provide a more friendly interface for developers of our application.

Because all of this can be difficult to imagine in the abstract, let's explore some concrete examples of how we use components together within a Vue.js project.

Using Child Components in Templates

In any given component template, we can reference "child" components. These child components are revealed to the template as a custom HTML element, and they are defined in the `components` property. Borrowing inspiration from the Vue.js documentation about Components, here is an example of how we could implement a simple `<counter>` element as a child component:

The `Home.vue` file

```
<template>
  <div class="home">
    <h1>Child Components Demo</h1>
    <counter></counter> | <counter></counter> | <counter></counter>
  </div>
</template>

<script>
import Counter from '@/components/Counter';

export default {
  name: 'Home',
  data () {
    return {

    }
```

```
  },
  components: {
    counter: Counter
  }
}
</script>
```

The `Counter.vue` file

```
<template>
  <button class="fave" v-on:click="incrementCounter">
    {{ counter }}
  </button>
</template>

<script>
export default {
  name: 'favorite',
  data () {
    return {
      counter: 0
    }
  },
  methods: {
    incrementCounter: function () {
      this.counter++;
    }
  }
}
</script>
```

We can see that in these two files we have two components. The `Counter.vue` file defines a component with a very simple template: It is just a `<button>`. It uses a single data value (`counter`), and it has an event listener defined to increment the counter on each click using the `incrementCounter` method. This component could be used on its own at a specific URL route, but it would not be a very interesting view. Instead, we have defined this component with the intention of using it in another component.

In the `Home.vue` file, we can see that there is a new property in the component logic called `components`. We can also see that the `Counter` component has been imported at the beginning of the logic. It is required to import each child component before using it in our component logic. The `components` property is an object, and each property of the `components` object defines a relationship between a child component and the name that will be used for the custom HTML element in the component template.

In the template of the `Home` component, we can see that we have the child component tag (`<counter></counter>`) used three times. Each of these uses will result in a button that we can click to increment the number it displays. When we view the `Home` component in the web browser, we can click the buttons and see a result that looks like this:

Counter Buttons

As we can see in the image above, when we click each button it keeps its own tally using its own counter. Although these buttons are each a *copy* of the `Counter` component, they are individual instances. The ability to isolate functionality and data into a child component can be useful in itself, but it's often necessary to pass some information to the child component in order to do something with it.

Setting Properties on Child Components

Components can define a property called `props` (properties), which is an object listing the names (and, possibly, additional details) of any values that should be passed into the component. These properties can be referenced in the templates using the normal mustache syntax (`{{ propName }}`) or within the component logic like normal (`this.propName`). Properties passed to a child component are bound to the parent component logic, so if the value of the property changes in the parent component, that will refresh the value within the child component, too.

This is very useful for writing child components to handle things like show/hide of individual items. Here is an example using an imaginary FAQ page:

Parent Component: `FAQ.vue`

```
<template>
  <div>
    <h1>FAQ</h1>
    <ul class="faqs">
      <li v-for="faq in faqs">
        <question v-bind:question="faq.question" v-bind:answer="faq.answer"></question>
      </li>
    </ul>
  </div>
</template>

<script>
import Question from '@/components/Question';

export default {
  name: 'FAQ',
  data () {
    return {
      faqs: [
        {
          question: 'Why does the sun shine?',
          answer: 'The sun is a miasma of incandescent plasma.'
        },
        {
          question: 'What is the meaning of life?',
          answer: '42.'
        },
        {
          question: 'How to become a web developer?',
          answer: "Never gonna give you up, never gonna let you down"
        },

      ]
    }
  },
  components: {
    question: Question
  }
}
</script>
```

Notice that this parent component controls the "Frequently Asked Questions" view. There is an array called `faqs` that contains the questions and answer we wish to show the user. We want to have these listed by questions, and then reveal the answer when the user clicks on the question. If we were to try to do this directly inside the `FAQ` component, we would need to coordinate a lot of information and add a data property to each of the question/answer sets in order to know which one should be shown or hidden when a user clicks. This logic would get overly complex and it is not necessary.

Instead, we handle this component like we would handle any content presentation to the user, except we use a child component called `Question` to display the content. The `Question` component is imported at the top of the `FAQ` component logic, and then the `Question` component is listed under the `components` property.

We can see that in the template for the `FAQ` component, we loop through each `faq` in the `faqs` array. Within the loop, we insert a `<question>` element and we bind the `question` and `answer` attributes to their corresponding values `faq.question` and `faq.answer`. These attributes will be translated into the properties that the `Question` component expects.

Let's take a look at the child component to see how these are used to display the content to the user.

Child Component: `Question.vue`

```
<template>
  <div class="question">
    <h2><a v-on:click="toggleAnswer">{{ question }}</a></h2>
    <p v-show="showAnswer" class="answer">{{ answer }}</p>
  </div>
</template>

<script>
export default {
  name: 'FAQ',
  data () {
    return {
      showAnswer: false
    }
  },
  props: {
    question: String,
    answer: String
  },
  methods: {
    toggleAnswer: function () {
      this.showAnswer = !this.showAnswer;
    }
  }
}
</script>
```

In the `Question` component, we have a simple template defined: We output the `question` and `answer` properties in the template. In the component logic we define those two values in the `props` object, and we also define a simple `toggleAnswer` method that toggles the `showAnswer` value between `false` and `true`. The `showAnswer` value is used to control the `v-show` directive on the answer content.

Here is an example of what this looks like in action:

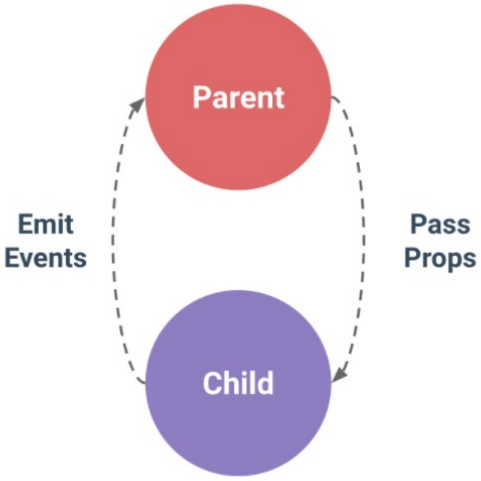
FAQ Example in Action

In this case, what would have taken several extra lines of JavaScript, and a significant increase in complexity, without composing components has been accomplished in a much more straightforward way. It's easy to understand how the components work together when reading through the code, and we have successfully encapsulated the functionality of the show/hide answers inside the `Question` component. We could even use the `Question` component outside the scope of the FAQ page if we had the need. Essentially we have created a component that will accept content of a certain structure and handle displaying it according to our rules. This is a modular piece of our application that could be used anywhere.

Using Events and Listeners

The data flow in Vue.js is "one-way," which means that data is only allowed to be passed from parent component to child component. This works great for child components that are purely display-oriented, but we often need to trigger some kind of action in the parent component when an action takes place in the child element.

In this situation, we can use custom events to trigger the action in the parent component. This is how the Vue.js information flow was designed to work:



Vue.js information flow

As we can see from the diagram, information is put into a child component via "properties" and the child component can signal back to the parent component when an event has occurred. Let's look at a simplified catalog and shopping cart setup to get a clearer picture of how this works.

Parent Component: `Catalog.vue`

```
<template>
  <div>
    <h1>Store</h1>
    <p>Number of items in your cart: {{ numItems }}</p>
    <ul class="items">
      <li v-for="item in catalog">
        <item v-on:addedItem="incrementItemCount" v-bind:name="item.name" v-bind:price="item.price"></item>
      </li>
    </ul>
  </div>
</template>

<script>
import Item from '@/components/Item';

export default {
  name: 'FAQ',
  data () {
    return {
      catalog: [
        {
          name: 'Widget',
          price: '12.43'
        },
        {
          name: 'Gidget',
          price: '74.98'
        },
        {
          name: 'Fidget',
          price: '3.47'
        },
```

```
      ],
      numItems: 0
    }
  },
  components: {
    item: Item
  },
  methods: {
    incrementItemCount: function () {
      this.numItems++;
    }
  }
}
</script>
```

We can see that in the `Catalog` component, we have a `catalog` array defined and a `numItems` value. The `catalog` array provides the items for us to loop through and display. The `numItems` value shows how many items we have already added to our shopping cart. We can also see that the `Catalog` component uses the `Item` component in its template. This is defined in the `components` property, and it corresponds to the `<item>` element used in the template. Finally, there is an `incrementItemCount` method that is used to increase the item count whenever a new item is added to the shopping cart.

The one thing that really makes this example different than the previous one is that we have added an event listener to that custom `<item>` element:

```
v-on:addedItem="incrementItemCount"
```

That listener is waiting for the `addedItem` event to be triggered. When an `addedItem` event is detected, the `Catalog` component will execute the `incrementItemCount` method, which will increase the value of `numItems` and update the display for the user. Let's take a look at the `Item` component to see how that works.

Child Component: `Item.vue`

```
<template>
  <div class="item">
    <h2>{{ name }}</h2>
    <p>Price: ${{ price }}</p>
    <button v-on:click="addToShoppingCart">Add to Cart</button>
  </div>
</template>

<script>

export default {
  name: 'item',
  data () {
    return {

    }
  },
  props: {
    name: String,
    price: Number
  },
  methods: {
    addToShoppingCart: function () {
      console.log(`Adding ${ this.name } to cart.`);
      this.$emit('addedItem');
    }
  }
}
</script>
```

In the `Item` component, we have a simple template defined that shows some information and provides an "Add to cart" button. The button has a `v-on` directive applied to it, and it's looking for a `click` event. When the user clicks the button, it executes the `addToShoppingCart` function. In a real-world implementation, this method would probably perform an API request to record the user's data in a persistent database. For the sake of this example, it merely logs a message in the browser console. It also emits a custom event trigger, `addedItem`.

When all of this is put together, the result is a working counter that increments regardless of which item we add to the shopping cart:

Custom Event Example

The parent `Catalog` component is managing the display of the number of items the user has added to their shopping cart. Each instance of the `Item` child component is only concerned with watching itself and triggering the `addedItem` event when the user clicks the "add to cart" button. Using the one-way data flow in Vue.js, we have been able to send data to the child component and then receive a signal back from the child component. In a real-world example, that event would probably trigger the parent component to refresh an API call or perform some other action to update the information about the shopping cart being shown to the user.

By creatively combining these two abilities, there are very few cases that cannot be handled. (And when those rare cases arise, Vue.js does provide some ways for us to create exceptions to these rules.) Following solid principles of composing components makes our Vue.js applications easier to build, easier to maintain, and easier to improve.

Organizing Data and Configuration Code

As we work on different parts of our applications, we often find that we have similar needs in different places. We might have specific formatting we want to remain consistent across templates, or system data that needs to be available to different components. We might find that we are making API requests to the same API in different components and we want to minimize the amount of repetitive code we are using. In each of these cases, what we absolutely want to avoid is duplicating the same data or logic in multiple locations.

As we saw in previous sections, we will make our applications easier to maintain, build, and enhance if we can adhere to a clean separation of concerns and utilize some common strategies for managing components that need to be reused throughout the application. In this section we will look at some techniques for organizing reusable pieces of our system.

ES6 Modules

The Vue CLI setup we have been using supports ECMAScript 6 (ES6) and uses a tool called Babel to provide some level of backwards compatibility for browsers that do not yet support ES6 features. Luckily, most of the browser market and related technologies now fully support ES6 features, which opens the door for us to improve how we write code. (For developers who are unaware, ECMAScript is the standard that governs JavaScript so the two names should be synonymous, although they often are not.)

One powerful feature of ES6 that we will make extensive use of in this section is the creation, export, and import of "modules." JavaScript has had a notion of modules for awhile now: It was first created using external libraries like RequireJS and AMD, and then it made its way to ES5. But the latest version of modules in JavaScript is a major improvement and allows for techniques like those described in this section.

We can learn more about using modules to export and import components in our software on the Mozilla Developers Network pages for Import and Export, respectively. There are many other great resources to explore online, and more will be created in the near future, too. Seize the power of this new modules system to make our apps as powerful as possible.

Common Data Techniques

A common case that often comes up is the need to store system-specific data that can be accessed from any component. This is often used for sets of static metadata that are specific to the project and which are not subject to change. These are usually considered "constants" in the system: They are not meant to be altered by the user or to change during the use of the application or site.

Since we can easily import objects into our components, all we need to do is make sure we have properly structured our data file to export an object. We can store these files wherever makes sense: a `/common/` directory in our `/src/` directory, a well-named file, or somewhere else within the project.

Here is an example of a data file that can be imported into a Vue.js component. Assume this code is in the file `/src/common/constants.js`.

```
export default {
```

```
    'metadataProperty': 'Some value',
    'someChoices': [
        { name: 'Choice One', value: 1 },
        { name: 'Choice Two', value: 2 },
        { name: 'Choice Three', value: 3 }
    ]
}
```

We could make use of this data object by importing it into a Vue.js component:

```
<script>
import SystemData from `@/common/constants.js`;

export default {
  name: 'demoComponent',
  data () {
    return {
      formOptions: SystemData.someChoices
    }
  }
}
</script>
```

As we've seen in previous projects and examples, we can import the content of `constants.js` with the name `SystemData` inside our component. We can then use that data however we would like within our component logic. In this case, we have set the component's `formOptions` value to the `SystemData.someChoices` array. This could be used to provide a form input with consistent choices and formatting across the entire application.

This technique also shows the fundamental principle at play in the following examples. First we create a `.js` file that contains some object. We must make sure that object is properly "exported" with the `export` command. Then, we can import that object wherever we need it. This is fundamentally the same process we use to accomplish the other techniques for organizing information in our Vue.js applications.

Common Configuration Techniques

Previously in this book we explored using the Axios module to perform HTTP requests to API endpoints. This is a powerful tool, and on a modern web project we might use several different API services that are all controlled by our backend systems. It is very common for developers to consume their own API services to build multiple frontends (e.g. mobile app and website).

When using multiple endpoints on the same API service provider, it is often necessary to provide basic authentication information, to complete some sort of authentication handshake, or to otherwise use a common configuration. If we are making API calls from multiple components, we might find that we've duplicated this common data and logic several times throughout our application. We can refactor those API calls to use the same base instance.

This approach can work in many situations with JavaScript modules that rely on some form of common configuration. Let's take a look at an Axios example to get a better idea of what this looks like. The following code is stored in the file `/src/common/api.js`.

```
import axios from 'axios';

export const API = axios.create({
  baseURL: `http://api.openweathermap.org/data/2.5/`
})
API.interceptors.request.use(function (config) {
    // Set APPID on each request
    config.params.APPID = 'YOUR API KEY HERE';
    return config;
```

```
  }, function (error) {
    // Do something with request error
    return Promise.reject(error);
  });
```

In this example, we are using the Open Weather Map API, which requires an API Key (called `APPID` in the querystring parameters). Obviously, we would prefer not to repeat this configuration in every single `.vue` file where we make a call to a different API endpoint. There are several endpoints (`find` , `weather` , `forecast` , etc.) and if we were building a full app we would use each of those endpoints.

Now that we have our basic API configuration abstracted into a standalone file, we can use it in another component. Let's imagine we have a component called `CitySearch.vue` that allows users to look up the weather summary for a city:

```
<script>
import {API} from '@/common/api.js';

export default {
  name: 'CitySearch',
  data () {
    return {
      results: null,
      errors: [],
      query: ''
    }
  },
  methods: {
    getCities: function () {
      API.get('find', {
        params: {
            q: this.query,
            units: 'imperial'
        }
      })
      .then(response => {
        this.results = response.data
      })
      .catch(error => {
        this.errors.push(error)
      });
    }
  }
}
</script>
```

Notice that in this example we don't need to specify anything beyond the endpoint path (`weather`). Our API call is much smaller than in previous examples. And if we used a different endpoint to get the forecast for a post, the API call would look something like this:

```
<script>
import {API} from '@/common/api.js';

export default {
  name: 'Forecast',
  data () {
    return {
      weatherData: null,
      errors: [],
      query: ''
    }
  },
  created () {
    API.get('forecast', {
```

```
      params: {
          id: this.$route.params.cityId,
          units: 'imperial'
      }
    })
    .then(response => {
      this.weatherData = response.data
    })
    .catch(error => {
      this.errors.push(error)
    });
  }
}
</script>
```

In this example, we have a component that expects to receive a `cityId` parameter as part of the URL. This component is going to make a request to get the forecast for a given city and display the results. The API request is once again formed by importing the `API` object, and the URL for the endpoint is `forecast`. Some query string parameters are added to the request (setting the `id` value), but otherwise the request looks the same as the previous request. And once again we have not duplicated the basic configuration information.

Not only is this a cleaner way of using the the same API service in multiple components, but it also opens the door for us to provide a mechanism to switch between a "production" and "development" API server. Now that our configuration is abstracted into a single location, we could enhance that configuration to properly alter which API server the application should contact. This is a very common use case for developers, who must often work with new functionality or data that is unavailable on the production API service.

Common Filters and Methods

When working with components, we often find ourselves performing the same tasks over and over. For example, formatting dates, text, or monetary values is a common need in our templates. This formatting is best accomplished with a "filter", which can be applied to the output of a value in the template. Since most of the logic within a component is packaged as a JavaScript object, it is easy to define objects that can be imported and used within multiple components.

Let's consider the example of a filter that capitalizes a String value. This is often needed when displaying user data in a template because we cannot be sure the user themselves capitalized the text. We can follow the same patterns we used above to accomplish this goal.

First, let's make a file called `/src/common/filters.js`:

```
export default {
  capitalize: function (value) {
    if (!value){
     return '';
    }
    value = value.toString();
    return value.charAt(0).toUpperCase() + value.slice(1);
  }
}
```

This file defines an object that has one property: `capitalize`. That property is a function, that expects an argument called `value` and returns a modified form of the `value`. (This is the standard format for a Vue.js filter.)

We can use this filter in a component by importing it in the component logic and then using it in the template.

```
<template>
  <div class="item">
```

```
    <h2>{{ name|capitalize }}</h2>
    <p>Price: ${{ price }}</p>
    <button v-on:click="addToShoppingCart">Add to Cart</button>
  </div>
</template>

<script>
import CommonFilters from '@/common/filters.js';

export default {
  name: 'item',
  data () {
    return {

    }
  },
  props: [
    'name',
    'price'
  ],
  methods: {
    addToShoppingCart: function () {
      console.log(`Adding ${ this.name } to cart.`);
      this.$emit('addedItem');
    }
  },
  filters: CommonFilters
}
</script>
```

In this component, we have used the `capitalize` filter to make sure the name of each item is capitalized. We could define additional filter functions in the `/src/common/filters.js` file, and each of those would also become available to every component that makes use of this technique. It is much easier to maintain consistent formatting and functionality by consolidating filters in this way.

This same technique can be used to consolidate other aspects of a component definition, too: If the same methods are used on multiple components, they can also be abstracted into a common file. Similar techniques can work for managing sets of `props` or other properties, too.

Each of the techniques described in this section revolve around creating common sources that can be accessed from anywhere in our application to provide consistent functionality. By organizing our application with these sorts of consolidations, we can provide a more understandable and less error-prone development environment for the team.

Quiz: Application Architecture

Please enjoy this self-check quiz to help you identify key concepts, points, and techniques discussed in this section.

Visit Quiz Online

The quiz on this page has been removed from your PDF or ebook format. You may take the quiz by visiting this book online.

Project: Application Architecture and Refactoring Practice

The Refactoring Practice project repository provides you with a working weather application that can be improved through refactoring. The weather app uses three major views: City Search, Current Weather, and 5 Day Forecast. Users can search for their city and then view weather data. The weather data is requested from the OpenWeatherMap.org API, which is free and available for any developer to use. (**NOTE:** You will need to sign up to get an OpenWeatherMap.org API Key to complete this project.)

In the initial state, this application will work (once you insert your API key into the proper locations), but the structure of the application could be significantly improved. We will practice creating child components that can accept data from the parent component, abstract base API configurations away from each individual API call, and consolidate other HTML and CSS blocks to minimize the pain of maintenance.

Review the Requirements

In order to complete this project, we must fulfill the following Basic Requirements. These requirements are all focused around refactoring the application into a more organized structure that will make future maintenance and improvements easier. Remember that we are **not** trying to improve the performance or add any new features to the project. The goal here is purely around refactoring. The end result should appear to the user exactly like the current version.

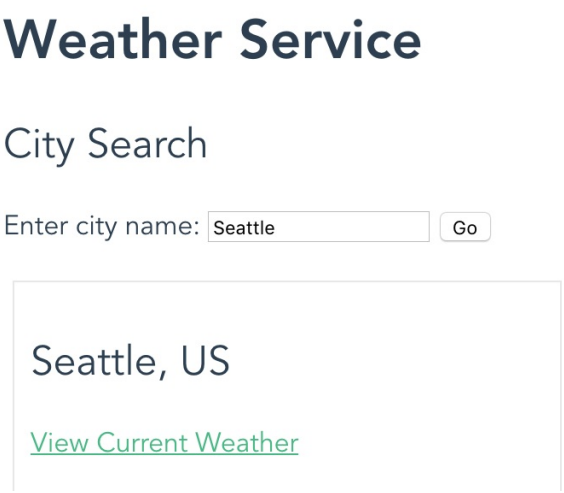
- Sign up to OpenWeatherMap.org and generate an API Key.
- Paste your API Key (which will be used as the `APPID` parameter) into the appropriate locations in the `CitySearch.vue`, `CurrentWeather.vue`, and `Forecast.vue` files.
- Verify the site works with your key. You should be able to search for a city and see weather data.
- Abstract the base configuration for the API requests to a common file to reduce duplication of the base URL and `APPID`.
- Create child components that can accept weather information and produce a well-formatted display.
- Use the child components in each of the views to eliminate the redundant HTML and CSS styles used.
- Create a child component called `ErrorList` to handle display of error messages. Replace the error message handling in the templates of the three parent components with this child component.
- Clean up any extraneous code, comments, or files that are unused.
- Add comments where they would be helpful to improve the readability of the project.

Working the Project

In order to get this project working, we should first fork the repository from the main Refactoring Practice repository, then we should clone the files to our local development area. We will need to install the project dependencies by running `npm install` from the project root directory.

Make an OpenWeatherMap.org API Key

To get the project running, we need to make an account on OpenWeatherMap.org and generate an API key. Once we have created an account, the API Keys can be found under our Account page (located here). Create a new API Key and then open the project repository in a preferred editor. We must find the `YOUR_APPID_HERE` placeholders in the `CitySearch.vue`, `CurrentWeather.vue` and `Forecast.vue` files and replace those placeholders with our actual API Key (called an `APPID` by OpenWeatherMap.org).



Working home screen

Once we've replaced that information the project should become operational. Run `npm run dev` and verify that the project works. Once we've made sure the project is working, we can begin refactoring.

Abstract the Base API Configuration

The first thing we can do to make a major improvement in this application is to consolidate the base configuration for our API into a common module that can be imported into whatever component we need. We can do this using a few basic features available in `axios`, the module we are using to handle HTTP requests to our API services.

Let's start by making a new directory under `src/` called `common/`. Then, we will create the file `src/common/api.js`. Inside that file, we will write the following code:

```
import axios from 'axios';

export const API = axios.create({
  baseURL: `//api.openweathermap.org/data/2.5/`
})
API.interceptors.request.use(function (config) {
    // Set common parameters on each request
    config.params.APPID = 'YOUR_APPID_HERE';
    config.params.units = 'imperial';
    return config;
  }, function (error) {
    return Promise.reject(error);
  });
```

The code above creates a new `const` variable called `API`. The `API` object is initialized with the `axios.create()` method, which returns an HTTP request object. We use the `export` keyword to declare the `API` value because we want to be able to import the `API` object wherever we need to use it. Notice that we set the `base_url` to `//api.openweathermap.org/data/2.5/`. We have omitted the `http:`, which is a common technique to allow the browser to fill in whatever the current protocol is. If our site is deployed on an `http` server, then it will prepend `http:`; if it is deployed on an `https` server, then the browser will prepend `https:`. This way we avoid any security warnings by using `http` or `https` at the wrong time.

After we have created the `API` object, we set up an `interceptor` on the base configuration. The `interceptor` will watch for any request that is made using the `API` object. For each request it will "intercept" the data before it is sent, and it will add two properties to the `params` object: `APPID` and `units`. Since these values were the same in each request, we can safely include them in this interceptor and we do not need to repeat them in each component where we use the `API` object.

Once we have this base configuration in place, we can update our components to make use of the new, leaner API object. We will need to update the API calls in all three of our components. Here is the first update for an example:

`CitySearch.vue`

```
<script>
import {API} from '@/common/api';

export default {
  name: 'CitySearch',
  data () {
    return {
      results: null,
      errors: [],
      query: ''
    }
  },
  methods: {
    getCities: function () {
      API.get('find', {
        params: {
            q: this.query
        }
      })
      .then(response => {
        this.results = response.data
      })
      .catch(error => {
        this.errors.push(error)
      });
    }
  }
}
</script>
```

Notice that we have updated the import statement to import the `API` object instead of `axios` directly. We have simplified the API request in the `getCities` method, too: it is now several lines shorter. It only needs to provide the unique endpoint (in this case, `find`) and the unique params (in this case the `q` param is populated with the user's `query` value).

Once we have made these changes, the component functions just as it did before. We can now make the changes in each of the remaining components and then we are done with the first portion of this refactoring project.

Create Child Components to Display Weather Information

There are two places where it appears we have serious redundancy in our templates. These two locations can be isolated into child components that will accept data from their parent. They do not need to process any data or make any additional API requests, so they should be relatively lean components.

Weather Summary

First, we will break up the `weatherSummary` section into a child component. We can see that this code is repeated in each of the three main components:

```
<div v-for="weatherSummary in city.weather" class="weatherSummary">
    <img v-bind:src="'http://openweathermap.org/img/w/' + weatherSummary.icon + '.png'" v-bind:alt="weatherSumm
ary.main">
    <br>
    <b>{{ weatherSummary.main }}</b>
</div>
```

This summary area of the data is always an array that must be iterated through. It often only has one item in the array, but sometimes it will have more. For each item in the array we want to show the icon and the summary text. We can create a new component called `WeatherSummary` that will handle this HTML code:

```
<template>
  <div>
    <div v-for="weatherSummary in weatherData" class="weatherSummary">
      <img v-bind:src="'http://openweathermap.org/img/w/' + weatherSummary.icon + '.png'" v-bind:alt="weatherSu
mmary.main">
      <br>
      <b>{{ weatherSummary.main }}</b>
    </div>
  </div>
</template>

<script>
export default {
  name: 'WeatherSummary',
  data () {
    return {

    }
  },
  props: {
    weatherData: {}
  }
}
</script>

<style scoped>
  .weatherSummary {
    display: inline-block;
    width: 100px;
  }
</style>
```

As we can see, this new child component is completely dedicated to displaying these weather summary items. It does not have any terribly complex behavior. It simply expects an array of `weatherSummary` objects, and it will process those the same way each time. We can now add this component into our main components that control our views. Here is what it looks like when used in the `CitySearch` component:

```
<template>
  <div>
    ... template code ...

    <weather-summary v-bind:weatherData="city.weather"></weather-summary>

    ... more template code ...
  </div>
</template>

<script>
import {API} from '@/common/api';
import WeatherSummary from '@/components/WeatherSummary';

export default {
  name: 'CitySearch',
  data () {
    return {
      results: null,
      errors: [],
      query: ''
    }
  },
  methods: {
```

```
    getCities: function () {
      API.get('find', {
        params: {
            q: this.query
        }
      })
      .then(response => {
        this.results = response.data
      })
      .catch(error => {
        this.errors.push(error)
      });
    }
  },
  components: {
    'weather-summary': WeatherSummary
  }
}
</script>
```

In this example, we can see that the `<weather-summary>` element is used. We have imported the `WeatherSummary` child component at the top of our component logic, and then we have defined a `components` object that indicates we will be using the `WeatherSummary` in our templates. We must use `v-bind` to pass the value of `city.weather` to the `WeatherSummary`, where it comes in as the `WeatherData` property.

Now that we have this component in place in the `CitySearch` component we can test things out and then follow the same process to add the child component to our `CurrentWeather` and `Forecast` components. Note that the name of the value we pass to the `WeatherSummary` component changes in each instance where it is used:

`CurrentWeather.vue`

```
<weather-summary v-bind:weatherData="weatherData.weather"></weather-summary>
```

`Forecast.vue`

```
<weather-summary v-bind:weatherData="forecast.weather"></weather-summary>
```

In each case, we pass in the proper `weather` array that can be processed by the `WeatherSummary` component.

Weather Data

The other portion of the component templates that is repetitive is the weather data display. This structure is repeated almost verbatim in several places:

```
<dl>
    <dt>Humidity</dt>
    <dd>{{ forecast.main.humidity }}%</dd>
    <dt>High</dt>
    <dd>{{ forecast.main.temp_max }}&deg;F</dd>
    <dt>Low</dt>
    <dd>{{ forecast.main.temp_min }}&deg;F</dd>
</dl>
```

There are also several style definitions that target this HTML, and those are also duplicated in each component. If we allowed the code to remain so repetitive we would undoubtedly see the gradual evolution of these styles and this markup diverge as small tweaks were made to different components and developers unwittingly reproduced work that had already been done.

Rather than living with this duplication, we can isolate this display into a single component that can be used whenever we need this structure. Let's call this component `WeatherConditions`. Here is a draft of what this component might look like:

```
<template>
  <div>
    <dl>
      <dt v-if="conditions.temp != conditions.temp_min">Current Temperature</dt>
      <dd v-if="conditions.temp != conditions.temp_min">{{ conditions.temp }}%</dd>
      <dt>Humidity</dt>
      <dd>{{ conditions.humidity }}%</dd>
      <dt>High</dt>
      <dd>{{ conditions.temp_max }}&deg;F</dd>
      <dt>Low</dt>
      <dd>{{ conditions.temp_min }}&deg;F</dd>
    </dl>
  </div>
</template>

<script>
export default {
  name: 'WeatherConditions',
  data () {
    return {

    }
  },
  props: {
    conditions: {}
  }
}
</script>

<style scoped>
  dl {
    padding: 5px;
    background: #e8e8e8;
  }
  dt {
    float: left;
    clear: left;
    width: 120px;
    text-align: right;
    font-weight: bold;
    color: blue;
  }
  dd {
    margin: 0 0 0 130px;
    padding: 0 0 0.5em 0;
  }
  dt::after {
    content: ":";
  }
</style>
```

This component can be passed the `main` object from any of our API responses and it will output the properly-formatted weather condition data: humidity, high temp, and low temp. This has removed almost 20 lines of CSS and 8 lines of HTML from each of the other components, and it has isolated this functionality into a single location where we could focus our work if we needed to enhance this output. We can use this child component in our other component templates just like we did with the `WeatherSummary` component (see the full file text below for an example of how to use this component inside of our other components).

Create `ErrorList` Child Component

Now that we've done this twice, separating the errors list out into a child component should be a snap. This component will expect the `errors` array as a property, and then it will process and display those errors. Again, this child component only needs to receive and display data, so it will not need to implement any custom events.

To accomplish this refactoring, we need to create a new file for the `ErrorList` component. This is what it should look like:

```
<template>
  <div>
    <div v-if="errorList.length > 0">
      <h2>There was an error fetching data.</h2>
      <ul class="errors">
        <li v-for="error in errorList">{{ error }}</li>
      </ul>
    </div>
  </div>
</template>

<script>
export default {
  name: 'ErrorList',
  data () {
    return {

    }
  },
  props: {
    errorList: []
  }
}
</script>

<style scoped>
  .errors li {
    color: red;
    border: solid red 1px;
    padding: 5px;
  }
</style>
```

This new child component can be used in each of our main components and will allow us to eliminate another dozen or so lines of code in each of those main components. It will also consolidate the error listing feature, which allows us to more easily focus on improving that feature when we choose to do so (without risking unnecessary complications in our other components).

We can implement this new child component using the same patterns we established before. See the example code below for details about how to implement this child component.

Clean Up Extraneous Files

There are several files that appear to be remnants of previous development. Clean up the extra file in the `src/components/` directory and the extra experimental data file. Take another pass through all of the code files and remove any extraneous `TODO` comments or other irrelevant comments.

Enhance Comments

If we haven't already been conscientious about commenting our changes so developers reviewing our work will be able to tell how things work, then now is the time to go back through and do that work. We must leave comments that allow developers to more quickly understand non-obvious details about the code and how to use it.

Wrapping Up

Now that we've completed the project, here are what each of our files look like in their entirety. We can reference these examples to check our own work.

Components

CitySearch.vue

```
<template>
  <div>
    <h2>City Search</h2>
    <form v-on:submit.prevent="getCities">
        <p>Enter city name: <input type="text" v-model="query" placeholder="Paris, TX"> <button type="submit">Go
</button></p>
    </form>
    <ul class="cities" v-if="results && results.list.length > 0">
        <li v-for="city in results.list">
            <h2>{{ city.name }}, {{ city.sys.country }}</h2>
            <p><router-link v-bind:to="{ name: 'CurrentWeather', params: { cityId: city.id } }">View Current We
ather</router-link></p>

            <weather-summary v-bind:weatherData="city.weather"></weather-summary>

            <weather-conditions v-bind:conditions="city.main"></weather-conditions>

        </li>
    </ul>
    <error-list v-bind:errorList="errors"></error-list>

  </div>
</template>

<script>
import {API} from '@/common/api';
import WeatherSummary from '@/components/WeatherSummary';
import WeatherConditions from '@/components/WeatherConditions';
import ErrorList from '@/components/ErrorList';

export default {
  name: 'CitySearch',
  data () {
    return {
      results: null,
      errors: [],
      query: ''
    }
  },
  methods: {
    getCities: function () {
      API.get('find', {
        params: {
            q: this.query
        }
      })
      .then(response => {
        this.results = response.data
      })
      .catch(error => {
        this.errors.push(error)
      });
    }
  },
  components: {
    'weather-summary': WeatherSummary,
    'weather-conditions': WeatherConditions,
```

```
    'error-list': ErrorList
  }
}
</script>

<style scoped>
h1, h2 {
  font-weight: normal;
}

ul {
  list-style-type: none;
  padding: 0;
}
li {
  display: inline-block;
  width: 300px;
  min-height: 300px;
  border: solid 1px #e8e8e8;
  padding: 10px;
  margin: 5px;
}
a {
  color: #42b983;
}
</style>
```

CurrentWeather.vue

```
<template>
  <div>
    <h2>Current Weather <span v-if="weatherData"> for {{ weatherData.name }}, {{weatherData.sys.country }}</span
></h2>
    <p>
      <router-link to="/">Home</router-link> |
      <router-link v-bind:to="{ name: 'Forecast', params: { cityId: $route.params.cityId } }">View 5-Day Foreca
st</router-link>
    </p>
    <div v-if="weatherData && errors.length===0">

      <weather-summary v-bind:weatherData="weatherData.weather"></weather-summary>
      <weather-conditions v-bind:conditions="weatherData.main"></weather-conditions>
    </div>
    <div v-else>
      <h2>Loading...</h2>
    </div>
    <error-list v-bind:errorList="errors"></error-list>
  </div>
</template>

<script>
import {API} from '@/common/api';
import WeatherSummary from '@/components/WeatherSummary';
import WeatherConditions from '@/components/WeatherConditions';
import ErrorList from '@/components/ErrorList';

export default {
  name: 'CurrentWeather',
  data () {
    return {
      weatherData: null,
      errors: [],
      query: ''
    }
  },
  created () {
    API.get('weather', {
```

```
      params: {
          id: this.$route.params.cityId
      }
    })
    .then(response => {
      this.weatherData = response.data
    })
    .catch(error => {
      this.errors.push(error)
    });
  },
  components: {
    'weather-summary': WeatherSummary,
    'weather-conditions': WeatherConditions,
    'error-list': ErrorList
  }
}
</script>

<style scoped>
h1, h2 {
  font-weight: normal;
}

ul {
  list-style-type: none;
  padding: 0;
}
li {
  display: inline-block;
  width: 300px;
  min-height: 300px;
  border: solid 1px #e8e8e8;
  padding: 10px;
}
a {
  color: #42b983;
}
</style>
```

Forecast.vue

```
<template>
  <div>
    <h2>Five Day Hourly Forecast <span v-if="weatherData"> for {{ weatherData.city.name }}, {{weatherData.city.
country }}</span></h2>
    <p>
      <router-link to="/">Home</router-link> |
      <router-link v-bind:to="{ name: 'CurrentWeather', params: { cityId: $route.params.cityId } }">Current Wea
ther <span v-if="weatherData"> for {{ weatherData.city.name }}, {{weatherData.city.country }}</span></router-li
nk>
    </p>

    <ul v-if="weatherData && errors.length===0" class="forecast">
      <li v-for="forecast in weatherData.list">
        <h3>{{ forecast.dt|formatDate }}</h3>
        <weather-summary v-bind:weatherData="forecast.weather"></weather-summary>
        <weather-conditions v-bind:conditions="forecast.main"></weather-conditions>
      </li>
    </ul>
    <div v-else>
      <h2>Loading...</h2>
    </div>
    <error-list v-bind:errorList="errors"></error-list>

  </div>
</template>
```

```
<script>
import {API} from '@/common/api';
import WeatherSummary from '@/components/WeatherSummary';
import WeatherConditions from '@/components/WeatherConditions';
import ErrorList from '@/components/ErrorList';

export default {
  name: 'Forecast',
  data () {
    return {
      weatherData: null,
      errors: [],
      query: ''
    }
  },
  created () {
    API.get('forecast', {
      params: {
          id: this.$route.params.cityId
      }
    })
    .then(response => {
      this.weatherData = response.data
    })
    .catch(error => {
      this.errors.push(error)
    });
  },
  filters: {
    formatDate: function (timestamp){
      let date = new Date(timestamp * 1000);
      const months = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'O
ctober', 'November', 'December'];
      const weekdays = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday'];
      let daynum = date.getDate();
      let month = date.getMonth();

      let hour = date.getHours();
      if (hour === 12) {
        hour = 'Noon';
      } else if (hour === 0) {
        hour = 'Midnight';
      } else if (hour > 12) {
        hour = hour - 12 + 'PM';
      } else if (hour < 12) {
        hour = hour + 'AM';
      }
      return `${ months[month] } ${ daynum } @ ${ hour }`;
    }
  },
  components: {
    'weather-summary': WeatherSummary,
    'weather-conditions': WeatherConditions,
    'error-list': ErrorList
  }
}
</script>

<style scoped>
h1, h2 {
  font-weight: normal;
}

ul {
  list-style-type: none;
  padding: 0;
}
li {
```

```
  display: inline-block;
  width: 200px;
  min-height: 300px;
  border: solid 1px #e8e8e8;
  padding: 10px;
  margin: 5px;
}

a {
  color: #42b983;
}
</style>
```

WeatherSummary.vue

```
<template>
  <div>
    <div v-for="weatherSummary in weatherData" class="weatherSummary">
      <img v-bind:src="'http://openweathermap.org/img/w/' + weatherSummary.icon + '.png'" v-bind:alt="weatherSu
mmary.main">
      <br>
      <b>{{ weatherSummary.main }}</b>
    </div>
  </div>
</template>

<script>
export default {
  name: 'WeatherSummary',
  data () {
    return {

    }
  },
  props: {
    weatherData: {}
  }
}
</script>

<style scoped>
  .weatherSummary {
    display: inline-block;
    width: 100px;
  }
</style>
```

WeatherConditions.vue

```
<template>
  <div>
    <dl>
      <dt v-if="conditions.temp != conditions.temp_min">Current Temperature</dt>
      <dd v-if="conditions.temp != conditions.temp_min">{{ conditions.temp }}%</dd>
      <dt>Humidity</dt>
      <dd>{{ conditions.humidity }}%</dd>
      <dt>High</dt>
      <dd>{{ conditions.temp_max }}&deg;F</dd>
      <dt>Low</dt>
      <dd>{{ conditions.temp_min }}&deg;F</dd>
    </dl>
  </div>
</template>

<script>
export default {
  name: 'WeatherConditions',
```

```
  data () {
    return {

    }
  },
  props: {
    conditions: {}
  }
}
</script>

<style scoped>
  dl {
    padding: 5px;
    background: #e8e8e8;
  }
  dt {
    float: left;
    clear: left;
    width: 120px;
    text-align: right;
    font-weight: bold;
    color: blue;
  }
  dd {
    margin: 0 0 0 130px;
    padding: 0 0 0.5em 0;
  }
  dt::after {
    content: ":";
  }
</style>
```

ErrorList.vue

```
<template>
  <div>
    <div v-if="errorList.length > 0">
      <h2>There was an error fetching data.</h2>
      <ul class="errors">
        <li v-for="error in errorList">{{ error }}</li>
      </ul>
    </div>
  </div>
</template>

<script>
export default {
  name: 'ErrorList',
  data () {
    return {

    }
  },
  props: {
    errorList: []
  }
}
</script>

<style scoped>
  .errors li {
    color: red;
    border: solid red 1px;
    padding: 5px;
  }
</style>
```

There are also a couple of small clean-up tasks that should be completed as part of the steps above, but those have been left out of this summary of file changes to keep this section shorter.

Base API Configuration

The contents of `src/common/api.js`:

```
import axios from 'axios';

export const API = axios.create({
  baseURL: `//api.openweathermap.org/data/2.5/`
})
API.interceptors.request.use(function (config) {
    // Set APPID on each request
    config.params.APPID = 'd9947bfbe4d5f42fa39c0d5e08ff915f';
    config.params.units = 'imperial';
    return config;
  }, function (error) {
    // Do something with request error
    return Promise.reject(error);
  });
```

Build and Deploy

Once we've finished our work, we can build and deploy the project. This project has been configured to build to the `docs/` directory, so we can follow the same pattern we used before:

1. Execute the `npm run build` command to build the files into the `docs/` directory.
2. Commit all of our code.
3. Push the code up to GitHub.
4. Go into the repository settings and set the GH Pages section to publish from the `docs/` directory.

The project should now be up and available to the public through GH Pages.

Stretch Goals

Stretch goals are provided to gain extra practice and a higher degree of challenge. If this work has been straightforward, then we are encouraged to keep pushing. There are several more opportunities to improve the organization and structure of this project, so feel free to keep honing it until it is as lean and mean as possible.

- Abstract the addition of the `&deg; F` formatting on temperatures to a filter used in a common file.
- Abstract the `formatDate` filter to a common file.
- Create a child component to provide navigation between city search, current weather, and forecast views. Implement this component on each URL.

There may be other opportunities to improve this codebase that are not listed here. Feel free to explore and experiment with ways of rearranging the features in this application to get a better idea of what can be accomplished through refactoring.

Visual Feedback and Enhancement

We have explored a lot of concepts about building web applications, but we have not focused serious attention on the visual presentation of our work. All of the good things we already know about building website or application interfaces applies to building projects with Vue.js. The framework gives us great tools for creating visual effects that can enhance the user experience.

This is not unique or unusual. Many frameworks use similar approaches for allowing developers to react to changes in data, user actions, and other events in the interface. Frameworks often provide methods to use shorthand for toggling specific classes or automate management of specific style classes to allow the animation of new or deleted data. In this section, we will review these concepts as they are broadly defined across many frameworks, and we will look more specifically at how to use these techniques in our own Vue.js projects.

Building the base functionality is often the first step in creating a website or application that provides a value to users. But we can never rely on our first or basic implementation to do the full job. It's always essential to think about the ways that we can support users throughout whatever process we have defined. We often do this with messaging and alerts, but we also accomplish this through animations to call attention to information changes on the screen and intelligent organization of the interface.

Now that we have some idea of how to build the basic features we need for a project, let's spend some time working on making it a more pleasant experience for our users to use our creations.

Core Concept: Visual Feedback

As a user interacts with our software (website, application, game, etc.), they cause changes to happen in the system. The user might search for a city and then request the weather forecast for that city. They might search for a word and receive a set of data back from a dictionary API. They will undoubtedly interact with whatever features we have built for them, and that will causes changes in the system.

It might seem obvious to state that those changes are meaningless unless we actually show the results of those changes to the user. If we do not display the weather forecast or list of definitions for a word, then the user will find no value in using our software. So it is obvious that we need to actually show the information, but what is less obvious (especially as we begin our practice as web developers) is how important the visual presentation of that information can be in helping the user both succeed at using our software and enjoy the experience.

There are many people in the world who aim to write software that people need. In many cases, being the first person to create a software tool has guaranteed a successful project. But as competition increases, as problems become more complex, and as users become more savvy, it takes more than simply performing the task to make a successful software project. It is important to use all of our tools and abilities to increase user engagement and appreciation of the tool, and that is the direct result of building a tool that better serves the user.

Interface Design

Providing the user with a pleasurable and useful experience relies on a solid interface design. This should always be the first place we begin to consider the user's needs and how the software we are building fits into their larger experience (both online and off). It might be that our site will primarily be used by users on mobile devices or tablets. It might be that one portion of the tasks our users want to perform will be completed in the office computer, and another portion of those tasks will be completed on mobile devices. There are myriad possibilities for how our software will fit into lives, processes, and workflows beyond the screen (and another myriad for how it will interact with those same things "within" the screen, too).

For our purposes here, we will focus primarily on improving the visual feedback of our software through messaging and animation. However, both of those things rely on already having a sensible approach to the interface. And all of these techniques and thoughts go hand-in-hand, so it's important to do our best to think about them in relation to each other.

Here are some general tips that will help us make better use of messages and animation in our projects:

- Use proper HTML structures to contain information. Use specific structures where possible (e.g. lists, asides, headers, footers), and respect proper tag nesting and hierarchy rules.
- Organize information into distinct areas contained within parent elements. (This becomes very easy to do if we are building our pages from distinct components, each of which handle one specific set of data or feature.)
- Use "wrapper" elements (e.g. `div` or `span`), or use containers like list elements (e.g. `ul` and `ol`), to contain information and create a logical hierarchy.

Messaging

In web applications, being able to let the user know about changes to the system they are using is a critical aspect of designing a responsive, friendly system. In order to communicate changes that may not be possible to represent more clearly to the user through any other means, we often use "messages". Messages come in many shapes and sizes. The "loading throbber" that we see when loading data in software is a form of message. There are global messages, local messages, and all different styles and conventions of displaying messages.

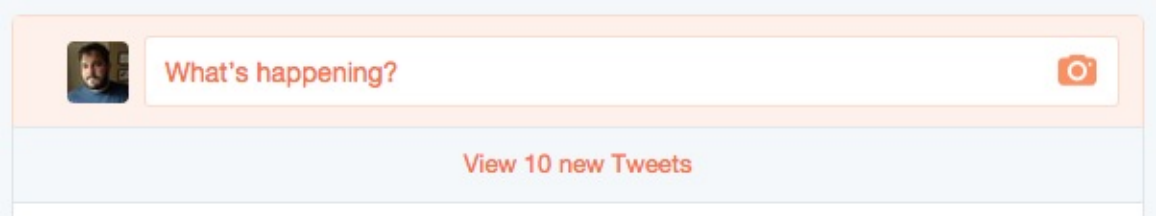
Sometimes messages are called "alerts" or "notifications." Sometimes they are called "toast" or "flash" messages. All of these words indicate a slightly different take on the use and approach to displaying messages. There are many ways to accomplish the goal of explicitly informing our user about the state of the system, and in Vue.js we have many different available methods to approach this problem. Before we get too deep into a specific implementation of messages on our website, let's consider some general characteristics of how we use messages in software.

There are many ways to use messaging in websites and apps, and developers are always coming up with clever new ways to indicate changes in content and system status. But a few conventions have grown up around messaging, and it's worthwhile to be aware of these guiding principles.

When generally thinking about messaging, the different varieties can be usefully categorized as one of two types: global and local. These are useful labels, and most software makes use of both types of messages at different times. Understanding when to use which type is important for using messages effectively in our application.

Global Messages

Global messages apply to everything the user is seeing. On Twitter, for example, as we read tweets in our timeline we may notice an alert that shows up at the top of the timeline telling us that there are a number of new tweets available to read. In many email clients, when we receive a new message, or when ew file a message away, we will see an alert at the top of the screen.



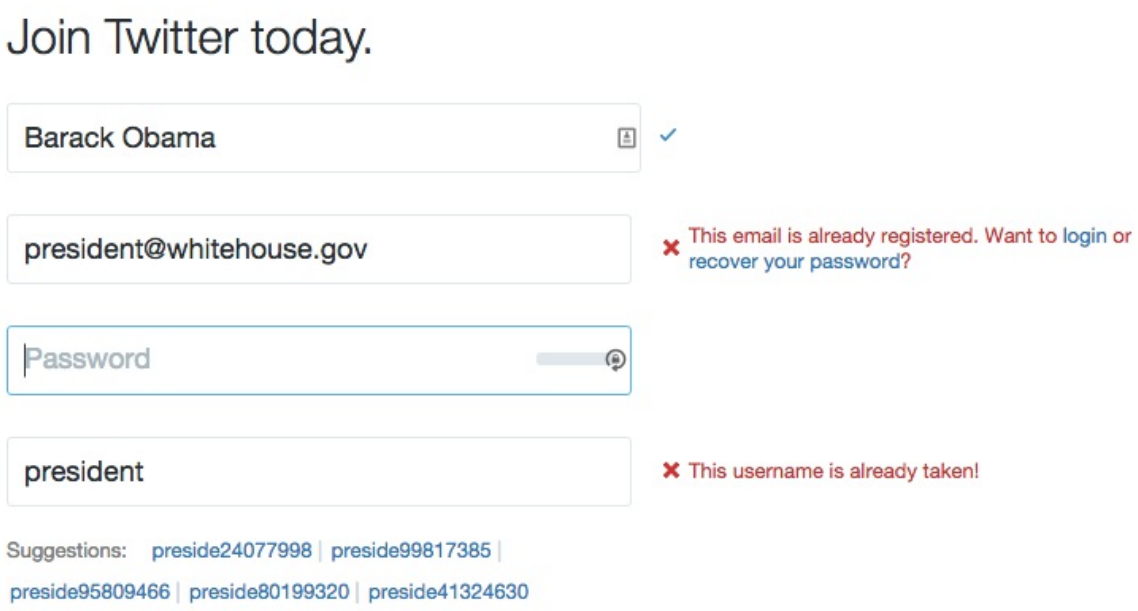
New tweets!

Global messages are great for letting us know that things are happening that generally apply to what we're doing. Depending on how they are presented, they can be more or less effective. For example, the alert pictured above is only visible when we scroll to the top of the page. If we are lower on the page, we will never know how many tweets you have.

Local Messages

Local messages appear closer to "where the action is". This is commonly seen in form field validation, especially when we're filling out more complex forms. In order to indicate where we have gone wrong, messages may be shown very close to the field (or in some way even styled as part of the form field).

Here is an example from Twitter's signup form:



Twitter signup with error notifications

As we can see in the image above, as we fill in the form the Twitter website is checking to see if our information is valid. It indicates to us clearly if we have successfully filled in the field or if we must change our information. Twitter even goes so far as to suggest alternate usernames based on the data we've filled in when our chosen name is unavailable.

Local messages are also often used in locations where data is being loaded. We often see loading "throbbers" (those spinning or pulsing animated icons that are used to indicates something is happening) in the location where a specific set of data will be loaded. We see these loading indicators in sidebars where related articles will be listed, and we see them in image or video viewers as the media files are loaded. Since our applications tend more and more to be assembled from small components, each of which might make its own request to a remote API server, these location-based loading messages are likely to remain common.

Meaningful Animations and Transitions

One of the side-effects of writing highly performant Javascript applications is that when we change the data on the screen it can be difficult for users to notice the alterations. It's not at all uncommon when working with an unstyled app for even developers, bleary-eyed from looking closely at code, may miss that a value has changed in a corner of the screen.

In order to make it more evident that changes are happening on the page, we often turn to animation. Humans are very good at noticing even small movements, especially if everything else on a page is generally stationary. We can animate position, size, color, and shape to help us draw attention to whatever we have changed.

Animations can help us show that an item has been added to a list, that an item which has been deleted is actively being removed, or that some information requires immediate attention. Animations help regain the feeling of "moving" through a website, which is somewhat lost when we abandon full page refreshes. Although the users appreciate the speediness of single page applications, they also find comfort in the indication that they are moving from one place to another. Transitional animations when replacing content on the screen help us give that feeling back to users.

By combining all of these techniques, we can create a much more enjoyable and understandable experience for users. We can prevent users from feeling confused about what just happened, and we can avoid the impression that our applications are not working by properly conveying information about loading and errors as needed. The more

explicit and clear our applications and sites can be about what is happening in front of the user, the better the user experience will be.

Dynamic Classes

In Vue.js it is possible to bind the `class` attribute on an element to data values. This can allow us to more easily toggle class assignments from within our component logic. By altering the classes on components we can cause visual changes to happen when the user interacts with our application, or call the user's attention to some part of the interface that has been updated.

These techniques are not unique to Vue.js; many frameworks allow similar kinds of class toggling and binding.

Binding Data to `class` attributes

We often have a situation where we wish to apply a class when a specific condition is true. For example, when showing a list of items with "Favorite" buttons, it would be useful to be able to apply a class when the user clicked to add an item to their "Favorites." We see this kind of interface all the time.

To bind data to class attributes we can use the `v-bind` directive. The binding of values works the same when using it with a class/style attribute as with other attributes, but Vue.js can interpret specific object structures to fill in the proper class/style changes. Here is an example of how a favorite button might indicate whether or not it has been clicked:

```
<button v-bind:class="{ favorited: isFavorite }" v-on:click="toggleFavorite">Favorite</button>
```

We can see in this example that we have used `v-bind` to bind the `class` attribute to a JavaScript object. The object defines a `favorited` property, and the value of the property is `isFavorite`. (The `isFavorite` value is defined as part of the component's data object, and it is toggled via the `toggleFavorite` method.) Whenever `isFavorite` equals `true`, the `favorited` class will be applied to this element. Whenever `isFavorite` is `false`, the `favorited` class will be removed.

We can use objects with multiple properties to add/remove multiple classes to an object in the same way. Each property should point to a boolean value that will either be `true` or `false`. This value can be altered through any normal means in our code.

For the vast majority of cases, being able to add/remove specific classes is a useful way of altering the visual appearance of an element. However, sometimes we encounter situations where binding the `style` attribute itself is required in order to achieve our desired effect.

Binding Data to `style` attributes

In some cases, we want to use some dynamic data to actually alter the style of an element. This is often required when writing complex applications such as data dashboards or browser-based games. But there is a more simple case that often requires us to directly insert a style definition: Placing the user's avatar in the background of an element. This method can be advantageous because we can more easily crop and control the display of avatars that might have differing dimensions.

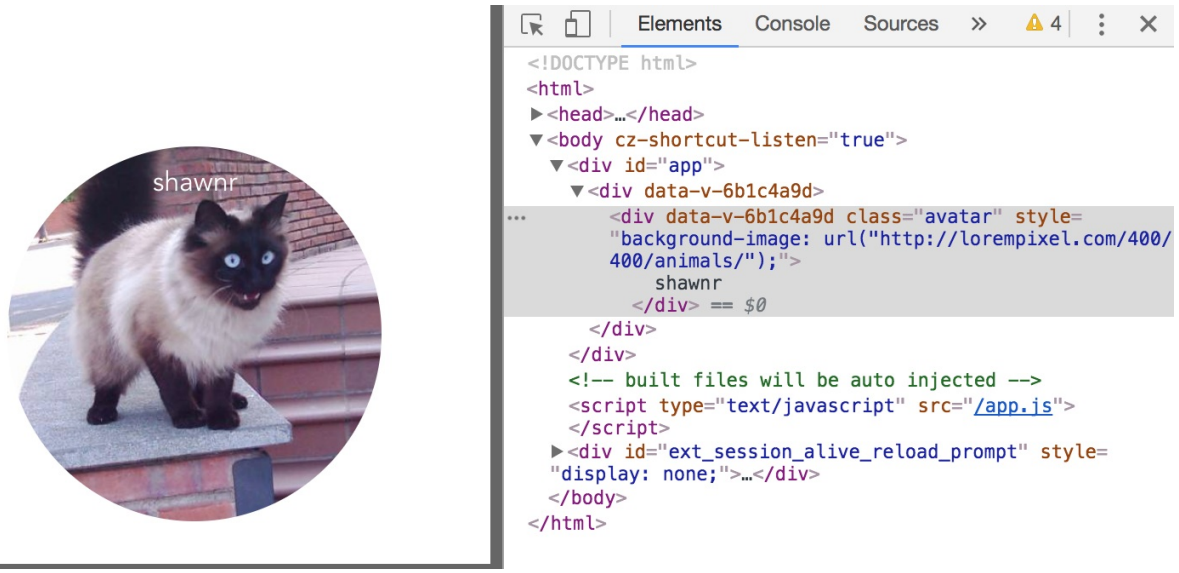
Here is an example using `v-bind` to bind the data value for a user profile image to the background of a style:

```
<template>
  <div>
    <div class="avatar" v-bind:style="{ backgroundImage: 'url('+ user.image +')' }">
      {{ user.username }}
    </div>
  </div>
```

```
</template>

<script>
export default {
  name: 'ProfileImage',
  data () {
    return {
      user: {
        username: 'shawnr',
        image: 'http://lorempixel.com/400/400/animals/'
      }
    }
  }
}
</script>

<!-- Add "scoped" attribute to limit CSS to this component only -->
<style scoped>
.avatar {
  background-repeat: no-repeat;
  background-size: cover;
  background-position: center center;
  background-color: #666;
  color: white;
  text-align: center;
  padding: 10px;
  width: 200px;
  height: 200px;
  border-radius: 50%;
}
</style>
```

We can see that this component displays a profile image. It uses a data object called `user` to provide the username and the URL for the user avatar image. We have defined the styles to shape the avatar image and control its size, but we cannot write a style definition that includes the user's specific image URL. Instead, we must apply the image URL to the `background-image` style property for the `div.avatar` element.

To accomplish this, we bind another data object to the `style` property. This time, Vue.js knows to read the properties of the data object looking for terms that correspond to CSS properties. **Please note:** We can use either "kebab case" (with dashes, like `background-image`) or "camel case" (with capital letters, like `backgroundImage`) to reference CSS properties. Vue.js will map these properties and their values to CSS properties and insert them into the `style` attribute. The end result in our example looks like this:



User Avatar with Dynamic Background

Using this technique, we gain a lot of flexibility. We can define the data object in our template, or we can define a data object in our component logic and then reference the name of that object in the template. We can set up event listeners to dynamically change values based on user interactions, which can then be translated into visual effects on the screen. This opens the door for all sorts of interesting interface concepts and designs.

Transitions and Animations

When data changes on the screen, we often use transitions and animations to call the user's attention to what has just been added or removed. The movement we use often mimics the change that we are trying to illustrate. For example, a user might delete an item from a list, which might slide out and fade away. Or a refresh of data might cause items to enter a list, sliding in from the side and landing with a bounce.

This movement makes our interfaces feel more lively and helps conceptually ground the action the user has taken. The addition of movement makes our software feel more real. Fortunately, Vue.js gives us a few powerful tools for animating transitions within our applications.

Enter/Leave/List Transitions

Whenever content enters or leaves the screen, or whenever information is added or removed from a list, Vue.js will automatically add some specific classes to the HTML elements. These classes can be defined by developers to trigger whatever animation or transition we wish.

Here is the full list of transition classes provided by Vue.js.

Class Name	When Applied	When Removed	Definition
`.v-enter`	Before the element is inserted.	One frame after the element is inserted.	Starting state for enter.
`.v-enter-active`	During entire enter phase.	When transition finishes.	Active state for enter.
`.v-enter-to`	One frame after element is inserted (when `.v-enter` is removed).	When transition finishes.	Ending state for enter.
`.v-leave`	Immediately when leaving animation is triggered.	One frame after leaving animation is triggered.	Starting state for leave.
`.v-leave-active`	During entire leave phase.	When transition finishes.	Active state for leave.
`.v-leave-to`	One frame after leaving animation is triggered.	When transition finishes.	Ending state for leave.
~~	~~	~~	~~

These are the default names of the classes that are applied as content is added to or removed from the display. It is possible when defining a transition to provide a unique prefix for these classes that will replace the `v-` part of the name. For example, if we named a transition `foo`, then we would write the styles `.foo-enter` and `.foo-leave` instead of `.v-enter` and `.v-leave`.

By defining different styles for each of these points, and setting up CSS transitions between these styles (or using CSS animations to provide more complex animations), we can create fade in and out effects. But in order to apply these styles, we must wrap content in our templates in a `<transition>` component in order to let Vue.js know we want to use these styles when elements are added and removed from the display.

Transitioning Single Elements

To transition a single element, we can use the `<transition>` component, which is provided by the Vue.js framework and can be used without importing it or adding it to our project. The `<transition>` component accepts a `name` attribute, and it should wrap a single HTML element. The `name` attribute will cause that name to be prepended to the transition class names applied by Vue.js. Transitions can be applied to any element that is controlled with a `v-show`, `v-if`, or any dynamic components or component root nodes. So we can always animate the display or removal of an entire component, but this is not the component that handles transitions for other elements (such as list items). (We will explore that other component shortly.)

As an example, we can look at how we might animate the display of the FAQ answers from our old FAQ example. The FAQ display uses a `Question` component that handles the showing/hiding of the answer text. We can wrap the answer in a transition and make the display of that information more eye-catching:

```
<template>
  <div class="question">
    <h2><a v-on:click="toggleAnswer">{{ question }}</a></h2>
    <transition name="fade">
      <p v-show="showAnswer" class="answer">{{ answer }}</p>
    </transition>
  </div>
</template>

<script>
export default {
  name: 'FAQ',
  data () {
    return {
      showAnswer: false
    }
  },
  props: [
    'question',
    'answer'
  ],
  methods: {
    toggleAnswer: function () {
      this.showAnswer = !this.showAnswer;
    }
  }
}
</script>

<style scoped>
.fade-enter-active, .fade-leave-active {
  transition: opacity .5s
}
.fade-enter, .fade-leave-to {
  opacity: 0
}
h1, h2 {
  font-weight: normal;
}
a {
  color: #42b983;
  cursor: pointer;
}
</style>
```

We can see that the `p.answer` element has been wrapped in a `<transition>` component with the name `fade`. This corresponds to the styles that have been defined: `.fade-enter-active`, `.fade-leave-active`, `.fade-enter`, and `.fade-leave-to`. Note that it is not required to define styles for every single class used in the transition. Using different classes will lead to different results, and the possibilities are incredibly diverse. We could just as easily be using other CSS animations to do fly-ins, bounce-outs, or whatever other effect we feel best serves our users.

FAQ Example Fades

This single element transition is commonly found, but it is also common for us to want to animate items being added to or removed from other elements (or multiple elements together). In these cases, we must use the `<transition-group>` component.

Transitioning Groups of Elements

In order to transition groups of elements (list items or other things generated using a `v-for` directive), we can use the `<transition-group>` component. This component functions in a way similar to the single `<transition>` component. It also takes a name attribute to modify the names of the classes that are automatically added to the elements as they transition. We can also use the other attributes that are available on the `<transition>` element (such as the `appear` attribute, which makes an animation take place when the content appears on the page without being toggled by the user).

In order to use the `<transition-group>` component, we must label each item we want to transition with a key, which we can do by binding a unique value to the key as we process a `v-for` loop. We must also specify a tag that the transition will wrap around the group of elements included in the effect. The `<transition>` component defaults to using a `<span>` tag to wrap the elements. But we can modify that easily. Here is an example of how we add animations to list items using the `Forecast` screen from the previous project.

template

```
<transition-group name="fade" tag="div" appear>
  <li v-for="forecast in weatherData.list" v-bind:key="forecast">
    <h3>{{ forecast.dt|formatDate }}</h3>
    <weather-summary v-bind:weatherData="forecast.weather"></weather-summary>
    <weather-data v-bind:weatherData="forecast.main"></weather-data>
  </li>
</transition-group>
```

styles

```
.fade-enter-active, .fade-leave-active {
  transition: opacity 1s
}
```

```
.fade-enter, .fade-leave-to {
  opacity: 0
}
```

We can see that we have specified a `<div>` element to wrap all of our list items. We have not modified our list output at all except for the addition of the `key` attribute on the `<li>` tag. We have bound the `forecast` value to the key, which gives us the unique identifier the transition group needs to operate. In the styles block, we have defined the same fade effects we used on the previous example. We could have defined any other transition effect if we preferred. We have also used the `appear` attribute on the `<transition-group>` component so the list items will be animated the first time they appear on the screen (without any change from the user).

Forecast Fade-In

This effect creates a nice visual cue that information has been updated. In other situations, we could add effects to make it more obvious when data is removed from a list or added to the list. We can even explore more of the features of transition groups to animate the re-ordering of a list, which is a valuable effect in some situations.

Pushing Further

Although we have learned enough in this section to dramatically improve our interface, it is possible to do more with transitions than we have explored here. If we're feeling like a challenge, or more exploration, it's worthwhile to dig into the Vue.js Guide and learn about list move transitions and reusable transitions among other fascinating details.

In addition to the built-in features of Vue.js, it is worthwhile to explore some of the other third-party modules that have been created to provide some great animation and transition experiences. The Animation section on the Awesome Vue list is a great place to start. The `vue2-animate` package will be used in the project that goes with this section.

Messaging

Clear messaging is a great technique for informing the user about what is happening in our software at any given point in time. As discussed earlier, using either global or local messages on a page can really improve the user experience and help the user succeed at accomplishing their goals. The ways we implement messages can vary widely, but they don't need to be terribly complex. There are many ways to solve the messaging problem in Vue.js, and plenty of third-party tools we can rely on to help us handle messages.

Loading Throbbers

We can create a simple loading throbber component for use whenever we need to load data from a third-party API. The component can be used like any other child component within a template, and easily turned "on" and "off" using a boolean value in our component logic. Here is what a load throbber component might look like:

```
<template>
    <div v-if="showThrobber" class="spinner"></div>
</template>

<script>
export default {
  name: 'LoadThrobber',
  data () {
    return {

    }
  },
  props: {
    showThrobber: true
  }
}
</script>

<style scoped>
.spinner {
  width: 40px;
  height: 40px;
  background-color: #333;

  margin: 100px auto;
  -webkit-animation: sk-rotateplane 1.2s infinite ease-in-out;
  animation: sk-rotateplane 1.2s infinite ease-in-out;
}

@-webkit-keyframes sk-rotateplane {
  0% { -webkit-transform: perspective(120px) }
  50% { -webkit-transform: perspective(120px) rotateY(180deg) }
  100% { -webkit-transform: perspective(120px) rotateY(180deg)  rotateX(180deg) }
}

@keyframes sk-rotateplane {
  0% {
    transform: perspective(120px) rotateX(0deg) rotateY(0deg);
    -webkit-transform: perspective(120px) rotateX(0deg) rotateY(0deg)
  } 50% {
    transform: perspective(120px) rotateX(-180.1deg) rotateY(0deg);
    -webkit-transform: perspective(120px) rotateX(-180.1deg) rotateY(0deg)
  } 100% {
    transform: perspective(120px) rotateX(-180deg) rotateY(-179.9deg);
    -webkit-transform: perspective(120px) rotateX(-180deg) rotateY(-179.9deg);
  }
```

```
}
</style>
```

Notice that this component is very simple: Merely a single `div` element with a class applied to it. It uses a boolean value to show/hide itself. The load throbber (or "spinner" as it's labeled in the CSS) was borrowed from the SpinKit set, which is a great resource. There are many ways to make an animated load throbber, including using animated GIFs, SVGs, or pure CSS animations.

As with so many aspects of web development, if we would prefer to use a solution created by somebody else, we can leverage one of the many projects listed on the Awesome Vue list under Loaders.

Global Messages

Global messages can also be handled nicely with child components. Message styles can be contained in the dedicated component, and that component can be used by any other components that need to produce global messages. Messages can be styled to show like "toast" displays (popping up from the bottom or out from another side of the viewport), or they can simply fade in or overlay on the page content.

Here is an example of a simple message list component that could be used in multiple locations throughout an application:

```
<template>
    <ul v-if="messages.length > 0" class="message-container">
      <li v-bind:class="message.type" v-for="message in messages">
        <h2>{{ message.title }}</h2>
        <p>{{ message.content }}</p>
      </li>
    </ul>
</template>

<script>
export default {
  name: 'Messages',
  data () {
    return {

    }
  },
  props: {
    messages: []
  }
}
</script>

<style scoped>
.message-container li {
  padding: 10px;
  margin: 5px;
  font-size: 1rem;
}
.error {
  color: red;
  background: pink;
}
.info {
  color: black;
  background: #e8e8e8;
}
.success {
  color: white;
  background: green;
}
```

```
.warning {
  color: black;
  background: yellow;
}
h2 {
  font-size: 1rem;
}
</style>
```

We can see that we have a simple template defined to output messages. The message `type` is used to determine the styles that should be applied to the message. Each message consists of a `title` and `content`. These are common attributes for generic messages. We could easily maintain an array of messages that should be displayed. We could even break this down further and enhance this component with child components of its own that would allow us to close a message when the user clicks an icon or provide a timeout on messages so they fade away after some period.

Of course, there is really no reason to build our own messaging system other than the fact that we can learn more and control more by doing it that way. The Awesome Vue list has several Notifications modules we can leverage to easily add robust messages and notifications to our applications.

Local Messages

Local messages can be inserted into templates and controlled via `v-if` directives to provide whatever other messaging or information we need. These methods of controlling the display of content in a very localized fashion have already been covered in previous sections of this book. Those same methods will work for very many local message use cases.

Where our previously-covered methods may not work as well is when we are validating form data. We have previously looked at providing feedback in terms of form validation, and we can always write our own validation logic. But it is also possible to, once again, leverage the efforts of the community. We can bring in any number of form validation packages for Vue.js and use those.

Another use case for very localized messaging is in the "tooltip", which is a message usually tied to a specific HTML element. We often see tooltips when we hover over elements on the screen. They are also used to explain details about form fields and other interface elements. And, once again, there are numerous packages that provide tooltip features for Vue.js projects.

Although Vue.js gives us the tools to easily create these features for ourselves, we still can benefit from using existing packages to help us move along and focus on more unique aspects of our projects. Each of the packages suggested on the Awesome Vue list works in a slightly different way, and each one might be better or worse for your purposes. Be sure to browse around and get an idea of how these different packages work, but the only way to know for sure is to dive in and try some of them out. If we don't find what we need, we can always build it ourselves.

Quiz: Visual Feedback and Enhancement

Please enjoy this self-check quiz to help you identify key concepts, points, and techniques discussed in this section.

Visit Quiz Online

The quiz on this page has been removed from your PDF or ebook format. You may take the quiz by visiting this book online.

Project: Visual Feedback and Enhancement

This project uses the Visual Enhancement starter repository. Be sure to fork and clone this repository to get started.

For this project, we will revisit the Datamuse API to enhance the search tools we worked with a few projects ago. We will present one search with multiple options, and we will allow users to save their results to a "Word List" so they can more effectively browse for words.

This project is mostly built, but it requires some love and affection by way of visual enhancements to make it more useful. With such a complex search tool, it is possible for users to get no results. When that happens, we don't want them to think the application has malfunctioned.

We are also updating information in the search results and in the sidebar Word List. We want to use messages and animation to help users identify when data in these areas of the page has changed.

To make it a little easier to create professional quality animations, we will rely on the `vue2-animate` project, which makes a set of common animations available for use with Vue.js transition components. We will also use a load spinner from Spinkit.

Review the Requirements

In order to complete this project, we will mainly be adding elements to enhance the messaging and visual presentation of the application. We must complete the following requirements, which will have us editing the `src/components/WordSearch.vue` file. (Each requirement corresponds to the `TODO` notes, so look for those.) Here are the basic requirements:

- Use the `showSpinner` value to modulate the display of the `CubeSpinner` component when appropriate
- Add an animation to the items of the results list when a search is completed
- Add an animation to the items of the WordList for when new items are added and removed
- Add messaging to results display area let the user know when no results are found
- Add global messaging child component (`MessageContainer`) to `WordSearch` component
- Add a global "success" message to let the user know that a word has been successfully added to the WordList
- Add a global "info" message to let the user know when they try to add a word to the WordList that has already been added
- Add a global "success" message to let the user know that a word has been successfully removed from the WordList
- Add a global "error" message to display any errors from the API request (aside from "no results found")

Working the Project

The following guide offers a walkthrough of how to complete almost every aspect of the project.

Add `CubeSpinner` to Indicate Loading

Like many of the tasks in this project, this one should be familiar based on the previous project. We will add the `import` statement for the `CubeSpinner` component, and then will add the component to the list of components. Once we have done that, we will add the `<spinner>` element to our template. We will use a regular `v-if` directive on the `<spinner>` element to determine whether or not to show the spinner. Here is what the basic implementation looks like:

template

```
<spinner v-if="showSpinner"></spinner>
```

script

```
import CubeSpinner from '@/components/CubeSpinner';

export default {
  name: 'WordSearch',
  components: {
    spinner: CubeSpinner
  },
  // ... additional code ... //
```

This is just like how we added the child components in the previous project. Now we must modulate the value of `this.showSpinner` from within the `findWords` method in order to control the show of the loading spinner. Here is how we can do that:

```
findWords: function() {
  // Show spinner when API request begins here.
  this.showSpinner = true;
  this.results = null;
  axios.get('https://api.datamuse.com/words', {
    params: {
      ml: this.phrase,
      sl: this.soundsLike,
      sp: `${this.startLetter}*${this.endLetter}`
    }
  })
  .then( response => {
    // Turn off spinner.
    this.showSpinner = false;
    this.results = response.data;
  })
  .catch( error => {
    // Turn off spinner.
    this.showSpinner = false;
  })
}
```

As we can see here, we can easily use the structure of the API request to turn on and off the spinner animation to indicate loading. Usually, the Datamuse API loads very quickly, but on slower connections or for more complex searches this loading spinner will help our users know exactly what is happening in the app.

Load Spinner

Now we should be able to test and see that we have a load spinner showing up whenever the application is requesting data from the server.

Animate the Items in the Search Results

To animate the items in the search results, we can wrap the list items in a `<transition-group>` element. We must also add a `key` attribute to each list item (`key` attributes are required whenever we use a `<transition-group>`).

```
<transition-group name="fade" tag="div" appear>
  <li v-for="item in results" class="item" v-bind:key="item.word">
    <p class="result-word">{{ item.word }}</p>
    <p><button v-on:click="addWord(item.word)" class="add-word">Add to WordList</button></p>
  </li>
</transition-group>
```

Here we are using the `item.word` value as the `key` , and we have set the `tag` attribute on the `<transition-group>` to `"div"` . This will wrap the set of list items in a `div` tag (as opposed to the default `<span>` tag that the `<transition-group>` uses).

At this point, we should notice that this project has the `vue2-animate` module added. We can see the module listed in our `package.json` file. And we can notice at the top of the logic in our `WordSearch` component is a somewhat odd-looking line:

```
require('vue2-animate/dist/vue2-animate.min.css');
```

The `vue2-animate` module is unusual because it is only a CSS file. It contains style classes that have been defined to work with the conventions of the `Vue.js` transition system. This means that we can access a large range of premade animations by simply naming our `<transition-group>` with the name of the animation. In the case above, we will see the items in the results list fade in. We can get a better idea of all the animations `vue2-animate` makes available by looking at the `vue2-animate` Demo Page.

Experiment with a different animation and see what it looks like. It is easy to switch between animations and trying different animations will help us get an idea for how the system works.

Animate the items in the Word List

Animating the items in the Word List works almost exactly the same way as the items in the results list. We can wrap the list items in a very similar `<transition-group>` element. We make the same changes to add a `key` to the list items and set the `name` and `tag` attributes of the `<transition-group>` .

```
<transition-group name="slideRight" tag="div" appear>
  <li v-for="word in wordList" v-bind:key="word">{{ word }} <button v-on:click="removeWord(word)" class="r
emove-word">x</button></li>
</transition-group>
```

Notice that this time we have used the `slideRight` animation. We also have the `appear` attribute on the `<transition-group>` to make sure the animation will be shown even when the first item is added to the list. We are using the `word` value as the `key` for each list item.

It's worthwhile to notice that when we remove a word from the list, the opposite animation happens. This is a feature of the way the `vue2-animate` styles have been defined. They combine animations to work for both entry and exit. Try different animations and see what it looks like when words are added to the list or removed.

Add `MessageContainer` for Global Messages

A `MessageContainer` component has been provided to handle display of messages. These components can be easy to create for basic purposes, but it's also very common to use third-party components to display messages in our applications. It all depends on our specific project needs.

To use the `MessageContainer` , we need to import it into the `WordSearch` component and add it to the list of components used in `WordSearch` . This is very similar to how we previously imported and set up the `CubeSpinner` component.

```
<script>
import axios from 'axios';
require('vue2-animate/dist/vue2-animate.min.css');

import CubeSpinner from '@/components/CubeSpinner';
import MessageContainer from '@/components/MessageContainer';

export default {
  name: 'WordSearch',
  components: {
    spinner: CubeSpinner,
    'message-container': MessageContainer
  },
  // ... more script ... //
</script>
```

Once we have added the child component to `WordSearch` , we can use it in the template:

```
<message-container v-bind:messages="messages"></message-container>
```

The `MessageContainer` component looks for a property called `messages`, which we want to bind to the `messages` data value in the `WordSearch` component. Now we can cause messages to be displayed by adding or removing messages to the `messages` array inside of `WordSearch`.

Add Messages to `addWord` Method

Adding messages is just a matter of pushing a `message` object into the `messages` array. The `message` object expects two properties: `type` and `text`. This allows the `MessageContainer`, and the `MessageItem` child component it uses, to properly display each message. Here is how we can add a message to indicate that a word has been added to the Word List:

```
addWord: function (word) {
  if (this.wordList.indexOf(word) === -1) {
    this.wordList.push(word);
    this.messages.push({
      type: 'success',
      text: `${word} added to WordList.`
    });
  } else {
    this.messages.push({
      type: 'info',
      text: `${word} is already on the WordList.`
    });
  }
}
```

In the `addWord` method, we receive `word` as an argument. This `word` is compared to the existing `this.wordList` array to see if it already exists in the list. If the `word` is not found in the array, then it is a new word, and it is added to `this.wordList`. We also add a `message` object to the `this.messages` array with the `type` set to `"success"` and the `text` set to a helpful message.

If the `word` already exists in the `this.messages` array, then we do not re-add the word, and we instead create an "info" message that lets the user know why the word was not added. This is a common situation when allowing users to add items to a list of things.

Add Messages to `removeWord` Method

Adding the message creation to the `removeWord` method is even easier than adding it to the `addWord` method because there is only one case. We can add it like so:

```
removeWord: function (word) {
  this.wordList.splice(this.wordList.indexOf(word), 1);
  this.messages.push({
    type: 'success',
    text: `${word} removed from WordList.`
  });
}
```

We use the `splice()` method, which exists on every JavaScript Array, to remove the word we're after. Once we've done that, we alert the user. Once we get the hang of this, it becomes very easy to add messages to all the parts of our application.

Now we should see pretty much all of these features at play in our application:

Add/Remove Words

Display Errors with Messages

The last place we need to add message display is to our possible API errors. We want the user to know if our application is malfunctioning because of the API; otherwise, they will assume that our application is broken and never realize that the API we depend upon is having troubles. It might be cold comfort, but it makes us feel better as developers to let our users know that we're still looking out for them even when things are going wrong beyond our control.

Adding messages to the error clause of our API request is almost the same as adding them everywhere else:

```
.catch( error => {
  this.showSpinner = false;
  this.messages.push({
    type: 'error',
    text: error.message
  });
})
```

The only unique thing here is that we are using the `"error"` type of message, and our message content is coming from the `error` object returned by `axios` when a request goes wrong. If we want to test these messages we can do so by adding some bogus characters to the domain of our API server and then viewing our site. When we click "search" we should see a "Network Error" show up.

Wrapping Up

Now that we've completed the project, here is what the `WordSearch.vue` file looks like:

```
<template>
  <div>
    <div class="messages">
      <message-container v-bind:messages="messages"></message-container>
    </div>
    <div class="word-search">
      <form v-on:submit.prevent="findWords">
        <p><label>Find synonyms for <input type="text" v-model="phrase" placeholder="word or phrase"> that:</la
bel></p>
        <ul>
          <li><label>sounds like <input type="text" v-model="soundsLike" placeholder="word or phrase"></label></
li>
          <li><label>start with the letter <input type="text" v-model="startLetter" placeholder="single letter">
</label></li>
          <li><label>end with the letter <input type="text" v-model="endLetter" placeholder="single letter"></l
abel></li>
        </ul>
        <p><button type="submit">Search</button></p>
      </form>
    </div>
    <div class="word-list-container">
      <h2>Word List</h2>
      <ul class="word-list">
        <transition-group name="slideRight" tag="div" appear>
          <li v-for="word in wordList" v-bind:key="word">{{ word }} <button v-on:click="removeWord(word)"
class="remove-word">x</button></li>
        </transition-group>
      </ul>
    </div>
    <div class="results-container">
      <spinner v-if="showSpinner"></spinner>
      <h2 v-if="results && results.length > 0">{{ results.length }} Words Found</h2>
      <ul v-if="results && results.length > 0" class="results">
        <transition-group name="fade" tag="div" appear>
          <li v-for="item in results" class="item" v-bind:key="item.word">
            <p class="result-word">{{ item.word }}</p>
            <p><button v-on:click="addWord(item.word)" class="add-word">Add to WordList</button></p>
          </li>
        </transition-group>
      </ul>
      <div v-else-if="results && results.length === 0" class="no-results">
        <h2>No Words Found</h2>
        <p>Please adjust your search to find more words.</p>
      </div>

    </div>
  </div>
</template>

<script>
import axios from 'axios';
require('vue2-animate/dist/vue2-animate.min.css');
import CubeSpinner from '@/components/CubeSpinner';
import MessageContainer from '@/components/MessageContainer';

export default {
  name: 'WordSearch',
  components: {
    spinner: CubeSpinner,
    'message-container': MessageContainer
  },
  data () {
    return {
      results: null,
      wordList: [],
      messages: [],
      phrase: '',
      soundsLike: '',
```

```
      startLetter: '',
      endLetter: '',
      showSpinner: false
    }
  },
  methods: {
    addWord: function (word) {
      if (this.wordList.indexOf(word) === -1) {
        this.wordList.push(word);
        console.log(`Added ${word} to wordList.`);
        this.messages.push({
          type: 'success',
          text: `${word} added to WordList.`
        });
      } else {
        console.log('Word is already on wordlist.');
        this.messages.push({
          type: 'info',
          text: `${word} is already on the WordList.`
        });
      }
    },
    removeWord: function (word) {
      this.wordList.splice(this.wordList.indexOf(word), 1);
      this.messages.push({
        type: 'success',
        text: `${word} removed from WordList.`
      });
    },
    findWords: function() {
      this.showSpinner = true;
      this.results = null;
      axios.get('https://api.datamuse.com/words', {
        params: {
          ml: this.phrase,
          sl: this.soundsLike,
          sp: `${this.startLetter}*${this.endLetter}`
        }
      })
      .then( response => {
        this.showSpinner = false;
        this.results = response.data;
      })
      .catch( error => {
        this.showSpinner = false;
        this.messages.push({
          type: 'error',
          text: error.message
        });
      })
    }
  }
}
</script>

<style scoped>
.word-search {
  font-size: 1.2rem;
  white-space: nowrap;
  display: inline-block;
  width: 70%;
  float: left;
}
.word-list-container {
  display: inline-block;
  width: 25%;
  background: #e8e8e8;
  padding: 0.5rem;
}
```

```
.results-container {
  clear: both;
}
input[type="text"]{
  border-top: none;
  border-left: none;
  border-right: none;
  border-bottom: 1px solid #333;
  width: 300px;
  font-size: 1.2rem;
  color: #2c3e50;
  font-weight: 300;
  background: rgba(0,0,0,0.02);
  padding: 0.5rem;
}
button{
  background: #333;
  padding: 0.5rem;
  font-weight: 300;
  color: #fff;
  border: none;
  cursor: pointer;
  font-size: 1.4rem;
  border-radius: 0;
}
button.add-word {
  background: #e8e8e8;
  color: #333;
  font-size: 0.8rem;
}
button.add-word:hover {
  background: #fde300;
}
button.remove-word {
  font-size: 0.5rem;
  padding: 2px;
  display: inline-block;
  color: #333;
  background: none;
}
button.remove-word:hover {
  background: #aa0000;
  color: #fde300;
}
h1, h2 {
  font-weight: normal;
}
ul.results, ul.word-list {
  list-style-type: none;
  padding: 0;
}
.word-list li {
  margin: 5px 0;
  padding: 5px 0;
  border-bottom: 1px solid #333;
}
.results li {
  display: inline-block;
  margin: 10px;
  border: solid 1px #333;
  padding: 0.5rem;
  width: 200px;
  min-height: 100px;
  color: #fff;
  font-weight: 300;
  font-size: 1.2rem;
  background: rgba(0,0,0,0.7);
}
ul.errors {
```

```
    list-style-type: none;
  }
  .errors li {
    border: 1px solid red;
    color: red;
    padding: 0.5rem;
    margin: 10px 0;
  }
  a {
    color: #42b983;
  }
  </style>
```

Build and Deploy

Once we've finished our work, we can build and deploy the project. This project has been configured to build to the `docs/` directory, so we can follow the same pattern we used before:

1. Execute the `npm run build` command to build the files into the `docs/` directory.
2. Commit all of our code.
3. Push the code up to GitHub.
4. Go into the repository settings and set the GH Pages section to publish from the `docs/` directory.

The project should now be up and available to the public through GH Pages.

Stretch Goals

If we crave more challenges, try tackling some of these suggestions.

- Try making custom animations for everything, and don't use `vue2-animate` animations at all
- Add a shuffle feature to the wordlist along with the accompanying animation
- Make other layout or design improvements that help the application be more helpful to our users

Caching Data

As frontend developers, we do not need to do much with the creation of databases and management of those systems. We typically interact with those systems through established means, especially in the world of single page applications that use APIs to provide the bulk of their functionality. Although the API itself is an application running on a server, and most APIs deal with the retrieval of information from a database, as frontend developers our challenge is primarily to get to know the data and features the API provides.

Although we get a bit of a break in terms of designing data models and managing database systems, we do have a set of challenges that uniquely affect the frontend: How can we leverage tools for caching information in order to make our applications more performant and friendly to API providers?

"Caching information" in the web application context refers to the process of saving a copy of data so it can be accessed without making another network request. Saving data locally provides many opportunities for us to improve our websites and applications, and to provide a more efficient user experience.

Most API providers limit the frequency with which we can make API requests. If we have established a private API server to handle the functions of our product system, then we have a vested interest in minimizing the impact we put on that system in order to stretch our resources further and provide speedier responses. We want to provide a higher level of convenience for our users, so they do not have to re-configure the system or re-enter data required by the system. We also want to be smart about when we make a request to a remote API server, preferring to avoid those calls when possible.

We can accomplish these goals using some basic tools provided in the browser and made available through JavaScript. We can even leverage some helpful modules to enhance those core features so we can more effectively work with the tools provided by the browser. These tools allow us to store data and retrieve it without making a network request.

As with many aspects of web development, there are multiple ways of approaching the task of saving information about the user and/or caching API responses. In this section we will explore a couple of them.

Core Concept: Methods of Storing Frontend Data

There are several methods made available to us for storing data on the frontend (in the browser). These include cookies, session storage, local storage, IndexedDB, and Web SQL (which is deprecated, so we won't be covering it). All of these technologies get wrapped up into the "Storage" concept in the browser, which seeks to unify the many methods of storing information on behalf of the user.

Later in this section of the book, we will work with `localStorage` to improve the functionality of our webapps. But let's first look at how each of these technologies works, and what we use it for.

Cookies

One of the oldest tools for storing information in the browser on behalf of the user is the "cookie." Cookies have often been maligned in mainstream media, and reliance on cookies can be problematic in some cases, but these have always been useful tools for storing small pieces of information. We often use cookies to store a "logged in" token for a user, so they do not have to login each time they return to the site. We control that cookie with the "remember me" checkbox that is on so many login forms around the web.

We also use cookies to store other small bits of information, such as site preferences or tokens. Since cookies can contain just about any blob of text data, they can be used to store all sorts of information (such as tracking numbers, passwords, etc.). These uses are regulated in some areas, such as in the EU where a site must make us aware if they are using cookies.

Cookies are tied to the domain that set the cookie in the browser. Domains can only read cookies associated with them, and when a user hits a domain that corresponds to a stored cookie, then the cookie is automatically sent to that domain.

Cookies default to existing for the duration of the user's session on the server, and they will disappear when the user closes their browser. However, it is possible to specify an expiration date for a cookie, allowing it to be "permanent" until that specified date. This is how cookies used for maintaining login states work: They are set for some period of time, after which the user must login again to re-establish a new cookie.

It is possible to increase the security of cookies by marking them "secure" or "HTTP Only." These denotations help systems maintain a higher level of security around the cookie, but they should never be considered fully secure. Sensitive data should **never** be stored in a cookie.

There are many known security issues with cookies. Cross-site scripting (XSS) attacks can be used to harvest cookies from sites where the cookies have been set to secure and/or HTTP Only. Cross-site request forgery (CSRF) attacks can be used to execute a request on a third-party site on behalf of the user without them being aware of the request. In both cases, it is possible to prevent this abuse, and many application frameworks and tools take these risks into account.

Cookies are still a commonly-used technique in the web development landscape, but they are limited in how useful they can be. For the purposes of persisting sessions and tracking lightweight configuration data, they can be helpful. But they are not meant to store lots of information and they are not secure enough to store sensitive information.

`localStorage` and `sessionStorage`

A common replacement for cookie storage is to use `localStorage` and `sessionStorage`. These forms of storage are revealed through the JavaScript API and accessible to our web applications. It is common to use third-party tools to provide helpful features for managing our storage options, but it is not necessary.

The main difference between `localStorage` and `sessionStorage` is that `localStorage` is persistent (meaning it exists even after the user closes the tab/browser and then returns to the site at a later date), while `sessionStorage` goes away when the user ends their session. Because of the ability of `localStorage` to persist across sessions, it is an ideal tool for caching information in the browser that the user will need on subsequent sessions.

Both forms of storage allow us to create "key/value pairs", which means that we can give a name (the "key") to a "value" (which can be any String information). Since we can reduce any data to a String, we can store any data structure in these storage options. We can turn JSON data into a String using `JSON.stringify` and we can turn the string back into a JSON object using `JSON.parse`.

A variety of third-party tools allow us to move between `sessionStorage` and `localStorage` and give us built-in features to manage the manipulation of data into and out of String formats. These tools (such as `vue-ls`) can make it very easy to use these storage mechanisms in our applications to cache API data and store user-generated information. We can allow a lot of functionality in our applications even if the network is not available by caching information on behalf of our users.

We should keep in mind that these storage mechanisms are all susceptible to user manipulation. Users can clear their browser data at any time, which will wipe out any information stored in `localStorage` or `sessionStorage` (or cookies, for that matter). As developers, we must take care to provide mechanisms to detect and restore cached data when the user clears it, and we should never leave the user with a broken application because we assume too much about what data will exist in storage.

IndexedDB

Cookies are good for storing tiny bits of information: A 32-bit token, a handful of interface preferences, etc. The storage mechanisms are good for storing larger chunks of data in individual keys, which is useful for things like caching user profiles, individual API responses, or lists of data objects created by the user. But what about when we want to cache significant amounts of data in the browser for the user? What if we want to provide a quality experience for the user to browse their movie collection or email while they are offline? These other methods will not suffice.

This is where the IndexedDB technology comes into play. IndexedDB is a non-relational database technology that stores key/value pairs. Unlike the key/value pairs of the storage mechanisms, IndexedDB can store complex objects as the value, and properties of those objects can be used to create "indexes" in the system. The indexes provide faster methods of searching the data in an IndexedDB, and developers can use complex queries to get exactly the data they need at any given time.

The IndexedDB gives database abilities on par with any other non-relational database. This database is suitable for storing large data sets that can even contain images or files. A user could store the data about their entire movie library, their book collection, or their emails, and that data could be referenced even when the network is not available. An email client can use IndexedDB to support an offline mode and to speed functions for the user. A todo list could use IndexedDB to allow for offline functionality so it even works inside buildings with no mobile signal.

IndexedDB databases are tied to a specific domain and implement a "same domain policy," which means they can only be accessed by code running at the location where they were created. This means that a malicious site cannot reach into the data of any other site for nefarious purposes. Although IndexedDB databases are safe(r) from hacking, they are still vulnerable to user manipulation.

The user is always in control of the data on their computer, and they can easily wipe out any IndexedDB database. This means that care should always be taken to build proper synchronization routines into web software to rebuild the database in the event of corruption or deletion. It also means that the IndexedDB database should never be the primary source of information for a website or application. It is a caching tool to provide additional user convenience, but not a substitute for a proper data storage system.

Saving User-Generated Data

As users interact with our sites and applications, they will create data that we wish to store. Many times, we will be working with a server-based remote API that provides data, and in many cases we will store the data a user creates on our server. For example, if a user is writing blog posts, we will likely store the blog post on the server and not worry about providing a local cache of that blog post. However, there are times when we want to cache information generated by the user for quick use within our application.

Imagine an ecommerce experience where the user is adding items to a shopping cart. We might wish to provide a panel in our interface that can slide out to show the items the user has selected. Of course, we would be saving the shopping cart on the server so we could retrieve it if the user logs in from another device, but we might also want to store the shopping cart items in the browser's `localStorage` to allow quick and easy access to this information.

Storing and Retrieving Data

If we are already maintaining an Array called `shoppingCart` then we can easily just save that value to `localStorage` and then retrieve it from `localStorage` when we need it again. Assuming we use a tool such as `vue-ls` to help us work with `localStorage`, here is what that might look like.

First, we would need to configure the use of `vue-ls` in our `main.js` file where we define our `Vue` instance:

```
import VueLocalStorage from 'vue-ls';

let options = {
  namespace: 'catalog__'
};

Vue.use(VueLocalStorage, options);
```

We import the `vue-ls` module for use in this component. Then we define an `options` object, which only contains one property: `namespace`. This is a name that will be prepended to any data we save in the object. In some very complex situations, it would be helpful to use this namespace to keep data properly delineated. Finally, the `Vue.use(VueLocalStorage, options)` command makes `vue-ls` available as `this.$ls` in our components. We can use it in any of our components.

Inside a component, we could have code like this to add items from the catalog into a shopping cart.

```
addItem: function (item) {
  if (this.shoppingCart.indexOf(item) === -1) {
    this.shoppingCart.push(word);
    this.$ls.set('shoppingCart', this.shoppingCart);
    console.log(`Added ${item} to shoppingCart.`);
  } else {
    console.log('Item is already in shoppingCart.');
  }
},
removeItem: function (item) {
  this.shoppingCart.splice(this.shoppingCart.indexOf(item), 1);
  this.$ls.set('shoppingCart', this.shoppingCart);
}
```

We use the `get` and `set` methods to work with the values in `localStorage`. In the example above, we see the methods for `addItem` and `removeItem` are using `this.$ls.set` to set the updated values in `localStorage` at the same time as the value is set on the `this.shoppingCart` object.

We can look at this next example to see how we can initialize the `shoppingCart` object when we load a component:

```
created () {
  if (this.$ls.get('shoppingCart')){
    this.shoppingCart = this.$ls.get('shoppingCart');
  } else {
    this.$ls.set('shoppingCart', this.shoppingCart);
  }
}
```

In this example we can see that we have defined a `created` function on our component, which will be executed when the component is loaded. It will try to get the `shoppingCart` value from `localStorage`. If there is no `shoppingCart` value in `localStorage`, then the `this.$ls` module will return `undefined`, which evaluates to `false`. So if there is no `shoppingCart` value in `localStorage`, then we initialize that value to the `this.shoppingCart` value. If there *is* a value for `shoppingCart` in `localStorage`, then we use that value as our starting point.

The `vue-ls` module handles converting any data objects into Strings and then rendering them back into data objects when we retrieve them. This saves us several repetitive lines we would need to use whenever we saved/retrieved items from storage, and it allows us to have very simple statements to handle the `get` and `set` of data.

Passing Values Between Components

In Vue.js we have a one-way data flow: Properties are passed from parent components to child components. Child components can emit an event trigger, but they (normally) cannot sync data between values in the parent and child components. This setup seems clunky at first, but once we have a means of storing data in a single place that is accessible to all of the components in our application, we have an easier time.

We can imagine all kinds of situations where components could modify the same value in `localStorage` and then just emit the signal that the `localStorage` value has changed. This would prompt parent components to re-render their lists and allow everything to stay in sync across the system.

This can also be very helpful in other situations where, for example, we might have a component that handles the presentation of site "preferences" to the user. These preferences could be written into storage of some kind and then read by the other components that would use those preferences. For example, a user could select to always see english subtitles on videos, and the component responsible for embedding videos could check that preference value and properly modulate the display of the video for the user.

Now that we have a readily available and easy way of passing data between parts of our application, we can explore all kinds of interface and workflow possibilities. We should never feel compelled to cram a bunch of features into a single component because we cannot figure out how to share data properly between more targeted components. We could improve several previous projects we've seen in this book by using these techniques to refactor and break apart large components. (Take a look back at the `WordSearch` project in Section 12 for a good example of how this technique could allow us to have a nicer organization within that project.)

Caching Data from APIs

When we work with APIs we often run up against API "rate limits." Most API providers do not want developer applications to hit their API endpoints more than necessary. Extra requests to an API create additional delay for our users, and they potentially impact the performance of the API server, causing delays for other users as well. It is both neighborly and pragmatic to do what we can to minimize the number of API requests that we make to a server.

We can reduce the number of API calls we make to a server by caching responses. When working with RESTful APIs, each unique request can be cached. Depending on the API, we might not expect the data to change very often, so we can usually cache the data returned by the API for some period.

For example, an API that provides access to an encyclopedia of data (such as all the Pokémon or everything about Star Wars) is unlikely to update very often. Most weather APIs publish a refresh window that is 15 minutes or more. Whatever we are building, there is a good chance that we could delay the refresh of data by some amount of time.

This is especially relevant when we put an API call on a `created()` function in our components. These components are designed to show data, so they must request the data when they are instantiated, but they do not always need to make a call to the API server. If we consider how many times we might refresh our page in the browser during development, we can see how quickly we could eat up any API rate limit that might apply.

Of course, storing API data isn't our only problem. We also need to be able to know when our data was cached and clear it if it gets too old. This is often referred to as telling the difference between a "fresh" and "stale" cache. For user-generated data, we want to keep the information around forever. But for API data, we want to expire the cache when needed.

Caching vs. Browser Cache

It's worthwhile to note that when we discuss caching throughout this section, we are talking about the general concept of storing data that will be referenced again later. This is the definition of the term "caching" as we mean it. A "cache" is some store of data that can be referenced by an application. This is a general concept in computing, and we find caches of data stored all over our systems for use by all different software.

In the browser, we also have a "browser cache," which we think about often because we must "clear the cache" when we have issues with web pages. The reason we "clear" the cache is to delete the stored data, which in the case of web pages can sometimes cause problems (especially as we develop new features). This browser cache is used to speed up websites in a variety of ways, so it will do all kinds of things to try to store requests and avoid repeating them. This is helpful, but it's not what we are working with when we use cookies, localStorage, sessionStorage, or IndexedDB.

Determine How Often

We must figure out how often our API needs to refresh. If we are making calls to create new data on a system (such as submitting a new blog post), then obviously that call needs to be made whenever the user clicks "save." But if we are making calls to retrieve weather data, we do not need to refresh any given API call for at least 15 minutes.

Each data set will have a different refresh rate depending on what data it is using. Each API will also have a different rate at which they publish new information. Be sure to consult the API documentation and use some well-reasoned consideration to determine an interval for caching API data.

Keep in mind that times in JavaScript are generally calculated using `Date.now()`, which returns the number of milliseconds since the Unix epoch began. This number is precise (down to the millisecond), but it requires us to think in milliseconds when we are determining lengths of time. Because of this, we must calculate the times we need to compare.

For example, if we had a timestamp of `1510013577637`, we could check to see if 15 minutes has gone by like this:

```
let timestamp = 1510013577637;
let cacheExpiry = 15 * 60 * 1000; // 15 minutes in milliseconds

if ( Date.now() < (timestamp + cacheExpiry) ) {
    console.log('Do not refresh cache.');
} else {
    console.log('Refresh cache. 15 minutes has passed.');
}
```

We calculate the 15 minute time difference by multiplying `15 * 60` to get the number of seconds, and then that number times `1000` to get the number of milliseconds. (There are 1000 milliseconds in a second.) This kind of calculation allows us to easily determine the difference in time and make decisions based on that difference. If we needed to do more complex calculations (like number of days) we could calculate new `Date` objects and use the methods provided there to add/subtract days, weeks, or months. But we usually cache data for minutes or hours in our web applications.

Make an Identifier

When we make an API request, or when we just store user data, we often need to construct a unique key. These are sometimes called "slugs" in web development lingo: a "safe" term that indicates one piece of data in the system. We use slugs to indicate blog posts, or items, or users, etc. The slug is a unique ID, and regardless of what we call it, that is the key piece of information we need to make caching work in our application.

Sometimes it's easy to make a unique ID. If I have an API to an endpoint that looks up a city's weather by City ID, then I can likely use the City ID as the unique identifier for the cache. There are many situations where we are looking up data by a single ID or a set of latitude and longitude coordinates, and those things are relatively easy to put together into a unique ID.

In other cases, we must combine several pieces of information into a unique ID. Let's once again consider the `WordSearch` project from the previous section of the book. In that application, we make a call to the Datamuse API, and it would be nice to cache those queries. But those queries have several data points the user can modify.

In order to cache those queries, we need to assemble a unique ID inside the `findWords` method:

```
let queryID = `${this.phrase}-${this.soundsLike}-${this.startLetter}-${this.endLetter}`;
```

We have simply mashed together the unique elements into a query ID. We can then use that to establish the identifier for the cached data in `localStorage`. Care needs to be taken when determining a unique ID for any cache we try to store, but there is always a way to indicate a unique query.

Implement Some Cache Checking

We can use more complex tools for caching API requests, and if we were writing an application with many requests then it would be recommended to use a more robust tool for caching API requests. But for our purposes here, we can manually manage our API requests and avoid making those requests if we prefer. To help us a little bit, we will rely on the `expiry` ability of the `vue-ls` module.

The `vue-ls` module allows us to specify an expiration time for information we store. This allows us to set the expiration time when we store the data and then we don't need to worry about clearing it later. We can set an expiry time in milliseconds that will allow us to expire our cached information without any additional effort.

We can make the following changes to the `WordSearch` project:

```
findWords: function () {
  this.results = null;
  let queryID = `${this.phrase}-${this.soundsLike}-${this.startLetter}-${this.endLetter}`;
  let cacheExpiry = 24 * 60 * 60 * 1000; // 24 hours cache duration
  console.log('queryID: ' + queryID);
  let cachedQuery = this.$ls.get(queryID);
  if (cachedQuery) {
    console.log('Previous query cache detected.');
    this.results = cachedQuery;
  } else {
    console.log('No cache detected for: ' + queryID);
    this.fetchAPIResults();
  }
},
fetchAPIResults: function() {
  this.results = null;
  let queryID = `${this.phrase}-${this.soundsLike}-${this.startLetter}-${this.endLetter}`;
  let cacheExpiry = 24 * 60 * 60 * 1000; // 24 hour word search cache.
  axios.get('https://api.datamuse.com/words', {
    params: {
      ml: this.phrase,
      sl: this.soundsLike,
      sp: `${this.startLetter}*${this.endLetter}`
    }
  })
  .then( response => {
    this.$ls.set(queryID, response.data, cacheExpiry); // Cache the API response for 24 hours
    console.log('Cache created for: ' + queryID);
    this.results = response.data;
  })
  .catch( error => {
    this.errors.push(error);
  })
}
```

We have separated the API call out of the `findWords` method, and moved it to the `fetchAPIResults` method. The `findWords` method now tries to load the data from the cache, and if it can find no cache then it will execute the `fetchAPIResults` method. The `fetchAPIResults` method now also saves the results into `localStorage` before it sets `this.results` equal to `response.data`. We are applying a 24 hour expiry to the information stored in `localStorage`. After 24 hours, the data will no longer be available in `localStorage`, and the API request will happen again.

This kind of caching will greatly reduce the number of queries we send to the API server. Many API services require some form of request caching on the application side in order to minimize the load on the server itself. This not only helps the API server, but it also saves our users from repetitive waits as they move around inside our application.

Quiz: Caching Data

Please enjoy this self-check quiz to help you identify key concepts, points, and techniques discussed in this section.

Visit Quiz Online

The quiz on this page has been removed from your PDF or ebook format. You may take the quiz by visiting this book online.

Project: Caching Data

This project uses the Caching Data repository as a starting point. Please fork and clone this repository to your preferred work environment.

For this project, we will add some user information storage and caching to the weather application we refactored a couple sections ago. We willwork with a weather application that could have come out of that prior refactoring experience.

The weather application has three major views: `CitySearch`, `CurrentWeather`, and `Forecast`. These three views use several child components to display all of their information. When we first open the repository, we can notice that there is no way to save any data, and the site is making many requests to the weather API.

In order to improve the performance and user experience in our weather app, we will add the ability for users to save favorite cities for easy viewing when they return to the application. We will also cache our API queries so we do not risk running into a rate limit, and so our users do not have to wait for information to load as often.

These changes will dramatically enhance the utility and speed of our application. To accomplish these changes, we will cache data into `localStorage` using the `vue-ls` module. This module helps us manage our stored information more efficiently and makes it a snap to provide expiration times on information.

In order to complete this project, we will edit several files in the repository. Look for the `TODO` notes in the project files for guidance and indications of how we can accomplish our goals.

NOTE: Before we start work on this, it's crucial to obtain an APPID from OpenWeatherMap.org.

Review the Requirements

In order to successfully complete this project, we must fulfill the following requirements.

- Sign up to OpenWeatherMap.org and generate an API Key.
- Paste your API Key (which will be used as the `APPID` parameter) into the appropriate locations in the `CitySearch.vue`, `CurrentWeather.vue`, and `Forecast.vue` files.

`main.js`

- Add the base configuration for `vue-ls`.

`CitySearch.vue`

- Add the `FavoriteCities` component as a child to the `CitySearch` component (using proper imports, etc.).
- Add logic to the `created` function to initialize `this.favorites` to the value of the `favoriteCities` object in `localStorage`.
- Add logic to the `saveCity` function to update the `favoriteCities` cache in `localStorage`.
- Add logic to properly cache the API request in the `getCities` method (with proper label and expiry time).

`FavoriteCities.vue`

- Add logic in the `removeCity` method to remove the city from the `this.favoriteCities` array.
- Add logic to the `removeCity` method to remove the city from `localStorage`.

`CurrentWeather.vue`

- Add logic to properly cache the API request in the `created` function (with proper label and expiry time).

`Forecast.vue`

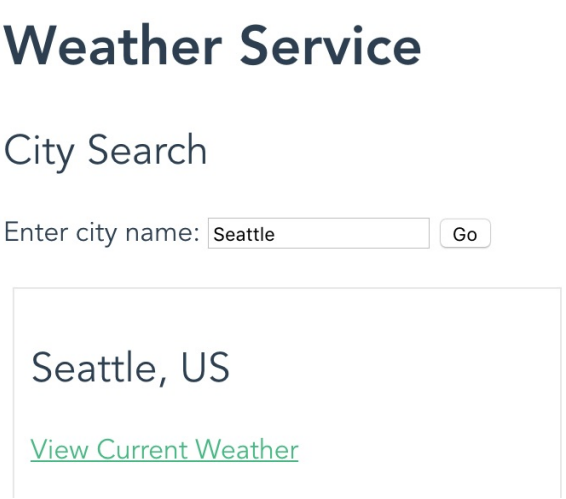
- Add logic to properly cache the API request in the `created` function (with proper label and expiry time).

Working the Project

In order to complete this project, we will edit several files in this repository. We begin with getting an API key for use on this project.

Make an OpenWeatherMap.org API Key

To get the project running, we need to make an account on OpenWeatherMap.org and generate an API key. Once we have created an account, the API Keys can be found under our Account page (located here). Create a new API Key and then open the project repository in a preferred editor. We must find the `YOUR_APPID_HERE` placeholder in the `src/common/api.js` file and update it.



Working home screen

Once we've replaced that information the project should become operational. Run `npm run dev` and verify that the project works. Once we've made sure the project is working, we can configure Vue LocalStorage for use in our application.

Configuring Vue LocalStorage

To add the `vue-ls` module to our project, we must import it in our `src/main.js` file and tell our `Vue` instance that we want to use it. We do this with these lines of code:

```
import VueLocalStorage from 'vue-ls';

let options = {
  namespace: 'weather__'
};

Vue.use(VueLocalStorage, options);
```

First, we have our import statement. This module has already been added to our project by running `npm install --save vue-ls`, so it can be imported like this. Then, we define an `options` object, which only has one property: `namespace`. We then execute a `Vue.use()` command to let our `Vue` instance know we are using the `VueLocalStorage` module (aka `vue-ls`).

Now we can access `this.$ls` in any component we use in our application.

Adding the `FavoriteCities` Component

In the `src/components/CitySearch.vue` file, we must add the `FavoriteCities` component as a child component. This requires us to import the component, then to define it in the `components` object, and finally to add a tag to our template where the component should be displayed.

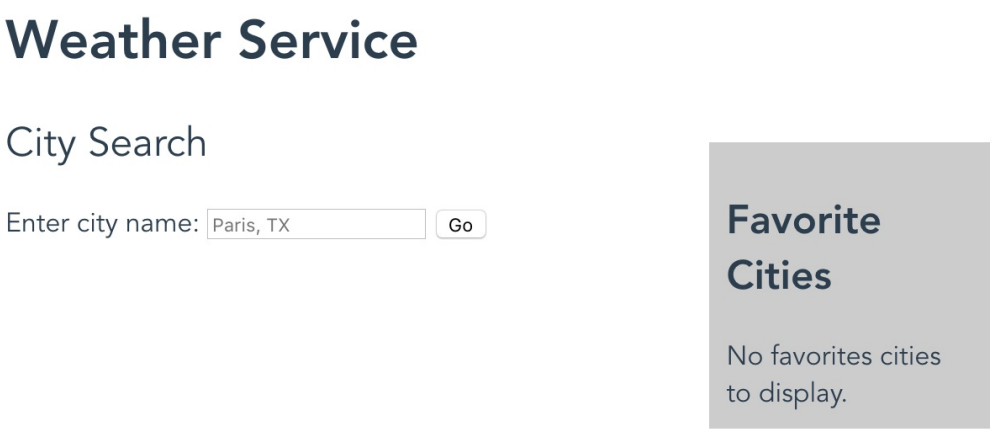
The logic changes to add the `FavoriteCities` component look like this:

```
// ... previous imports ... //
import FavoriteCities from '@/components/FavoriteCities';


export default {
  name: 'CitySearch',
  components: {
    'weather-summary': WeatherSummary,
    'weather-data': WeatherData,
    'load-spinner': CubeSpinner,
    'message-container': MessageContainer,
    'favorite-cities': FavoriteCities
  },

  // ... more code ... //
```

We can see how the `import` statement and the `components` object have been updated. In the template, the tag is straightforward to drop in:

```
<favorite-cities v-bind:favoriteCities="favorites"></favorite-cities>
```

We use the tag and bind the `favorites` value from the `CitySearch` component to the `FavoriteCities` component. At this point, we should see our component show up with no cities listed.



Empty Favorites Listing

Make Favorite Cities Work in `CitySearch`

In order to make our favorite cities listing work in our application, we must actually save data when the user clicks the "Save City to Favorites" button. In order to make it work, we must update two parts of our code. First, we need to allow users to save a city. There is a `saveCity` method defined for us, so we can fill in the logic there:

```
saveCity: function (city) {
  this.favorites.push(city);
  this.$ls.set('favoriteCities', this.favorites);
}
```

To save a city, we simply `push` the `city` object into the `this.favorites` array. We then cache these favorites using `this.$ls.set()`. Because we only have one list of `favoriteCities` we can just specify the name of the cache label directly, and we do not need to set a cache expiration time because we want this data to persist forever.

Now we should be able to see our information updating on the screen and in our devtools. Open the "Application" tab in our devtools panel and select our `localhost` domain under `localStorage`. We should see all of the values updating in all the right places.

Storing cities in localStorage

Make Favorite Cities Removable

Now that we have made it possible to save cities in both component data and localstorage, we should make it possible for our users to remove cities. Since we are syncing our data through the caching system using `localStorage`, we can remove cities in the `FavoriteCities` component, where it's easier to respond to the user's click action.

A `removeCity` method is provided for us, so we will place our logic there:

```
removeCity: function (city) {
  let cityIndex = this.favoriteCities.indexOf(city);
  this.favoriteCities.splice(cityIndex, 1);
  this.$ls.set('favoriteCities', this.favoriteCities);
}
```

In this example, we have used `indexOf` on the `favoriteCities` array to find the index of the city object. Once we have that index, we can use the `splice()` command to remove only that object from the `favoriteCities` array. Then, we can save the `favoriteCities` cache again. The next time the page loads, it will load the proper cache.

Once we have that code in place, we can add and remove values at will.

Cache `CitySearch` API Requests

Now that we've enhanced the application with the ability to save cities to a list of favorites, we can provide some performance enhancements by caching API requests so we do not make as many. In order to set up the caching of the API requests, we will tackle the same few tasks each time:

1. We will make a cache label (called `cacheLabel`) to represent the unique query. This label will allow us to cache individual queries.
2. We will create an expiration time (called `cacheExpiry`) to tell the system how long it should store the query cache.
3. We will wrap our API request in a conditional that will check for the existence of a cache. If it finds cached data, it will use that data. If the data has expired, or has never before been requested, then it will make the request.
4. When a request is made, the data will be cached.

For the next three sets of changes we are essentially implementing the same thing. This is what it looks like:

```
getCities: function () {
  this.results = null;
  this.showLoading = true;

  let cacheLabel = 'citySearch_' + this.query;
  let cacheExpiry = 15 * 60 * 1000; // 15 minutes

  if (this.$ls.get(cacheLabel)){
    console.log('Cached query detected.');
    this.results = this.$ls.get(cacheLabel);
    this.showLoading = false;
  } else {
    console.log('No cache available. Making API request.');
    API.get('find', {
      params: {
          q: this.query
      }
    })
    .then(response => {
      this.$ls.set(cacheLabel, response.data, cacheExpiry);
      console.log('New query has been cached as: ' + cacheLabel);
      this.results = response.data;
      this.showLoading = false;
    })
    .catch(error => {
      this.messages.push({
        type: 'error',
        text: error.message
      });
      this.showLoading = false;
    });
  }
}
```

We can see that the `cacheLabel` and `cacheExpiry` values are assigned right away. These will be used in various conditions, so they are necessary and worthwhile to compute. (Remember that the `cacheExpiry` time needs to be in milliseconds.) Then, we see the first conditional. If a value does not exist in `localStorage` it will be returned as `undefined`. Since `undefined` is a `false` value, this conditional will be true only if valid data is stored in the cache that matches the `cacheLabel`.

If data is found, we log a message to the console and then set the results equal to the cache data. These results can be processed just like the API results, so our application will show the information to the user properly.

If no data is found (meaning the cache has either expired or the data has never been requested before), then we log a message to console to say that we need to make the API request. We execute the same API request as before, but in the `then` clause, we add a `this.$ls.set()` statement to save the response into the `localStorage` cache. The data is stored with the `cacheLabel` we specified, and the expiration time is set to the `cacheExpiry` value. Once we have cached the data, we log another statement to the console so we can track that everything has happened the way we expect.

We should now be able to repeat queries in our console and see that a new query causes a new API request, but a repeated query uses cached data. We can also verify that API requests are (or are not) happening by watching the "Network" tab of our devtools while we execute searches.

The functionality of this page should be the same, with the only difference being the speed with which repeated searches are executed. Users will perceive our application as being much faster thanks to these caching changes.

Cache `CurrentWeather` and `Forecast` API Requests

We will make the same changes to cache the API requests in the `src/components/CurrentWeather.vue` and `src/components/Forecast.vue` files. In each case we must create `cacheLabel` and `cacheExpiry` values, and then we will use the same kind of conditional to check for the value in `localStorage` and perform the API request if it is not found.

Rather than repeating these same structures multiple times on the page, refer to the full file details below for more precise examples of what this process looks like in each file.

Wrapping Up

Once we have made all of our changes, we have an app that is more functional and faster for users. Most people would agree that makes any app better. The following files have been changed:

`main.js`

```
import Vue from 'vue'
import App from './App'
import router from './router'
import VueLocalStorage from 'vue-ls';

let options = {
  namespace: 'weather__'
};

Vue.use(VueLocalStorage, options);

Vue.config.productionTip = false

new Vue({
  el: '#app',
  router,
  template: '<App/>',
  components: { App }
})
```

`CitySearch.vue`

```
<template>
  <div>
    <favorite-cities v-bind:favoriteCities="favorites"></favorite-cities>
    <h2>City Search</h2>
    <message-container v-bind:messages="messages"></message-container>
    <form v-on:submit.prevent="getCities">
```

```
        <p>Enter city name: <input type="text" v-model="query" placeholder="Paris, TX"> <button type="submit">Go
</button></p>
    </form>
    <load-spinner v-if="showLoading"></load-spinner>
    <ul class="cities" v-if="results && results.list.length > 0">
      <li v-for="city in results.list">
        <h2>{{ city.name }}, {{ city.sys.country }}</h2>
        <p><router-link v-bind:to="{ name: 'CurrentWeather', params: { cityId: city.id } }">View Current Weather
</router-link></p>

        <weather-summary v-bind:weatherData="city.weather"></weather-summary>

        <weather-data v-bind:weatherData="city.main"></weather-data>
        <p><button class="save" v-on:click="saveCity(city)">Save City to Favorites</button></p>
      </li>
    </ul>
  </div>
</template>

<script>
import {API} from '@/common/api';
import WeatherSummary from '@/components/WeatherSummary';
import WeatherData from '@/components/WeatherData';
import CubeSpinner from '@/components/CubeSpinner';
import MessageContainer from '@/components/MessageContainer';
import FavoriteCities from '@/components/FavoriteCities';


export default {
  name: 'CitySearch',
  components: {
    'weather-summary': WeatherSummary,
    'weather-data': WeatherData,
    'load-spinner': CubeSpinner,
    'message-container': MessageContainer,
    // TODO: Add FavoriteCities child component here
    'favorite-cities': FavoriteCities
  },
  data () {
    return {
      results: null,
      query: '',
      showLoading: false,
      messages: [],
      favorites: []
    }
  },
  created () {
    if (this.$ls.get('favoriteCities')){
      this.favorites = this.$ls.get('favoriteCities');
    }
  },
  methods: {
    saveCity: function (city) {
      this.favorites.push(city);
      this.$ls.set('favoriteCities', this.favorites);
    },
    getCities: function () {
      this.results = null;
      this.showLoading = true;

      let cacheLabel = 'citySearch_' + this.query;
      let cacheExpiry = 15 * 60 * 1000;

      if (this.$ls.get(cacheLabel)){
        console.log('Cached query detected.');
        this.results = this.$ls.get(cacheLabel);
        this.showLoading = false;
      } else {
```

```
        console.log('No cache available. Making API request.');
        API.get('find', {
          params: {
              q: this.query
          }
        })
        .then(response => {
          this.$ls.set(cacheLabel, response.data, cacheExpiry);
          console.log('New query has been cached as: ' + cacheLabel);
          this.results = response.data;
          this.showLoading = false;
        })
        .catch(error => {
          this.messages.push({
            type: 'error',
            text: error.message
          });
          this.showLoading = false;
        });
      }
    }
  }
}
</script>

<style scoped>
.errors li {
  color: red;
  border: solid red 1px;
  padding: 5px;
}
h1, h2 {
  font-weight: normal;
}

ul {
  list-style-type: none;
  padding: 0;
}
li {
  display: inline-block;
  width: 300px;
  min-height: 300px;
  border: solid 1px #e8e8e8;
  padding: 10px;
  margin: 5px;
}
a {
  color: #42b983;
}
</style>
```

CurrentWeather.vue

```
<template>
  <div>
    <h2>Current Weather <span v-if="weatherData"> for {{ weatherData.name }}, {{weatherData.sys.country }}</span
></h2>
    <message-container v-bind:messages="messages"></message-container>
    <p>
      <router-link to="/">Home</router-link> |
      <router-link v-bind:to="{ name: 'Forecast', params: { cityId: $route.params.cityId } }">View 5-Day Foreca
st</router-link>
    </p>
    <load-spinner v-if="showLoading"></load-spinner>
    <div v-if="weatherData">
```

```
      <weather-summary v-bind:weatherData="weatherData.weather"></weather-summary>

      <weather-data v-bind:weatherData="weatherData.main"></weather-data>

    </div>

  </div>
</template>

<script>
import {API} from '@/common/api';
import WeatherSummary from '@/components/WeatherSummary';
import WeatherData from '@/components/WeatherData';
import CubeSpinner from '@/components/CubeSpinner';
import MessageContainer from '@/components/MessageContainer';

export default {
  name: 'CurrentWeather',
  components: {
    'weather-summary': WeatherSummary,
    'weather-data': WeatherData,
    'load-spinner': CubeSpinner,
    'message-container': MessageContainer
  },
  data () {
    return {
      weatherData: null,
      messages: [],
      query: '',
      showLoading: false
    }
  },
  created () {
    this.showLoading = true;

    let cacheLabel = 'currentWeather_' + this.$route.params.cityId;

    let cacheExpiry = 15 * 60 * 1000;

    if (this.$ls.get(cacheLabel)) {
      console.log('Cached value detected.');
      this.weatherData = this.$ls.get(cacheLabel);
      this.showLoading = false;
    } else {
      console.log('No cache detected. Making API request.');
      API.get('weather', {
        params: {
            id: this.$route.params.cityId
        }
      })
      .then(response => {
        this.$ls.set(cacheLabel, response.data, cacheExpiry);
        this.showLoading = false;
        this.weatherData = response.data;
      })
      .catch(error => {
        this.showLoading = false;
        this.messages.push({
          type: 'error',
          text: error.message
        });
      });
    }
  }
}
</script>

<style scoped>
.errors li {
```

```
  color: red;
  border: solid red 1px;
  padding: 5px;
}
h1, h2 {
  font-weight: normal;
}

ul {
  list-style-type: none;
  padding: 0;
}
li {
  display: inline-block;
  width: 300px;
  min-height: 300px;
  border: solid 1px #e8e8e8;
  padding: 10px;
}
a {
  color: #42b983;
}
</style>
```

Forecast.vue

```
<template>
  <div>
    <h2>Five Day Hourly Forecast <span v-if="weatherData"> for {{ weatherData.city.name }}, {{weatherData.city.
country }}</span></h2>
    <message-container v-bind:messages="messages"></message-container>
    <p>
      <router-link to="/">Home</router-link> |
      <router-link v-bind:to="{ name: 'CurrentWeather', params: { cityId: $route.params.cityId } }">Current Wea
ther <span v-if="weatherData"> for {{ weatherData.city.name }}, {{weatherData.city.country }}</span></router-li
nk>
    </p>

    <ul v-if="weatherData" class="forecast">
      <transition-group name="fade" tag="div" appear>
        <li v-for="forecast in weatherData.list" v-bind:key="forecast.dt">
          <h3>{{ forecast.dt|formatDate }}</h3>
          <weather-summary v-bind:weatherData="forecast.weather"></weather-summary>
          <weather-data v-bind:weatherData="forecast.main"></weather-data>
        </li>
      </transition-group>
    </ul>
    <load-spinner v-if="showLoading"></load-spinner>
  </div>
</template>

<script>
import {API} from '@/common/api';
import WeatherSummary from '@/components/WeatherSummary';
import WeatherData from '@/components/WeatherData';
import CubeSpinner from '@/components/CubeSpinner';
import MessageContainer from '@/components/MessageContainer';

export default {
  name: 'Forecast',
  components: {
    'weather-summary': WeatherSummary,
    'weather-data': WeatherData,
    'message-container': MessageContainer,
    'load-spinner': CubeSpinner
  },
  data () {
```

```
    return {
      weatherData: null,
      messages: [],
      query: '',
      showLoading: false,
      messages: []
    }
  },
  created () {
    this.showLoading = true;

    let cacheLabel = 'forecast_' + this.$route.params.cityId;

    let cacheExpiry = 15 * 60 * 1000;

    if (this.$ls.get(cacheLabel)) {
      console.log('Cached value detected.');
      this.weatherData = this.$ls.get(cacheLabel);
      this.showLoading = false;
    } else {
      console.log('No cache detected. Making API request.');
      API.get('forecast', {
        params: {
            id: this.$route.params.cityId
        }
      })
      .then(response => {
        this.$ls.set(cacheLabel, response.data, cacheExpiry);
        this.showLoading = false;
        this.weatherData = response.data;
      })
      .catch(error => {
        this.showLoading = false;
        this.messages.push({
          type: 'error',
          text: error.message
        });
      });
    }
  },
  filters: {
    formatDate: function (timestamp){
      let date = new Date(timestamp * 1000);
      const months = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'O
ctober', 'November', 'December'];
      const weekdays = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday'];
      //let weekday = date.getDay();
      let daynum = date.getDate();
      let month = date.getMonth();

      let hour = date.getHours();
      if (hour === 12) {
        hour = 'Noon';
      } else if (hour === 0) {
        hour = 'Midnight';
      } else if (hour > 12) {
        hour = hour - 12 + 'PM';
      } else if (hour < 12) {
        hour = hour + 'AM';
      }
      //let year = date.getFullYear();
      return `${ months[month] } ${ daynum } @ ${ hour }`;
    }
  }
}
</script>

<style scoped>
.fade-enter-active, .fade-leave-active {
```

```
  transition: opacity 1s
}
.fade-enter, .fade-leave-to {
  opacity: 0
}
h1, h2 {
  font-weight: normal;
}

ul {
  list-style-type: none;
  padding: 0;
}
li {
  display: inline-block;
  width: 200px;
  min-height: 300px;
  border: solid 1px #e8e8e8;
  padding: 10px;
  margin: 5px;
}

a {
  color: #42b983;
}
</style>
```

FavoriteCities.vue

```
<template>
    <ul class="favorite-cities">
        <li><h2>Favorite Cities</h2></li>
        <li v-if="favoriteCities.length < 1">No favorites cities to display.</li>
        <li v-for="city in favoriteCities">
          <router-link v-bind:to="{ name: 'CurrentWeather', params: { cityId: city.id } }">{{ city.name }}</rou
ter-link> <button v-on:click="removeCity(city)" class="remove">x</button>
        </li>
    </ul>
</template>

<script>
export default {
  name: 'FavoriteCities',
  data () {
    return {}
  },
  props: {
    favoriteCities: Array
  },
  methods: {
    removeCity: function (city) {
      this.favoriteCities.splice(this.favoriteCities.indexOf(city), 1);
      this.$ls.set('favoriteCities', this.favoriteCities);
    }
  }
}
</script>

<style scoped>
.favorite-cities {
  list-style-type: none;
  padding: 10px;
  background: #ccc;
  width: 25%;
  float: right;
}
.remove {
  font-size: 0.8rem;
```

```
    color: white;
    background: #AA0000;
    padding: 2px;
    cursor: pointer;
  }
  </style>
```

Note: We also need to add our `APPID` to the `src/common/api.js` file. Don't forget!

Build and Deploy

Once we've finished our work, we can build and deploy the project. This project has been configured to build to the `docs/` directory, so we can follow the same pattern we used before:

1. Execute the `npm run build` command to build the files into the `docs/` directory.
2. Commit all of our code.
3. Push the code up to GitHub.
4. Go into the repository settings and set the GH Pages section to publish from the `docs/` directory.

The project should now be up and available to the public through GH Pages.

Stretch Goals

If we crave more challenge, we can attempt these additional goals.

- Add more preferences to the system, such as the ability to load a single "favorite" city when the page is first loaded (with no clicks or search required).
- Add the ability for users to specify their own label for the favorite cities (e.g. "home", "Aunt Barb's House", etc.).
- Add a query to another API service, such as flickr, to augment this information. Build the proper caching to make efficient use of that query, too.
- Add animated transitions to the information in this project.

Conclusion

As I mentioned in the introduction to the book, this is not intended to be a final step in our journey towards better web development. There are many more things to learn about every topic we explore in this book. Hopefully we now have the information we need to continue growing and learning as developers and humans.

We have explored fundamental concepts like modern application architecture, third-party data APIs, templating, event handling, and refactoring. We have explored Vue.js concepts including component composition, routing, visual sugar, and more. We have worked with a half-dozen different projects to get hands-on experience with all parts of the application code and features.

It's been a lot of work. Congratulations on your efforts. It might seem frustrating to feel like you've come so far, and yet there is still so much to learn. But this is the fate of the web developer: everything we work with is constantly in flux. We are never allowed to rest.

Ask any group of developers a single technical question and you're likely to receive many different answers. But ask a group of developers what is the most important thing to learn as a new developer and most of them will tell you: The most important thing to learn is how to learn.

Hopefully the process of exploring this book has helped us become better learners overall. Continue the process by exploring some of the resources mentioned in the appendices. Return to previous projects and work on stretch goals or other modifications. Seek out additional material to read and watch and engage with.

Most importantly, keep building things. Explore the many novice-friendly APIs listed in the API Suggestions Appendix. Combine various APIs to create unique and compelling experiences. Keep building and creating. Keep writing code.

Thank you for reading Practical JavaScript 2: Building Applications. I hope it has been an enjoyable and useful experience. I have enjoyed creating this book, and I look forward to sharing it with as many developers as possible. Thanks again for being one of those readers.

Glossary

The terms listed here will be highlighted throughout the text. Hover over them to see the definition as you are reading.

abstraction

The process of moving or organizing segments of program logic or data into modular components that can be referenced from multiple locations in the code.

API (Application Programming Interface)

A feature that allows for functions and data within one system to be accessed programmatically by another system. For example, we can use the Open Weather Map API to request weather data for thousands of cities around the world, and we can use that data within our own applications.

API end point

The URL used by an API to indicate where a specific feature or data set exists. For example, `http://api.example.com/v1/users` could indicate the `users/` endpoint of the `example.com` API.

backend

Having to do with the server-side of a web-based project. Often used in phrases such as "backend programming languages", which would be languages that are executed on the server (Python, Java, Ruby, PHP, etc.). The opposite of *frontend*.

breakpoint

An intentional stopping or pausing place in a program used for debugging purposes.

directive

A function used in a template to invoke some special behavior (e.g. setting an event listener, invoking a loop, or creating a conditional) used to process the template. Custom directives may be created by developers, and frameworks often provide a selection of default directives.

encapsulation

The process of hiding information or complex logic behind a developer-friendly interface.

DOM

The Document Object Model (DOM) is the JavaScript representation of the HTML structure and status in the browser software.

frontend

Having to do with the web browser. Frontend technologies include HTML, CSS, and JavaScript. The opposite of *backend*.

modifier

An extra command that can be appended to a directive to alter its behavior. For example, the `.prevent` modifier can be used with the `v-on` directive to "prevent defaults" on event handlers.

mustache syntax

The name for the Vue.js default template syntax, which uses double curly braces (`{{ }}`) to denote variable interpolation in the template. Mustache syntax is only used to output the value of a variable in a template, and never used with directives.

query string parameters

Values appended to the URL using the `?` to denote the beginning of the parameters, and the `&` to denote each individual key/value pair. These values are often used with `GET` requests to an API endpoint. For example, in the URL `http://api.example.com/search?q=test&order_by=rating` the values `q` and `order_by` are the parameters.

refactor

To re-organize and re-structure code for an improved developer experience without altering functionality.

router

The component of an application that determines what view the user should see based on the URL they request.

single page application

A JavaScript application that does not refresh the browser in order to show different "pages" of information. This avoids use of the browser's navigation and history mechanism, which requires those features to be implemented in JavaScript. Thus, many single page applications use a robust JavaScript framework.

software architecture

According to Wikipedia: Software architecture is "the high level structures of a software system, the discipline of creating such structures, and the documentation of these structures."

syntax

The "grammar" of a programming language: A set of rules that dictate the way symbols are used to construct the logic that will be interpreted to run a program.

template context

The "context" of a template refers to the set of information (Objects, Arrays, methods, etc.) that is used to process the template into the final output.

template engine

The software component responsible for processing templates and outputting the results. The engine typically defines rules about syntax and features of the templates, and frameworks can often swap out templating engines to suit the needs and preferences of developers.

template rendering

The process of computing the final output of a template. This involves combining data in the template context using the template instructions and HTML structures written in the template itself.

website architecture

When considering the way a website is built and functions, we often use the term "website architecture" to describe the way these structures go together.

Appendix A: Additional Resources for Learning

This page lists additional resources for continuing our learning about web applications in general and Vue.js applications in particular.

Vue.js Resources

- Vue.js Official Guide
- Vue.js Feed
- Vue.js Style Guide

Helpful References

- Vue.js API Reference
- Curated Vue.js Module Listing
- Mozilla Developers Network

Other Tutorials / Guides / Courses

- Learn Vue 2 Step by Step from Laracasts
- Intro to Vue.js from CSS Tricks
- Learning Vue.js from Lynda.com
- Vue.js: Building an Interface from Lynda.com
- Getting Started with Vue.js from Pluralsight
- Single Page Applications with Vue.js from Pluralsight

Appendix B: API Suggestions

This book includes several sections dedicated to using third-party APIs in order to have access to dynamic data that is useful to a user. There are many APIs in the world, but not all of them are easy to use or friendly towards JavaScript applications. Some APIs require complex authentication in order to use them which is beyond the scope of this book. Other APIs are designed only to be used by certain types of systems and might cause issues when used with a JavaScript application in the browser.

The following APIs have been reviewed for use with the kinds of code we cover in this book. They should provide decent fodder for experimentation and minimal barriers to entry.

No API Key Required

- JSON Placeholder API
- Datamuse Word API
- Star Wars API
- Pokémon API
- Yes/No API
- iTunes Search API
- Sunrise/Sunset API
- Random (fake) User Data API
- Game of Thrones API
- Location of International Space Station API
- Number of People in Space API

API Key Required

- Soundcloud API
- Open Weather Map API
- Flickr API
- Open Movie Database API
- Google Books API
- Spotify API
- The Movie Database API

Cross-Origin Proxies

Some of the APIs listed here do not allow direct requests from JavaScript due to Cross-Origin Resource Sharing (CORS) restrictions. CORS is a concept that keeps browsers more secure by limiting the ways that JavaScript can communicate with third-party servers, possibly without the user's knowledge. Some APIs allow developers to configure the domain their apps are hosted on, which then allows the API service to provide the correct CORS headers to allow communication directly through the API. Other API services, such as the Datamuse service, allow all domains to access their server, so there is no need to fuss with domain configurations.

When attempting to use an API that does not allow domain configurations necessary for CORS to allow the API requests, it is necessary to use a "cross-origin proxy." These proxies use a server-based application to pass requests through to the API service and then relay responses back to the user's browser. There are two cross-origin proxies we suggest for new developers attempting to experiment with APIs using JavaScript:

- Crossorigin.me
- Rashrewind.com

Either of these proxies will work in most cases, although they do function slightly differently, so some situations may warrant choosing one over the other. These proxies will only allow GET requests, so it is impossible to use some features of some APIs that require POST or PUT requests. In order to protect the security of our user's data, **we should never send sensitive or confidential information through these public proxies.**

The list of APIs above was largely drawn from Terence Eden's list here.