Ember.js Cookbook

Ember.js is an open source JavaScript framework that will make you more productive. It uses common idioms and practices, making it simple to create amazing single-page applications. It also lets you create code in a modular way using the latest JavaScript features.

This book provides in-depth explanations on how to use the Ember.js framework to take you from beginner to expert. You’ll start with some basic topics and by the end of the book, you’ll know everything you need to know to build a fully operational Ember application.

What this book will do for you...

- Skip the boilerplate code with Ember CLI generators
- Create a component with actions and events
- Set up a model with Ember Data using fixture data
- Create several different types of test cases and run them
- Manage and set up user authentication using Ember Simple Auth
- Add animated transitions to your app with Liquid Fire
- Set up a service and initializer with dependency injection
- Create a working chat application

Inside the Cookbook...

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

Bringing the power of Ember.js to your fingertips, Ember.js Cookbook is your companion for getting up and running with the Ember.js framework. The book is divided into two parts: a brief introduction to the Ember.js framework and a series of hands-on recipes that will help you create fully operational Ember applications.


Erik Hanchett Ember.js Cookbook

Arm yourself with over 65 hands-on recipes to master the skills of building scalable web applications with Ember.js


Free Sample
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Ember CLI Basics'
- A synopsis of the book’s content
- More information on Ember.js cookbook
About the Author

Erik Hanchett is a software developer, blogger, and perpetual student who has been writing code for over 10 years. He currently resides in Reno, Nevada, with his wife and two kids. He blogs about software development at ProgramWithErik.com.
Preface

Single-page, client-side JavaScript frameworks may possibly be the future of the Web. Client-side frameworks have matured a lot in the last few years. They've made creating web applications easier and more responsive. Ember.js has been one of the leading frameworks behind this movement.

This book provides in-depth explanations of how to use the Ember.js framework to take you from being a beginner to an expert. You'll start with the basics, and by the end of the book, you'll have a solid foundation in creating real-time web applications in Ember.

We'll begin by explaining key points on how to use the Ember.js framework and associated tools. You'll learn how to use Ember CLI effectively and how to create and deploy your application. We'll take a close look at the Ember object model and templates by looking at bindings and observers. We'll move on to Ember components and models and Ember Data. Next, we'll look at testing with integration and acceptance tests using QUnit. Afterward, we'll take a look at authentication and services and working with Ember add-ons. We'll explore advanced topics such as services and initializers and how to use them together to build real-time applications.

What this book covers

Chapter 1, Ember CLI Basics, shows you how to use the Ember CLI command-line tool and deal with upgrading and deploying your project.

Chapter 2, The Ember Object Model, demonstrates how to create Ember objects and instances and use bindings, mixins, and enumerables.

Chapter 3, Ember Templates, tells you how to use templates and template helpers.

Chapter 4, Ember Router, demonstrates how to set up your model and work with loading states and redirection.
Preface

Chapter 5, *Ember Controllers*, explains how to use properties and manage dependencies between controllers.

Chapter 6, *Ember Components*, covers passing properties, events, and actions.

Chapter 7, *Ember Models and Ember Data*, explains how to manipulate records and customize adapters.

Chapter 8, *Logging, Debugging, and Testing*, demonstrates how to create acceptance and unit tests as well as the Ember Inspector.

Chapter 9, *Real-Life Tasks with Ember.js*, discusses how to use services, authentication, Bootstrap, and Liquid Fire.


Chapter 11, *Real-Time Web Applications*, discusses how to use dependency injection, application initializers, run loops, and create add-ons.
In this chapter, we will cover the following recipes:

- Installing Ember CLI
- Creating your first project
- Exploring pods and the folder layout
- Asset compilation
- Dependency management
- Upgrading your project
- Deployment

**Introduction**

Ember CLI is a Node.js-based command-line interface tool designed for application programming in Ember.js. Since its creation, this tool has become the preferred method to create Ember applications.

Simply put, Ember CLI makes starting a new Ember application easy. In other frameworks, you might need to learn gulp or grunt as your task runners. In Ember CLI, this is all built-in for you. Having Ember CLI in the Ember.js ecosystem is a game changer and makes this framework stand out above the rest.

Ember CLI handles testing, compiling, and upgrading, and even has a built-in web server. Ember CLI not only generates boilerplate code for you, but it also integrates nicely with many testing frameworks. It has a robust add-on system that extends the functionality well beyond its current capabilities.
Installing Ember CLI

The installation of Ember CLI is essential to learning Ember and will be used throughout this book.

Getting ready

Before the installation of Ember CLI, we must have the Node Package Manager (npm) installed. npm is a package manager for JavaScript and is installed by default with Node.js.

You must install version 0.12 or later of Node.js for Ember CLI to run. If you can, try to install version 4.0.0 or higher. This is the preferred version.

Node.js is available in several major platforms including Windows, Mac, and Linux. There are several ways to install Node.js:

- **One-click installers**: Many platforms such as Windows and Mac have this available
- **Homebrew or MacPorts**: This is useful for Mac OS users
- **Download TAR file**: Download a TAR file of Node.js and extract
- **Install via the Linux package management system**: Yum, apt-get, or pacman can be used to install on a Linux environment

A one-click installer for Windows or Mac

This method is by far the easiest. To install node, you'll need to open the node website at http://nodejs.org/download. Click on the pkg, msi, or exe installer for Windows or Mac. Run it after it's downloaded.

Homebrew or MacPorts for Mac

If you already have Homebrew installed, just run the following command:

```bash
$ brew install node
```

On the other hand, if you are running MacPorts, you can use the port install command:

```bash
$ sudo port install nodejs
```

A TAR file

A TAR file is a type of archive file. To install node via a TAR, you will need to download the TAR file from the Node.js website and extract and install it. One way of doing this is to use curl.

I would only recommend this method if you are using a Linux distribution. If you are running on Linux, you'll need the right tools installed to compile from source. On Ubuntu, you'll need to install the build-essential and curl packages:

```bash
$ curl http://nodejs.org/dist/node-latest.tar.gz | tar xz --strip-components=1
$ ./configure
$ sudo make install
```

The Linux package manager

All major Linux distributions offer Node.js packages. In Ubuntu, you can simply use `apt-get`:

```bash
$ sudo apt-get install nodejs
```

In Fedora, you can use `yum`:

```bash
$ yum install nodejs npm
```

Check with your Linux distribution to find out more details on how to install packages such as Node.js. Be aware that some distributions might offer outdated versions of Node.js. In this case, I would recommend that you use the Node Version Manager (NVM) installation method that will be discussed later.

Test installation

To test your installation, run the `-v` command:

```bash
$ node -v
$ npm -v
```

This will show the current installed version. Keep in mind that you must run v0.12 or above to run Ember CLI. If possible, try to run v4.0.0 or above.

The NVM is a bash script that helps manage multiple active Node.js versions. NVM offers a very simple command-line interface to install any version of Node.js without having to visit the Node.js website. It separates each installation making it very easy to change between versions. I would recommend most beginners on Mac and Linux to run this. You can download NVM at https://github.com/creationix/nvm.
How to do it...

We'll need to use `npm` to install Ember CLI. We'll install it globally with the `-g` option so that it can be run anywhere from the command line.

1. Open the command prompt and type the following command:
   
   ```
   $ sudo npm install -g ember-cli
   ```
   
   If NVM was installed, you don't need `sudo` at the start of the command.

2. After Ember CLI is installed, we'll need to download **Bower**. Bower is a package manager for client-side programming and another essential component of Ember.js. Node.js and npm must be installed before beginning the installation of Bower. We'll be using Bower to install all our client-side libraries:
   
   ```
   $ sudo npm install -g bower
   ```
   
   Similar to the last command, you don't need `sudo` at the start of the command if Node.js was installed via NVM.

3. The last step is to install **PhantomJS**. PhantomJS is a scripted headless browser used to automate and test web pages. It's preferred by Ember CLI and needs to be installed:
   
   ```
   $ npm install -g phantomjs
   ```

4. If you are on Windows, install the Ember CLI Windows tool:
   
   ```
   $ npm install ember-cli-windows -g
   ```

5. Once installed, this tool can be run in any project directory:
   
   ```
   $ ember-cli-windows
   ```


### Working with Windows

Build times on Windows can be longer than Mac or Linux. The Ember CLI Windows tool can help speed up and optimize build performance. Just run it in the project directory. You can also download it as an add-on instead.

Another way to help with performance is to always run PowerShell/CMD with elevated privileges. Otherwise, performance issues and errors might occur. Lastly, try to use npm version 3 or higher. You may run into issues with long file paths with older versions in Windows.

Another handy tip is as follows:
Optional: Install Watchman

Watchman is a file-watching service for OS X and UNIX-like operating systems. It was developed by Facebook and is a more effective way for Ember CLI to watch project changes. If it's not installed, Ember CLI will fall back to using NodeWatcher. NodeWatcher is more error-prone and has trouble observing large trees. Install Watchman if your platform supports it. To download and configure Watchman, visit https://facebook.github.io/watchman/.

How it works...

Ember CLI is written in Node.js and can be installed via npm. The tool interprets commands from the user to help create an Ember.js application. Each command from the user is looked up and then executed. Ember CLI relies on several other dependencies including Bower, Lodash, Broccoli, and Babel, to name a few.

There's more...

Let's take a look at commands and aliases.

Commands

Once Ember CLI is installed, we'll have access to several commands. Here is a short list of some of the more important ones:

<table>
<thead>
<tr>
<th>Command</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ember</td>
<td>This prints a list of available commands</td>
</tr>
<tr>
<td>ember new &lt;name-of-app&gt;</td>
<td>This creates a directory called <code>&lt;name-of-app&gt;</code> and creates the application structure</td>
</tr>
<tr>
<td>ember init</td>
<td>This creates an application in the current directory</td>
</tr>
<tr>
<td>ember build</td>
<td>This builds the application in the /dist folder</td>
</tr>
<tr>
<td>ember server</td>
<td>This starts a web server</td>
</tr>
<tr>
<td>ember generate &lt;generator-name&gt;</td>
<td>This runs a generator that builds scaffolding for the project</td>
</tr>
<tr>
<td>ember destroy &lt;generator-name&gt;</td>
<td>This uninstalls the module that was created by the generator</td>
</tr>
<tr>
<td>ember test</td>
<td>This runs tests using Testem</td>
</tr>
<tr>
<td>ember install &lt;addon-name&gt;</td>
<td>This installs add-ons</td>
</tr>
</tbody>
</table>
Aliases

Keep in mind that for every command, there is an alias. These aliases make it a little quicker to run commands. Suppose that you wanted to build a new project. Normally, you would type this:

$ ember build

This will work and is fine. It will generate a new project and application structure. You can also use an alias.

$ ember b

Here is a list of some common aliases that you can use. This is optional.

<table>
<thead>
<tr>
<th>Command</th>
<th>Alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>ember build</td>
<td>ember b</td>
</tr>
<tr>
<td>ember generate</td>
<td>ember g</td>
</tr>
<tr>
<td>ember init</td>
<td>ember i</td>
</tr>
<tr>
<td>ember server</td>
<td>ember s</td>
</tr>
<tr>
<td>ember destroy</td>
<td>ember d</td>
</tr>
<tr>
<td>ember test</td>
<td>ember t</td>
</tr>
<tr>
<td>ember version</td>
<td>ember v</td>
</tr>
</tbody>
</table>

Creating your first project

In this recipe we'll create our first project.

How to do it...

We'll begin with the Ember CLI tool to create our first project.

1. Open the command prompt and type the following command:

   $ ember new my-project

   This will create a brand new project called my-project. The project structure will have everything that we need to get started.
2. To display this project, we can simply run the server command:

$ cd my-project
$ ember server

The `ember server` command will start up a web server on port 4200. You can access this port by opening `http://localhost:4200`. You should see the default `Welcome to Ember website`.

It is a good idea to keep the Ember server running while developing your application. Ember CLI uses a tool called LiveReload to refresh the web browser when changes are made. This can be useful to see how new changes are affecting your application. To run LiveReload, simply type `ember server`. This will start the server with LiveReload.

3. The server command defaults to port 4200. You can easily change this using the `--port` argument:

$ ember server --port 1234

This will start the server on port 1234 instead of the default 4200.

4. Another useful option is the `--proxy` argument. This will proxy all **Asynchronous JavaScript and XML (Ajax)** requests to the given address. Let's say that we have a node server running on port 8080. We can run the server as follows:

$ ember server --proxy http://127.0.0.1:8080

For every Ajax request, Ember now will send these requests to the localhost at port 8080.

Keep in mind that as of Ember 2.0, **Internet Explorer (IE)** 8 support has been dropped. All modern web browsers and versions of IE after 8 work fine. If by chance IE 8 support is needed, Ember.js version 1.13 has extended browser support and should work with it.

**How it works...**

The `ember server` command creates a **Node.js Express server**. This server uses LiveReload and refreshes the web page whenever any changes are made. The server command accepts different arguments, including `--proxy` and `--port`. 
There's more...

When running the server, you have access to tests. After you start the server, you'll have access to the QUnit interface. QUnit is a JavaScript unit testing framework. It is used to run your integration and acceptance tests. To access the interface, navigate your browser to http://localhost:4200/tests. This will show all your tests in the project. From here, you can see which tests passed and which failed. We will cover this in the later chapters:

---

MyNewApp Tests

- Hide passed tests
- Check for Globals
- No try-catch
- Hide container

QUnit 1.20.0, Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)

Tests completed in 307 milliseconds.
23 assertions of 23 passed, 0 failed.

1. JSHint - .app.js should pass jshint (1)    ReRun
2. JSHint - components/my-component: components/my-component/component
3. JSHint - hello: hello/controller.js should pass jshint (1)    ReRun
4. JSHint - hello: hello/route.js should pass jshint (1)    ReRun

---

Exploring pods and the folder layout

Ember CLI will create our folder structure for us. Ember.js uses the model-view-controller (MVC) pattern. You'll see in this recipe how the folder structure is laid out and how the model, controller, and view (templates) are separated from each other.

---

Getting ready

Ember CLI relies on ES2015 modules. This means that you can write code today using tomorrow's JavaScript syntax. This is accomplished via the Ember Resolver.

---

ES2015

ECMAScript 6, also known as ES2015, is the upcoming version of the ECMAScript programming language. ES2015 includes several new features, including template strings, destructuring, arrow functions, modules, and class definitions, to name a few. This is all available now within your Ember project.
Pods
An ember pod is a different type of structure that organizes your modules by feature instead of type. As your project grows, you may want to organize your project by feature to help keep things organized. The Ember Resolver will look for a pod structure first before it looks at the traditional structure.

To set up the pod structure automatically, you can edit the .ember-cli file in the root of your project directory and add this line:

```javascript
{
    "usePods": true
}
```

**Downloading the example code**
You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

This will set the default structure to always use pods. When using pods, it is a good idea to set the location where all pods live. To do this, you will need to edit the config/environment.js file:

```javascript
...
var ENV = {
    modulePrefix: 'pod-example',
    podModulePrefix: 'pod-example/pods'

The podModulePrefix property sets the POD path with the following format, {appname}/ {poddir}. In the preceding example, the pod directory is now set to /pods in the app folder. If the location is not set, all new modules will be created in the app/ folder.

**How to do it...**
After a new project is created, a normal folder layout is generated. This layout consists of several different types of modules. Here is a short description of each directory:

<table>
<thead>
<tr>
<th>Directory</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td>app/adapters</td>
<td>Adapters help extend logic to communicate with a backend data store</td>
</tr>
<tr>
<td>app/components</td>
<td>Components are used to help reuse code and must have a dash in their name</td>
</tr>
</tbody>
</table>
### Ember CLI Basics

<table>
<thead>
<tr>
<th>Directory</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td>app/helpers</td>
<td>Helpers are used for HTML reuse</td>
</tr>
<tr>
<td>app/initializers</td>
<td>Initializers are run first and help set up your application</td>
</tr>
<tr>
<td>app/mixins</td>
<td>This is a special type of <em>Ember.Object</em> used with multiple inheritance</td>
</tr>
<tr>
<td>app/routes</td>
<td>Routes help move through different application states</td>
</tr>
<tr>
<td>app/serializers</td>
<td>This serializes your data model</td>
</tr>
<tr>
<td>app/transform</td>
<td>Transform is used to deserialize and serialize model attributes</td>
</tr>
<tr>
<td>app/utils</td>
<td>Utils are small utility classes</td>
</tr>
<tr>
<td>app/models</td>
<td>Models hold the data store</td>
</tr>
<tr>
<td>app/templates</td>
<td>Templates use HTMLBars to display HTML to the user</td>
</tr>
<tr>
<td>app/templates/components</td>
<td>These are templates used in your components</td>
</tr>
</tbody>
</table>

A new project `app` folder with a default layout will look similar to this:

```
./
  ├── app.js
  │    └── components
  │         ├── controllers
  │         │    └── helpers
  │         └── index.html
  │               └── models
  │                   └── routes.js
  │                           └── styles
  │                               └── app.css
  │                                     └── templates
  │                                              └── application.hbs
  │                                                  └── components
```

Each module will have its own directory. For example, the `templates` folder will store all the templates while the `components` controller will store all the components.

Let's say that we added a new post resource using pods. The following command will generate a new post model, route, and template, and it will update the router:

```
$ ember g resource posts
```
Now the filesystem will look like this:

```
|-- app.js
  |-- application
  |    |-- template.hbs
  |-- components
  |    |-- my-component
  |    |    |-- component.js
  |    |    |-- template.hbs
  |-- hello
  |    |-- controller.js
  |    |-- route.js
  |    |-- template.hbs
  |-- helpers
  |-- index.html
  |-- post
  |    |-- model.js
  |-- posts
  |    |-- route.js
  |    |-- template.hbs
  |-- router.js
  |-- styles
  |    |-- app.css
```

Pods sorts directories by features. The post and posts folders are features and the files are named after the function they serve.

**How it works...**

The directory structure in each Ember CLI project is by design. When creating a new project or generating new scaffolding, the CLI will place files in a certain directory with a certain naming structure that the Ember Resolver understands using the ES2015 format.

The Ember Resolver is responsible for the looking up of code in your application and converting the name conventions in the actual class files. With Ember pods, the resolver knows to look there first before the default structure.
Asset compilation

In this recipe we'll take a look at how assets are added to a project.

How to do it...

In your application, at some point, you may want to add assets and minimize or fingerprint your project. This is done in the root folder of your project in the ember-cli-build.js file or in the asset folder.

CSS and assets

All the assets should be placed in the public/assets folder. The assets can be referred throughout the program at assets/images/{image file}. CSS files should be placed in the app/styles folder.

Minifying

By default, CSS and JavaScript files are minified during the production build process. There are ways to turn this functionality on and off. For example, let's say that you want to turn off the minification for both CSS and JavaScript. To do this, we can simply edit the ember-cli-build.js file, and under the // Add options here section, add the minifyCSS and minifyJS section:

```javascript
module.exports = function(defaults) {
  var app = new EmberApp(defaults, {
    // Add options here
    minifyCSS: {
      enabled: false
    },
    minifyJS: {
      enabled: false
    }
  });
};
```

This will tell the compiler not to minify JavaScript and CSS. To build the application in the production mode, simply use the --environment argument:

$ ember build --environment=production
**Fingerprinting**

All files by default will be fingerprinted during the production build process. This will include all js, css, png, jpg, and gif assets. During this process, all these files will have an md5 checksum appended at the end of their filenames. During this process, all HTML and css files will be rewritten to include these new names.

There are several options available when fingerprinting a file. This is all controlled in the ember-cli-build.js file. Let's suppose that you wanted to disable fingerprinting:

```javascript
...
  fingerprint: {
    enabled: false
  }
...
```

Another useful option is to prepend a domain to all static files. This can be done using the prepend option. Once again, this needs to be added to the ember-cli-build.js file in the root of the application folder:

```javascript
...
  fingerprint: {
    prepend: 'http://www.example.com'
  }
...
```

Now, all assets will include the www.example.com domain. For example, a normal JavaScript src file will look like this:

```html
<script src="assets/script.js">
```

This will be transformed into the following:

```html
<script src="http://www.example.com/script-12324adfasdf123234.js">
```

Another useful option is exclude. This accepts an array of strings. Any filename in the exclude array will not be fingerprinted:

```javascript
fingerprint: {
  exclude: ['fonts/12424']
}
```

The ignore option also accepts an array of strings. Any filename that contains any item in the ignore array will not be processed or fingerprinted:

```javascript
fingerprint: {
  ignore: ['fonts/12424']
}
```
The extension option defaults to 'js', 'css', 'png', 'jpg', 'gif', and 'map'. This option can be used to add other file types to get fingerprinted:

```javascript
fingerprint: {
  extension: ['r3', 'html']
}
```

The replaceExtensions option defaults to 'html', 'css', and 'js'. If needed, new file types can be added to replace source code with new checksum file names:

```javascript
fingerprint: {
  replaceExtensions: ['html', 'htm']
}
```

**How it works...**

The import process is done via the Broccoli asset pipeline library. This build tool performs all the fingerprinting, minifying, and importing of the assets. In addition, Broccoli handles all the preprocessors if the appropriate plugins are installed.

The asset manifest is located in the `ember-cli-build.js` file in the root of the project folder. You can only import assets that are in the `bower_components` or `vendor` directories.

**Dependency management**

Let's look at dependency management and how we can use it in our Ember projects.

**How to do it...**

Bower is used for dependency management for Ember CLI. Bower is a frontend tool that is used to help fetch and install packages that you might need.

1. The `bower.json` file is located in the root folder of your project. It contains all the dependencies. Let's say that we want to install the **Bootstrap library**:

   ```bash
   $ bower install bootstrap --save
   ```

   This command will install `bootstrap` in the `bower_components` folder and save the package information in the `bower.json` file.
**Ember add-ons**

Another popular way of adding third-party libraries to Ember is using add-ons or addons as you sometimes see it. An add-on is Ember's way of sharing libraries between applications. There are well over a thousand of them available.

You can install add-ons using Ember CLI. For example, to install Bootstrap, you'd type this on the command line in the project directory:

```
$ ember install ember-bootstrap
```

You can easily find a list of add-ons at these websites:

- http://www.emberobserver.com
- http://www.emberaddons.com

This will be discussed in more detail in the *Working and creating add-ons* recipe in Chapter 11, *Real-Time Web Applications*.

2. If, for some reason, you need to reinstall your dependencies, you can run the `install` command by itself:

```
$ bower install
```

This will install all dependencies that are listed in the `bower.json` file.

**The app.import code**

Ember CLI allows you to load *Asynchronous Module Definition (AMD)* and non-AMD assets. It's a way of defining code modules and their dependencies.

1. To load a non-AMD asset, you'll need to import it using the `ember-cli-build.js` file:

   ```javascript
   ...
   app.import('bower_components/moment/moment.js');
   ```

2. This is useful as you can use Bower to install components, and then use the `app.import AMD` so that it's available in the program. You'll need to consult the package specification to see how to use it.

**Tip on JSHint**

JSHint is a community-driven tool that detects errors and potential problems with JavaScript code. It's built-in in Ember CLI. When using non-AMD assets, you may get errors with JSHint if you have global variables. To fix this, add `/* global MY_GLOBAL */` at the top of your module page. In the moment example, it would look like `/* global moment */`. 
3. AMD assets are imported in a similar way. You add the path in the first argument and a list of exports and modules in the second:

```javascript
app.import('bower_components/ic-ajax/dist/named-amd/main.js', {
  exports: {
    'ic-ajax': [
      'default',
      'defineFixture',
      'lookupFixture',
      'raw',
      'request',
    ]
  }
});
```

4. To use this asset in your application, you can import it as follows:

```javascript
import { raw as icAjaxRaw } from 'ic-ajax';
```

**How it works...**

Dependency management is done via Bower. After the dependency is installed, the Broccoli library is called on to add the assets to the pipeline. Both these tools are written in node and are built-in in Ember CLI.

**Upgrading your project**

Ember CLI is constantly being upgraded and every six weeks Ember.js has another release. It is important to keep your build tools and versions up to date. In this recipe we'll look at the best way of doing this.

**How to do it...**

To upgrade your Ember CLI version, you must perform the following steps:

1. Begin by uninstalling the old ember-cli:
   ```bash
   $ npm uninstall -g ember-cli
   ```

2. Clear the npm cache:
   ```bash
   $ npm cache clean
   ```

3. Clear the Bower cache:
   ```bash
   $ bower cache clean
   ```
4. Install the latest version of ember-cli:
   $ npm install -g ember-cli

5. If you need, you can specify the version to use represented by X.X.X:
   $ npm install -g ember-cli@X.X.X

Updating an existing project

In some situations, you might want to take an existing project and update it to the latest version of Ember CLI. In this case, you'll need to perform the following steps:

1. Begin by changing directories that you want to upgrade in the root of the project folder. Delete these temporary development directories:
   $ rm -rf node_modules bower_components dist tmp

2. Update the package.json file with the version of Ember that we're upgrading to using this command:
   $ npm install ember-cli@X.X.X --save-dev

   X.X.X represents the version of ember-cli. The --save-dev argument will save the information in the package.json file.

3. Install all the npm and Bower packages again:
   $ npm install
   $ bower install

4. The last step is to run the init command:
   $ ember init

   The init command will add the default project blueprint to your directory.

   ```
   init
   
   The init command will create a new application blueprint in your project directory. Follow the prompts and review all the changes. You may be asked to replace existing files. Press d to do a diff of the files and review the changes made. Create a backup of your project before you begin the upgrade process.
   ```

Keep in mind that after upgrading your project, you might have many new deprecation warnings to deal with. You will see these warnings when you run ember server. Each one will need to be addressed.
To address these deprecations, view the warnings that the applications provide. For instance, you may get a warning about Ember.View. The warning will describe that Ember.Component should be used instead. You'll then need to swap out the effected code with Ember components instead of Ember views.

**How it works...**

When you upgrade the tool, you are simply uninstalling the node package and reinstalling the latest one. It's a good idea to clear the Bower and Node cache as well so that Node and Bower won't have any conflicting packages.

When we update an existing project we first have to make sure that all the existing modules and packages are deleted. This is important because when we install the latest version of Ember CLI, some packages might change. After ember-cli is saved back in the package file, then you can install npm and Bower again.

Running ember init generates the application structure in the directory that you're in. This is important because some files may have changed since the last upgrade. You can always press d to diff the changes.

**Deployment**

After creating your application you'll need to be able to deploy it. Let's take a look at a few ways to deploy your application.

**How to do it...**

1. The first fundamental step before deployment is building your project. To build your application, run the build command:
   
   $ ember build

2. This command builds the contents of the project in the /dist folder. To build your project for production, you'll need to use the -prod argument:
   
   $ ember build -prod

3. If you need, you can designate the default output folder:
   
   $ ember build -prod -o<directory>

   Building your application for production will automatically minify your files as well as fingerprint them. This does not occur when your environment is set up for development, which is set by default.

   The /dist folder has everything that your web server needs. At this point, to deploy your application, all you need to do is copy the contents of the /dist folder to your web server.
Ember CLI Deploy

Another great way to deploy your Ember application is to use the Ember add-on called Ember CLI Deploy. This add-on helps you deploy your Ember application to a number of different services. It has a very active community behind it so you can expect frequent updates. As your Ember applications grows you may want to take a look at this. You can find more information about Ember CLI deploy here: https://github.com/ember-cli/ember-cli-deploy.

Deploying to Firebase

Firebase is a backend service that can handle data storage, user authentication, static hosting, and more. For this example we'll be using Firebase as a way to host our Ember application.

1. Sign up for an account with Firebase. This can be done at http://www.firebase.com.
2. Next, install the firebase-tools:
   
   $ npm install -g firebase-tools
3. After you have a project created and you're ready to deploy, run the `firebase init` command in the root of the folder:
   
   $ firebase init

   After running this command, you'll be asked a few questions. It will first ask you to sign in to your Firebase account. Enter your credentials to continue. Then, it will ask for the name of your Firebase application. Lastly, it will ask for the name of your app's public directory. In most cases, this should be /dist.
4. Edit the `firebase.json` file and add some rewrite rules:

   ```json
   {
   "firebase": "my-new-app",
   "public": "dist",
   "rewrites": [
   {"source": "**",
   "destination": "index.html"
   }],
   }
   ```

   This is needed to help with navigation in the application. Change `my-new-app` to the name of your app as well.
5. All that's left is to deploy to Firebase:

   $ firebase deploy
**How it works...**

The Ember CLI build process is compiled using the Broccoli asset pipeline and the build tool itself. It takes all the files and minifies, fingerprints, and organizes them in the `/dist` folder so that it is ready to be deployed.

Many services exist that can host static files. You can use Firebase or just host it in your own Nginx or Apache server after building it for production.
Where to buy this book

You can buy Ember.js cookbook from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.