 (https://github.com/ember-cli/ember-cli.github.io/blob/master/_posts/2014-04-03-getting-started.md)
Ember CLI (/)

Getting Started

Prerequisites

Node

First, install the latest version of Node.

Node is available for a variety of platforms at nodejs.org (<http://nodejs.org/>). It is important that you *not* install Node with `sudo` in order to avoid permission problems with some ember-cli commands. On Unix, `nvm` (<https://github.com/creationix/nvm>) provides a convenient way to do this. On OS X, you can also use Homebrew (<https://changelog.com/posts/install-node-js-with-homebrew-on-os-x>). On Windows, chocolatey (<https://chocolatey.org/packages/nodejs>) is an option.

After the installation is complete, verify that Node is set up correctly by typing the below commands on the command line. Both should output a version number:

```
node -v
npm -v
```

Ember CLI

Once you've installed Node, you'll need to globally install Ember CLI:

```
npm install -g ember-cli
```

This will give you access to the `ember` command-line runner.

Bower

You'll need to globally install Bower (<http://bower.io>), a package manager that keeps your front-end dependencies (including jQuery, Ember, and QUnit) up-to-date:

```
npm install -g bower
```

This will give you access to the `bower` command-line runner.

Watchman

On OSX and UNIX-like operating systems, we recommend installing Watchman (<https://facebook.github.io/watchman/>) version 3.x, which provides Ember CLI a more effective way for watching project changes.

File-watching on OSX is error-prone and Node's built-in `NodeWatcher` has trouble observing large trees. Watchman (<https://facebook.github.io/watchman/>) on the other hand, solves these problems and performs well on extremely massive file trees. You can read more about Facebook's motivations here (<https://www.facebook.com/notes/facebook-engineering/watchman-faster-builds-with-large-source-trees/10151457195103920>).

On OSX, you can install Watchman using Homebrew (<http://brew.sh/>):

```
brew install watchman
```

For complete installation instructions, refer to the docs on the Watchman website (<https://facebook.github.io/watchman/>). Note, there exists a similarly named `npm` package (`watchman`) which is **not** the intended installation. If you have this package installed you may see the following warning:

```
invalid watchman found, version: [2.9.8] did not satisfy [^3.0.0], falling back to NodeWatcher
```

If you intend on using the `npm` version for another purpose, make sure it's not on your `PATH` otherwise, remove it using:

```
npm uninstall -g watchman
```

When in doubt, use the following command to inspect which Watchman(s) you have:

```
which -a watchman
```

Lastly, when Watchman is not installed, a notice is displayed when invoking various commands. You can safely ignore this message:

```
Could not find watchman, falling back to NodeWatcher for file system events
```

PhantomJS

With Ember CLI, you can use the automated test runner of your choice, however most testing services will recommend or require PhantomJS (<http://phantomjs.org/>), which you can install via `npm` (<https://www.npmjs.com/package/phantomjs>) or the PhantomJS website (<http://phantomjs.org>). Note, PhantomJS is the default test runner for Testem (<https://github.com/airportyh/testem>) and Karma (<http://karma-runner.github.io/0.12/index.html>).

If you want to use PhantomJS to run your integration tests, it must be globally installed:

```
npm install -g phantomjs-prebuilt
```

Create a new project

Run the `new` command along with the desired app name to create a new project:

```
ember new my-new-app
```

Ember CLI will create a new `my-new-app` directory and in it, generate the application structure.

Once the generation process is complete, launch the app:

```
cd my-new-app
ember server
```

Navigate to `http://localhost:4200` to see your new app in action.

Navigate to `http://localhost:4200/tests` to see your test results in action.

Migrating an existing Ember project that doesn't use Ember CLI

If your app uses the deprecated Ember App Kit, there is a migration guide (<https://github.com/stefanpenner/ember-app-kit#migrating-to-ember-cli>) located on the README.

If your app uses globals (e.g. `App.Post`) from a different build pipeline such as Grunt, Ember-Rails, or Gulp, you can try using the Ember CLI migrator (<https://github.com/fivetanley/ember-cli-migrator>). The Ember CLI migrator is a command line tool that looks at your JavaScript code using a JavaScript parser and rewrites it to ES6 following Ember CLI's conventions. The migrator keeps your code style and keeps git history available via `git log --follow`.

Cloning an existing project

If you are checking out an existing Ember CLI-based project, you will need to install `npm` and `bower` dependencies before running the server:

```
git clone git@github.com:me/my-app.git
cd my-app
npm install
bower install
ember server
```

Using Ember CLI

Command	Purpose
<code>ember</code>	Prints out a list of available commands.
<code>ember new <app-name></code>	Creates a directory called <code><app-name></code> and in it, generates an application structure. If git is available the directory will be initialized as a git repository and an initial commit will be created. Use <code>--skip-git</code> flag to disable this feature.
<code>ember init</code>	Generates an application structure in the current directory.
<code>ember build</code>	Builds the application into the <code>dist/</code> directory (customize via the <code>--output-path</code> flag). Use the <code>--environment</code> flag to specify the build environment (defaults to <code>development</code>). Use the <code>--watch</code> flag to keep the process running and rebuilding when changes occur.
<code>ember server</code>	Starts the server. The default port is <code>4200</code> . Use the <code>--proxy</code> flag to proxy all ajax requests to the given address. For example, <code>ember server --proxy http://127.0.0.1:8080</code> will proxy all ajax requests to the server running at <code>http://127.0.0.1:8080</code> . Aliases: <code>ember s</code> , <code>ember serve</code>
<code>ember generate <generator-name> <options></code>	Runs a specific generator. To see available generators, run <code>ember help generate</code> . Alias: <code>ember g</code>
<code>ember destroy <generator-name> <options></code>	Removes code created by the <code>generate</code> command. If the code was generated with the <code>--pod</code> flag, you must use the same flag when running the <code>destroy</code> command. Alias: <code>ember d</code>
<code>ember test</code>	Run tests with Testem in CI mode. You can pass any options to Testem through a <code>testem.json</code> file. By default, Ember CLI will search for it under your project's root. Alternatively, you can specify a <code>config-file</code> . Alias: <code>ember t</code>
<code>ember install <addon-name></code>	Installs the given addon into your project and saves it to the <code>package.json</code> file. If provided, the command will run the addon's default blueprint (http://www.ember-cli.com/extending/#generators-and-blueprints).

Folder Layout

File/directory	Purpose
<code>app/</code>	Contains your Ember application's code. Javascript files in this directory are <i>compiled</i> through the ES6 module transpiler and concatenated into a file called <code><app-name>.js</code> . See the table below for more details.
<code>dist/</code>	Contains the <i>distributable</i> (optimized and self-contained) output of your application. Deploy this to your server!
<code>public/</code>	This directory will be copied verbatim into the root of your built application. Use this for assets that don't have a build step, such as images or fonts.
<code>tests/</code>	Includes your app's unit and integration tests, as well as various helpers to load and run the tests.
<code>tmp/</code>	Temporary application build-step and debug output.

File/directory	Purpose
bower_components/	Bower dependencies (both default and user-installed).
node_modules/	npm dependencies (both default and user-installed).
vendor/	Your external dependencies not installed with Bower or npm.
.jshintrc	JSHint (http://jshint.com/) configuration.
.gitignore	Git configuration for ignored files.
ember-cli-build.js	Contains the build specification for Broccoli (https://github.com/joliss/broccoli).
bower.json	Bower configuration and dependency list. See Managing Dependencies.
package.json	npm configuration and dependency list. Mainly used to list the dependencies needed for asset compilation.

Layout within app directory

File/directory	Purpose
app/app.js	Your application's entry point. This is the first executed module.
app/index.html	The only page of your single-page app! Includes dependencies, and kickstarts your Ember application. See app/index.html.
app/router.js	Your route configuration. The routes defined here correspond to routes in app/routes/.
app/styles/	Contains your stylesheets, whether SASS, LESS, Stylus, Compass, or plain CSS (though only one type is allowed, see Asset Compilation). These are all compiled into <app-name>.css.
app/templates/	Your HTMLBars templates. These are compiled to /dist/assets/<app-name>.js. The templates are named the same as their filename, minus the extension (i.e. templates/foo/bar.hbs -> foo/bar).
app/controllers/ , app/models/ , etc.	Modules resolved by the Ember CLI resolver. See Using Modules & the Resolver.

app/index.html

The app/index.html file lays the foundation for your application. This is where the basic DOM structure is laid out, the title attribute is set, and stylesheet/javascript includes are done. In addition to this, app/index.html includes multiple hooks - {{content-for 'head'}} and {{content-for 'body'}} - that can be used by addons to inject content into your application's head or body. These hooks need to be left in place for your application to function properly however, they can be safely ignored unless you are directly working with a particular addon.

Addons

Addons are registered in npm with a keyword of ember-addon. See a full list of existing addons registered in NPM here (<https://www.npmjs.org/browse/keyword/ember-addon>).

Developing on a subpath

If your app isn't running on the root URL (/), but on a subpath (like /my-app/), your app will only be accessible on /my-app/ and not on /my-app , too. Sometimes, this can be a bit annoying. Therefore you should take the following snippet as an example for a simple route which takes you to the right URL if you've entered the wrong one:

```
// index.js
app.get('/my-app', function(req, res, next) {

  if (req.path !== '/my-app/') {
    res.redirect('/my-app/');
  } else {
    next();
  }

});
```

Just place it within the index.js file of your app's /server directory (so that it gets applied when the ember-cli development server is being started). The snippet simply tests if the URL that is being accessed begins with /my-app/. And if it doesn't, you'll get redirected. Otherwise, the redirection will be skipped.

 (https://github.com/ember-cli/ember-cli/edit/gh-pages/_posts/2014-04-02-using-modules-and-the-resolver.md)

Using Modules & the Resolver

The Ember Resolver is the mechanism responsible for looking up code in your application and converting its naming conventions into the actual classes, functions, and templates that Ember needs to resolve its dependencies, for example, what template to render for a given route. For an introduction to the Ember Resolver, and a basic example of how it actually works, see this video (<https://www.youtube.com/watch?v=OY0PzrltMYc#t=51>) by Robert Jackson (<https://www.twitter.com/@rwjblue>).

In the past, Ember's Default Resolver worked by putting everything into a global namespace, so you will come across the following pattern:
Ember CLI (/)

```
App.IndexRoute = Ember.Route.extend({
  model: function() {
    return ['red', 'yellow', 'blue'];
  }
});
```

Today, Ember CLI uses a newer version of the Resolver (<https://github.com/stefanpenner/ember-resolver>) based on ES6 semantics. This means that you can build your apps using syntax from future JavaScript versions, but output AMD modules that can be used by existing JavaScript libraries today.

For example, this route definition in `app/routes/index.js` would result in a module called `your-app/routes/index`. Using the resolver, when Ember looks up the index route, it will find this module and use the object that it exports.

```
// app/routes/index.js
import Ember from "ember";

export default Ember.Route.extend({
  model: function() {
    return ['red', 'yellow', 'blue'];
  }
});
```

You can also require modules directly with the following syntax:

```
import FooMixin from "../mixins/foo";
```

You can reference a module by using either a relative or absolute path. If you would like to reference a module using absolute begin the path with the app name defined in `package.json`:

```
import FooMixin from "appname/mixins/foo";
```

Similarly, you can give any name to the variable into which you import a module when doing so manually; see how the module `mixins/foo` is assigned to variable `FooMixin` in the example above.

Using Ember or Ember Data

To use `Ember` or `DS` (for Ember Data) in your modules you must import them:

```
import Ember from "ember";
import DS from "ember-data";
```

Using Pods

One of the enhancements that the new Resolver brings is that it will first look for Pods before the traditional project structure.

Module Directory Naming Structure

Directory	Purpose
app/adapters/	Adapters with the convention <code>adapter-name.js</code> .
app/components/	Components with the convention <code>component-name.js</code> . Components must have a dash in their name. So <code>blog-post</code> is an acceptable name, but <code>post</code> is not.
app/helpers/	Helpers with the convention <code>helper-name.js</code> . Helpers must have a dash in their name. Remember that you must register your helpers by exporting <code>makeBoundHelper</code> or calling <code>registerBoundHelper</code> explicitly.
app/initializers/	Initializers with the convention <code>initializer-name.js</code> . Initializers are loaded automatically.
app/mixins/	Mixins with the convention <code>mixin-name.js</code> .
app/models/	Models with the convention <code>model-name.js</code> .
app/routes/	Routes with the convention <code>route-name.js</code> . Child routes are defined in sub-directories, <code>parent/child.js</code> . To provide a custom implementation for generated routes (equivalent to <code>App.Route</code> when using globals), use <code>app/routes/basic.js</code> .
app/serializers/	Serializers for your models or adapter, where <code>model-name.js</code> or <code>adapter-name.js</code> .
app/transforms/	Transforms for custom Ember Data attributes, where <code>attribute-name.js</code> is the new attribute.
app/utils/	Utility modules with the convention <code>utility-name.js</code> .

All modules in the `app` directory can be loaded by the resolver but typically classes such as `mixins` and `utils` should be loaded manually with an import statement. For more information, see [Naming Conventions](#).

Resolving Handlebars helpers

Ember CLI (7)

Custom Handlebars helpers are one of the ways that you can use the same HTML multiple times in your web application. Registering your custom helper allows it to be invoked from any of your Handlebars templates. Custom helpers are located under `app/helpers`.

```
// app/helpers/upper-case.js
import Ember from "ember";

export default Ember.Helper.helper(function([value]) {
  return value.toUpperCase();
});
```

In `some-template.hbs`:

```
{{upper-case "foo"}}
```

While previous versions of ember-cli required auto-resolved helpers only if they contain a dash, now all helpers are auto-resolved, regardless of whether they contain a dash or not.

A common pattern with helpers is to define a helper to use your views (e.g. for a custom text field view, `MyTextField` a helper `my-text-field` to use it). It is advised to leverage Components instead.

Do this:

```
// Given... app/components/my-text-field.js
import Ember from "ember";

export default Ember.TextField.extend({
  // some custom behaviour...
});
```

Using global variables or external scripts

If you want to use external libraries that write to a global namespace (e.g. `moment.js` (<http://momentjs.com/>)), you need to add those to the `predef` section of your project's `.jshintrc` file and set its value to `true`. If you use the lib in tests, you need to add it to your `tests/.jshintrc` file, too.

 (https://github.com/ember-cli/ember-cli.github.io/blob/master/_posts/2014-04-01-naming-conventions.md)

Naming Conventions

Overview

- `kebab-case`
 - file names
 - directory names
 - html tags/ember components
 - CSS classes
 - URLs
- `camelCase`
 - JavaScript
 - JSON

When using Ember CLI it's important to keep in mind that the Resolver changes some of the naming conventions you would typically use out of the box with Ember, Ember Data and Handlebars. In this section we review some of these naming conventions.

Module Examples

Adapters

```
// app/adapters/application.js
import Ember from "ember";
import DS from "ember-data";

export default DS.RESTAdapter.extend({});
```

Components

```
// app/components/time-input.js
import Ember from "ember";

export default Ember.TextField.extend({});
```

Controllers

Ember CLI (7)

```
// app/controllers/stop-watch.js
import Ember from "ember";

export default Ember.Controller.extend({});
```

And if it's a nested controller, we can declare nested/child controllers like such: `app/controllers/posts/index.js` .

Helpers

```
// app/helpers/format-time.js
import Ember from "ember";

export default Ember.Helper.helper(function(){});
```

Initializers

```
// app/initializers/observation.js
export default {
  name: 'observation',
  initialize: function() {
    // code
  }
};
```

Note: initializers are loaded automatically.

Mixins

```
// app/mixins/evented.js
import Ember from "ember";

export default Ember.Mixin.create({});
```

Models

```
// app/models/observation.js
import DS from "ember-data";

export default DS.Model.extend({});
```

Routes

```
// app/routes/timer.js
import Ember from "ember";

export default Ember.Route.extend({});
```

Nested routes as such: `app/routes/timer/index.js` OR `app/routes/timer/record.js` .

Serializers

```
// app/serializers/observation.js
import DS from "ember-data";

export default DS.RESTSerializer.extend({});
```

Transforms

```
// app/transforms/time.js
import DS from "ember-data";

export default DS.Transform.extend({});
```

Utilities

```
// app/utls/my-ajax.js
export default function myAjax() {};
```

Views

```
<!-- app/index.hbs -->
{{view 'stop-watch'}}
```

```
// app/views/stop-watch.js
import Ember from "ember";

export default Ember.View.extend({});
```

And views, which can be referenced in sub-directories, but have no inheritance.

```
app/templates/time-input.hbs -->
{{view 'inputs/time-input'}}
```

```
// app/views/inputs/time-input.js
import Ember from "ember";

export default Ember.TextField.extend({});
```

Filenames

It is important to keep in mind that the Resolver uses filenames to create the associations correctly. This helps you by not having to namespace everything yourself. But there are a couple of things you should know.

All filenames should be lowercased

```
// models/user.js
import Ember from "ember";
export default Ember.Model.extend();
```

Dasherized file and directory names are required

```
// controllers/sign-up.js
import Ember from "ember";
export default Ember.Controller.extend();
```

Nested directories

If you prefer to nest your files to better manage your application, you can easily do so.

```
// controllers/posts/new.js results in a controller named "controllers.posts/new"
import Ember from "ember";
export default Ember.Controller.extend();
```

You cannot use paths containing slashes in your templates because Handlebars will translate them back to dots. Simply create an alias like this:

```
// controllers/posts.js
import Ember from "ember";
export default Ember.Controller.extend({
  needs: ['posts/details'],
  postsDetails: Ember.computed.alias('controllers.posts/details')
});
```

```
<!-- templates/posts.hbs -->
<!-- because {{controllers.posts.details.count}} does not work -->
{{postsDetails.count}}
```

Tests

Test filenames should be suffixed with `-test.js` in order to run.

Pod structure

As your app gets bigger, a feature-driven structure may be better. Splitting your application by functionality/resource would give you more power and control to scale and maintain it. As a default, if the file is not found on the POD structure, the Resolver will look it up within the normal structure.

In this case, you should name the file as its functionality. Given a resource `users`, the directory structure would be:

- `app/users/controller.js`
- `app/users/route.js`
- `app/users/template.hbs`

Rather than hold your resource directories on the root of your app you can define a POD path using the attribute `podModulePrefix` within your environment configs. The POD path should use the following format: `{appname}/{poddir}`.

```
// config/environment.js
module.exports = function(environment) {
  var ENV = {
    modulePrefix: 'my-new-app',
    // namespaced directory where resolver will look for your resource files
    podModulePrefix: 'my-new-app/pods',
    environment: environment,
    baseURL: '/',
    locationType: 'auto'
    //...
  };

  return ENV;
};
```

Then your directory structure would be:

Ember CLI (/)

- `app/pods/users/controller.js`
- `app/pods/users/route.js`
- `app/pods/users/template.hbs`

✍️ (https://github.com/ember-cli/ember-cli.github.io/blob/master/_posts/2013-04-12-ember-data.md)

Using With Ember Data

The current version of Ember Data is included with Ember CLI.

Ember Data has undergone some major reboots, drastically simplifying it and making it easier to use with the Ember resolver. Here's some tips for using it within Ember CLI.

To use ember-cli without Ember Data remove the dependency from `package.json` (the same applies for `ic-ajax` (<https://github.com/rwjblue/ember-cli-ic-ajax>))

```
npm rm ember-data --save-dev
```

Models

Models are critical in any dynamic web application. Ember Data makes making models extremely easy.

For example, we can create a `todo` model like so:

```
// models/todo.js
import DS from "ember-data";

export default DS.Model.extend({
  title: DS.attr('string'),
  isCompleted: DS.attr('boolean'),
  quickNotes: DS.hasMany('quick-note')
});

// models/quick-note.js
import DS from "ember-data";

export default DS.Model.extend({
  name: DS.attr('string'),
  todo: DS.belongsTo('todo')
});
```

Note, that filenames should be all lowercase and dasherized - this is used by the *Resolver* automatically.

Adapters & Serializers

Ember Data makes heavy use of *per-type* adapters and serializers. These objects can be resolved like any other.

Adapters can be placed at `/app/adapters/type.js` :

```
// adapters/post.js
import DS from "ember-data";

export default DS.RESTAdapter.extend({});
```

And it's serializer can be placed in `/app/serializers/type.js` :

```
// serializers/post.js
import DS from "ember-data";

export default DS.RESTSerializer.extend({});
```

Application-level (default) adapters and serializers should be named `adapters/application.js` and `serializers/application.js` , respectively.

Mocks and fixtures

If you're used to using fixtures to get test data into your app during development, you won't be able to create fixture data like you're used to doing (i.e. as specified in the guides (<http://emberjs.com/guides/models/the-fixture-adapter/>)). This is because the models in your Ember CLI app (like all other objects) aren't attached to the global namespace.

Ember CLI comes with an **http-mock** generator which is preferred to fixtures for development and testing. Mocks have several advantages over fixtures, a primary one being that they interact with your application's adapters. Since you'll eventually be hooking your app up to a live API, it's wise to be testing your adapters from the onset.

To create a mock for a `posts` API endpoint, use

```
ember g http-mock posts
```

A basic ExpressJS (<http://expressjs.com/>) server will be scaffolded for your endpoint under `/your-app/server/mocks/posts.js` . Once you add the appropriate JSON response, you're ready to go. The next time you run `ember server` , your new mock server will be listening for any API requests from your Ember app.

Note: Mocks are just for development. The entire `/server` directory will be ignored during `ember build` and `ember test`.

If you decide to use fixtures instead of mocks, you'll need to use `reopenClass` within your model class definitions. First, create a fixture adapter, either for a single model or your entire application:

```
// adapters/application.js
import DS from "ember-data";

export default DS.FixtureAdapter.extend({});
```

Then add fixture data to your model class:


```
// models/author.js
import DS from "ember-data";

var Author = DS.Model.extend({
  firstName: DS.attr('string'),
  lastName: DS.attr('string')
});

Author.reopenClass({
  FIXTURES: [
    {id: 1, firstName: 'Bugs', lastName: 'Bunny'},
    {id: 2, firstName: 'Wile E.', lastName: 'Coyote'}
  ]
});

export default Author;
```

Your Ember app's API requests will now use your fixture data.

 (https://github.com/ember-cli/ember-cli.github.io/blob/master/_posts/2013-04-11-testing.md)

Testing

Running existing tests

Running your tests is as easy as one of these commands:

```
ember test           # will run your test-suite in your current shell once
ember test --server  # will run your tests on every file-change
ember t -s -m 'Unit | Utility | name' # will run only the unit test module for a utility by `name`, on every change
ember t -f 'match a phrase in test description(s)' # will run only the tests with a description that matches a phrase
```

Alternatively you can run the tests in your regular browser using the QUnit interface. Run `ember server` and navigate to `http://localhost:4200/tests`. When the app runs in `/tests` it runs in the development environment, not the test environment.

Writing a test

- `ember-testing`
- `helpers`
- `unit/acceptance`

The default tests in Ember CLI use the QUnit (<http://qunitjs.com/>) library, though Mocha (<http://mochajs.org/>) / Chai (<http://chaijs.com/>) testing is possible using `ember-cli-mocha` (<https://github.com/switchfly/ember-cli-mocha>). The included tests demonstrate how to write both unit tests and acceptance/integration tests using the new `ember-testing` package (<http://ianpetzer.wordpress.com/2013/06/14/getting-started-with-integration-testing-ember-js-using-ember-testing-and-qunit-rails/>).

Test filenames should be suffixed with `-test.js` in order to run.

If you have manually set the `locationType` in your `environment.js` to `hash` or `none` you need to update your `tests/index.html` to have absolute paths (`/assets/vendor.css` and `/testem.js` vs the default relative paths).

CI Mode with Testem

`ember test` will run your tests with `Testem` on CI mode. You can pass any option to `Testem` using a configuration file.

If you are capturing output from the `Testem` xunit reporter, use `ember test --silent` to silence unwanted output such as the ember-cli version. If you want to capture output to a file you can use `report_file: "path/to/file.xml"` in your `testem.json` config file.

By default, your integration tests will run on PhantomJS (<http://phantomjs.org/>). You can install via `npm` (<https://www.npmjs.org/>):

```
npm install -g phantomjs-prebuilt
```

We plan to make your test runner pluggable, so you can use your favorite runner.

Using `ember-qunit` for integration tests

All Ember Apps come with built-in ember test helpers (https://guides.emberjs.com/v2.0.0/testing/acceptance/#toc_test-helpers), which are useful for writing integration tests. In order to use them, you will need to import `tests/helpers/start-app.js`, which injects the required helpers.

Be sure to use the `module` function to invoke `beforeEach` and `afterEach`.

```
import Ember from "ember";
import { module, test } from 'qunit';
import startApp from '../helpers/start-app';
var App;

module('An Integration test', {
  beforeEach: function() {
    App = startApp();
  },
  afterEach: function() {
    Ember.run(App, App.destroy);
  }
});

test("Page contents", function(assert) {
  assert.expect(2);
  visit('/foos').then(function() {
    assert.equal(find('.foos-list').length, 1, "Page contains list of models");
    assert.equal(find('.foos-list .foo-item').length, 5, "List contains expected number of models");
  });
});
```

Using ember-qunit for unit tests

An Ember CLI-generated project comes pre-loaded with ember-qunit (<https://github.com/rpflorence/ember-qunit>) which includes several helpers (http://emberjs.com/guides/testing/unit-test-helpers/#toc_unit-testing-helpers) to make your unit-testing life easier, i.e.:

- `moduleFor`
- `moduleForModel`
- `moduleForComponent`

`moduleFor(fullName, description, callbacks, delegate)`

The generic resolver that will load what you specify. The usage closely follows QUnit's own `module` function. Its use can be seen within the supplied `index-test.js` (<https://github.com/stefanpenner/ember-app-kit/blob/master/tests/unit/routes/index-test.js>):

```
// tests/unit/routes/index-test.js
import { test, moduleFor } from 'ember-qunit';

moduleFor('route:index', "Unit - IndexRoute", {
  // only necessary if you want to load other items into the runtime
  // needs: ['controller:index']
  beforeEach: function () {},
  afterEach: function () {}
});

test("it exists", function(assert){
  assert.ok(this.subject());
});
```

`fullname`

The resolver friendly name of the object you are testing.

`description`

The description that will group all subsequent tests under. Defaults to the `fullname`.

`callbacks`

You are able to supply custom `beforeEach`, `afterEach`, & subject functionality by passing them into the `callbacks` parameter. If other objects should be loaded into Ember.js, specify the objects through the `needs` property.

`delegate`

To manually modify the container & the testing context, supply a function as the `delegate` matching this signature `delegate(container, testing_context)`.

`this.subject()` calls the factory for the object specified by the `fullname` and will return an instance of the object.

`moduleForModel(name, description, callbacks)`

Extends the generic `moduleFor` with custom loading for testing models:

Ember CLI

```
// tests/unit/models/post-test.js
import DS from 'ember-data';
import Ember from 'ember';
import { test, moduleForModel } from 'ember-qunit';

moduleForModel('post', 'Post Model', {
  needs: ['model:comment']
});

test('Post is a valid ember-data Model', function (assert) {
  var store = this.store();
  var post = this.subject({title: 'A title for a post', user: 'bob'});
  assert.ok(post);
  assert.ok(post instanceof DS.Model);

  // set a relationship
  Ember.run(function() {
    post.set('comment', store.createRecord('comment', {}))
  });

  assert.ok(post.get('comment'));
  assert.ok(post.get('comment') instanceof DS.Model);
});
```

name

The name of the model you are testing. It is necessary to only supply the name, not the resolver path to the object(`model:post => post`).

description

The description that will group all subsequent tests under. Defaults to the `name` .

callbacks

You are able to supply custom `beforeEach`, `afterEach`, & `subject` functionality by passing them into the `callbacks` parameter. If other objects should be loaded into Ember.js, specify the objects through the `needs` property.

Note: If the model you are testing has relationships to any other model, those must be specified through the `needs` property.

`this.store()` retrieves the `DS.Store` .

`this.subject()` calls the factory for the `DS.Model` specified by the `fullname` and will return an instance of the object.

`moduleForComponent(name, description, callbacks)`

Extends the generic `moduleFor` with custom loading for testing components:

```
// tests/integration/components/pretty-color-test.js
import Ember from "ember";
import { test, moduleForComponent } from 'ember-qunit';

moduleForComponent('pretty-color');

test('changing colors', function(assert){
  var component = this.subject();

  Ember.run(function(){
    component.set('name','red');
  });

  // first call to $() renders the component.
  assert.equal(this.$().attr('style'), 'color: red;');

  Ember.run(function(){
    component.set('name', 'green');
  });

  assert.equal(this.$().attr('style'), 'color: green;');
});
```

name

The name of the component you are testing. It is necessary to only supply the name, not the resolver path to the object(`component:pretty-color => pretty-color`).

description

The description that will group all subsequent tests under. Defaults to the `name` .

callbacks

You are able to supply custom `beforeEach`, `afterEach`, & `subject` functionality by passing them into the `callbacks` parameter. If other objects should be loaded into Ember.js, specify the objects through the `needs` property.

`this.subject()` calls the factory for the `Ember.Component` specified by the `fullname` and will return an instance of the object.

The first call `this.$()` will render out the component. So if you want to test styling, you must access the component via `jQuery`.

Writing your own test helpers

Ember testing provides that ability to register your own test helpers. In order to use these with ember-cli they must be registered before `startApp` is defined.

Depending on your approach, you may want to define one helper per file or a group of helpers in one file.

Single helper per file

```
// helpers/routes-to.js
import Ember from "ember";

export default Ember.Test.registerAsyncHelper('routesTo', function (app, assert, url, route_name) {
  visit(url);
  andThen(function () {
    assert.equal(currentRouteName(), route_name, 'Expected ' + route_name + ', got: ' + currentRouteName());
  });
});
```

This can then be used in `start-app.js`, like this

```
// helpers/start-app.js
import routesTo from './routes-to';

export default function startApp(attrs) {
  //...
```

Group of helpers in one file

An alternative approach is to create a bunch of helpers wrapped in a self calling function, like this

```
// helpers/custom-helpers.js
var customHelpers = function() {
  Ember.Test.registerHelper('myGreatHelper', function (app) {
    //do awesome test stuff
  });

  Ember.Test.registerAsyncHelper('myGreatAsyncHelper', function (app) {
    //do awesome test stuff
  });
}();

export default customHelpers;
```

which can be used in `start-app.js`

```
// helpers/start-app.js
import customHelpers from './custom-helpers';

export default function startApp(attrs) {
  //...
```

Once your helpers are defined, you'll want to ensure that they are listed in the `.jshintrc` file within the test directory.

```
// /tests/.jshintrc
{
  "predef": [
    "document",
    //...
    "myGreatHelper",
    "myGreatAsyncHelper"
```

 (https://github.com/ember-cli/ember-cli.github.io/blob/master/_posts/2013-04-09-asset-compilation.md)

Asset Compilation

Raw Assets

- `public/assets` VS `app/styles`

To add images, fonts, or other assets, place them in the `public/assets` directory. For example, if you place `logo.png` in `public/assets/images`, you can reference it in templates with `assets/images/logo.png` or in stylesheets with `url('/assets/images/logo.png')`.

This functionality of Ember-CLI comes from broccoli-asset-rev (<https://github.com/rickharrison/broccoli-asset-rev>). Be sure to check out all the options and usage notes.

JS Transpiling

Ember-cli automatically transpiles future javascript (ES6/ES2015, ES2016 and beyond) into standard ES5 javascript that runs on every browser using Babel JS (<https://babeljs.io>) with the ember-cli-babel (<https://github.com/babel/ember-cli-babel>) addon.

The default configuration handles most project's needs, but you can provide options to the transpile to disable specific transformations if by example your app only targets ES6 capable browsers, or on the contrary enable transpilation of some experimental features that not enabled by default just yet.

You can configure how babel works using the `babel` option in `ember-cli-build.js`. By example for disabling ES6/2015 features you'd do:

```
// ember-cli-build.js
var EmberApp = require('ember-cli/lib/broccoli/ember-app');

module.exports = function(defaults) {
  var app = new EmberApp(defaults, {
    babel: {
      blacklist: [
        'es6.arrowFunctions',
        'es6.blockScoping',
        'es6.classes',
        'es6.destructuring',
        'es6.parameters',
        'es6.properties.computed',
        // ...more options
      ]
    }
  });

  //...
  return app.toTree();
};
```

This options are just forwarded to Babel. Ember-cli uses Babel 5.X at the moment and you can check its documentation for a comprehensive list of all available transformations (<https://github.com/babel/babel.github.io/blob/5.0.0/docs/usage/transformers/index.md>) and options (<https://github.com/babel/babel.github.io/blob/5.0.0/docs/usage/options.md>).

Work is being done for upgrading to Babel 6. You can track the progress and help (<https://github.com/ember-cli/ember-cli/issues/5015>).

Minifying

The compiled css-files are minified by `broccoli-clean-css` or `broccoli-cssso`, if it is installed locally. You can pass minifier-specific options to them using the `minifyCSS:options` object in your `ember-cli-build`. Minification is enabled by default in the production-env and can be disabled using the `minifyCSS:enabled` switch.

Similarly, the js-files are minified with `broccoli-uglify-js` in the production-env by default. You can pass custom options to the minifier via the `minifyJS:options` object in your `ember-cli-build`. To enable or disable JS minification you may supply a boolean value for `minifyJS:enabled`.

For example, to disable minifying of CSS and JS, add in `ember-cli-build.js`:

```
// ember-cli-build.js
var EmberApp = require('ember-cli/lib/broccoli/ember-app');

module.exports = function(defaults) {
  var app = new EmberApp(defaults, {
    minifyJS: {
      enabled: false
    },
    minifyCSS: {
      enabled: false
    }
  });

  //...
  return app.toTree();
};
```

Exclude from minification

To exclude assets from `dist/assets` from being minificated, one can pass options for `broccoli-uglify-sourcemap` (<https://github.com/ef4/broccoli-uglify-sourcemap>) like so:

```
// ember-cli-build.js
var EmberApp = require('ember-cli/lib/broccoli/ember-app');

module.exports = function(defaults) {
  var app = new EmberApp(defaults, {
    minifyJS: {
      options: {
        exclude: ["**/vendor.js"]
      }
    }
  });

  //...
  return app.toTree();
};
```

This would exclude the resulting `vendor.js` file from being minificated.

Source Maps

Ember CLI()

Ember CLI supports producing source maps for your concatenated and minified JS source files.

Source maps are configured by the `EmberApp` `sourcemaps` option, and are disabled in production by default. Pass `sourcemaps: {enabled: true}` to your `EmberApp` constructor to enable source maps for javascript. Use the `extensions` option to add other formats, such as coffeescript and CSS:

`{extensions: ['js', 'css', 'coffee']}`. JS is supported out-of-the-box. CSS is not currently supported. For other source formats (Sass, Coffee, etc) refer to their addons.

Default `ember-cli-build.js`:

```
import EmberApp from 'ember-cli/lib/broccoli/ember-app';
var app = new EmberApp({
  sourcemaps: {
    enabled: EmberApp.env() !== 'production',
    extensions: ['js']
  }
});
```

Stylesheets

Ember CLI supports plain CSS out of the box. You can add your CSS styles to `app/styles/app.css` and it will be served at `assets/application-name.css`.

For example, to add bootstrap in your project you need to do the following:

```
bower install bootstrap --save
```

In `ember-cli-build.js` add the following:

```
app.import('bower_components/bootstrap/dist/css/bootstrap.css');
```

it's going to tell Broccoli (<https://github.com/joliss/broccoli>) that we want this file to be concatenated with our `vendor.css` file.

To use a CSS preprocessor, you'll need to install the appropriate Broccoli (<https://github.com/joliss/broccoli>) plugin. When using a preprocessor, Broccoli is configured to look for an `app.less`, `app.scss`, `app.sass`, or `app.styl` manifest file in `app/styles`. This manifest should import any additional stylesheets.

All your preprocessed stylesheets will be compiled into one file and served at `assets/application-name.css`.

If you would like to change this behavior, or compile to multiple output stylesheets, you can adjust the Output Paths Configuration

CSS

To use plain CSS with `app.css`:

- Write your styles in `app.css` and/or organize your CSS into multiple stylesheet files and import these files with `@import` from within `app.css`.
- CSS `@import` statements (<https://developer.mozilla.org/en-US/docs/Web/CSS/@import>) (e.g. `@import 'typography.css';`) must be valid CSS, meaning `@import` statements *must* precede all other rules and so be placed at the *top* of `app.css`.
- In the production build, the `@import` statements are replaced with the contents of their files and the final minified, concatenated single CSS file is built to `dist/assets/yourappname-FINGERPRINT_GOES_HERE.css`.
- Any individual CSS files are also built and minified into `dist/assets/` in case you need them as standalone stylesheets.
- Relative pathing gets changed (how to customize?)

Example `app.css` with valid `@import` usage:

```
/* @imports must appear at top of stylesheet to be valid CSS */
@import 'typography.css';
@import 'forms.css';

/* Any CSS rules must go *after* any @imports */
.first-css-rule {
  color: red;
}
...
```

CSS Preprocessors

To use one of the following preprocessors, all you need to do is install the appropriate NPM module. The respective files will be picked up and processed automatically.

LESS

To enable LESS (<http://lesscss.org/>), you'll need to add `ember-cli-less` (<https://github.com/gdub22/ember-cli-less>) to your NPM modules.

```
ember install ember-cli-less
```

SCSS/SASS

To enable SCSS/SASS (<http://sass-lang.com/>), you'll need to install the `ember-cli-sass` (<https://github.com/aemachina/ember-cli-sass>) addon to your project (*defaults to .scss, .sass allowed via configuration*).

```
ember install ember-cli-sass
```

You can configure your project to use `.sass` in your `ember-cli-build.js`:

```
Ember CLI (/)
ember-cli-build.js
var app = new EmberApp(defaults, {
  sassOptions: {
    extension: 'sass'
  }
});
```

Compass

To use Compass (<http://compass-style.org/>) with your ember-cli app, install ember-cli-compass-compiler (<https://github.com/quaertym/ember-cli-compass-compiler>) addon using NPM.

```
ember install ember-cli-compass-compiler
```

Stylus

To enable Stylus (<http://learnboost.github.io/stylus/>), you must first add ember-cli-stylus (<https://github.com/drewcovi/ember-cli-stylus>) to your NPM modules:

```
ember install ember-cli-stylus
```

CoffeeScript

To enable CoffeeScript (<http://coffeescript.org/>), you must first add ember-cli-coffeescript (<https://github.com/kimroen/ember-cli-coffeescript>) to your NPM modules:

```
ember install ember-cli-coffeescript
```

The modified `package.json` should be checked into source control. CoffeeScript can be used in your app's source and in tests, just use the `.coffee` extension on any file.

The ES6 module transpiler does not directly support CoffeeScript, but using them together is simple. Use the ``` character to escape out to JavaScript from your `.coffee` files, and use the ES6 syntax there:

```
# app/models/post.coffee
`import Ember from 'ember'`
`import User from 'appkit/models/user'`

Post = Ember.Object.extend
  init: (userId) ->
    @set 'user', User.findById(userId)

`export default Post`
```

Note that earlier versions of the transpiler had explicit support for CoffeeScript, but that support has been removed.

EmberScript

To enable EmberScript (<http://emberscript.com>), you must first add broccoli-ember-script (<https://github.com/aradabaugh/broccoli-ember-script>) to your NPM modules:

```
npm install broccoli-ember-script --save-dev
```

Note that the ES6 module transpiler is not directly supported with Emberscript, to allow use of ES6 modules use the ``` character to escape raw Javascript similar to the CoffeeScript example above.

Emblem

For Emblem (<http://emblemjs.com/>), run the following commands:

```
ember install ember-cli-emblem
```

If you're using the older broccoli-emblem-compiler addon, you need to switch to ember-cli-emblem. The older broccoli-emblem-compiler compiles directly to JS instead of Handlebars and therefore is broken on all newer version of HTMLBars.

Fingerprinting and CDN URLs

Fingerprinting is done using the addon broccoli-asset-rev (<https://github.com/rickharrison/broccoli-asset-rev>) (which is included by default).

When the environment is production (e.g. `ember build --environment=production`), the addon will automatically fingerprint your js, css, png, jpg, and gif assets by appending an md5 checksum to the end of their filename (e.g. `assets/yourapp-9c2cbd818d09a4a742406c6cb8219b3b.js`). In addition, your html, js, and css files will be re-written to include the new name. There are a few options you can pass in to `EmberApp` in your `ember-cli-build.js` to customize this behavior.

- `enabled` - Default: `app.env === 'production'` - Boolean. Enables fingerprinting if true. **True by default if current environment is production.**
- `exclude` - Default: `[]` - An array of strings. If a filename contains any item in the exclude array, it will not be fingerprinted.
- `ignore` - Default: `[]` - An array of strings. If a filename contains any item in the ignore array, the contents of the file will not be processed for fingerprinting.
- `extensions` - Default: `['js', 'css', 'png', 'jpg', 'gif', 'map']` - The file types to add md5 checksums.
- `prepend` - Default: `''` - A string to prepend to all of the assets. Useful for CDN urls like `https://subdomain.cloudfront.net/`
- `replaceExtensions` - Default: `['html', 'css', 'js']` - The file types to replace source code with new checksum file names.

- `customHash` - When specified, this is appended to fingerprinted filenames instead of the md5. Pass `null` to suppress the hash, which can be useful when using `Ember CLI`.

As an example, this `ember-cli-build` will exclude any file in the `fonts/169929` directory as well as add a cloudfront domain to each fingerprinted asset.

```
// ember-cli-build.js
var app = new EmberApp({
  fingerprint: {
    exclude: ['fonts/169929'],
    prepend: 'https://subdomain.cloudfront.net/'
  }
});
```

The end result will turn

```
<script src="assets/appname.js">
background: url('/images/foo.png');
```

into

```
<script src="https://subdomain.cloudfront.net/assets/appname-342b0f87ea609e6d349c7925d86bd597.js">
background: url('https://subdomain.cloudfront.net/images/foo-735d6c098496507e26bb40ecc8c1394d.png');
```

You can disable fingerprinting in your `ember-cli-build.js` :

```
// ember-cli-build.js
var app = new EmberApp({
  fingerprint: {
    enabled: false
  }
});
```

Or remove the entry from your `EmberApp` and `broccoli-asset-rev` from your `package.json` .

Application Configuration

Application configurations from your `ember-cli-build.js` file will be stored inside a special meta tag in `dist/index.html` .

sample meta tag:

```
<meta name="user/config/environment" content="%7B%22modulePre.your.config"%>
```

This meta tag is required for your ember application to function properly. If you prefer to have this tag be part of your compiled javascript files instead, you may use the `storeConfigInMeta` flag in `ember-cli-build.js` .

```
// ember-cli-build.js
var app = new EmberApp({
  storeConfigInMeta: false
});
```

Configuring output paths

The compiled files are output to the following paths:

Assets	Output File
<code>app/index.html</code>	<code>/index.html</code>
<code>app/*.js</code>	<code>/assets/application-name.js</code>
<code>app/styles/app.css</code>	<code>/assets/application-name.css</code>
other CSS files in <code>app/styles</code>	same filename in <code>/assets</code>
JavaScript files you import with <code>app.import()</code>	<code>/assets/vendor.js</code>
CSS files you import with <code>app.import()</code>	<code>/assets/vendor.css</code>

To change these paths, specify the `outputPaths` config option in `ember-cli-build.js` . The default setting is shown here:

Ember CLI

```
// ember-cli-build.js
var app = new EmberApp({
  outputPaths: {
    app: {
      html: 'index.html',
      css: {
        'app': '/assets/application-name.css'
      },
      js: '/assets/application-name.js'
    },
    vendor: {
      css: '/assets/vendor.css',
      js: '/assets/vendor.js'
    }
  }
});
```

You may edit any of these output paths, but make sure to update your `app.outputPaths.app.html` , default it is `index.html` , and `tests/index.html` .

```
// ember-cli-build.js
var app = new EmberApp({
  outputPaths: {
    app: {
      js: '/assets/main.js'
    }
  }
});
```

The `outputPaths.app.css` option uses a key value relationship. The *key* is the input file and the *value* is the output location. Note that we do not include the extension for the input path, because each preprocessor has a different extension.

When using CSS preprocessing, only the `app/styles/app.scss` (or `.less` etc) is compiled. If you need to process multiple files, you must add another key:

```
// ember-cli-build.js
var app = new EmberApp({
  outputPaths: {
    app: {
      css: {
        'app': '/assets/application-name.css',
        'themes/alpha': '/assets/themes/alpha.css'
      }
    }
  }
});
```

Integration

When using Ember inside another project, you may want to launch Ember only when a specific route is accessed. If you're preloading the Ember javascript before you access the route, you have to disable `autoRun` :

```
// ember-cli-build.js
var app = new EmberApp({
  autoRun: false
});
```

To manually run Ember: `require("app-name/app")["default"].create({/* app settings */});`

Subresource integrity

SRI calculation is done using the `addon ember-cli-sri` (<https://github.com/JonathanKingston/ember-cli-sri>) (which is included by default).

This plugin is used to generate SRI integrity (<http://www.w3.org/TR/SRI/>) for your applications. Subresource integrity is a security concept used to check JavaScript and stylesheets are loaded with the correct content when using a CDN.

Why

The reason to add this to your application is to protect against poisoned CDNs breaking JavaScript or CSS.

- JavaScript DDoS prevention (<https://blog.cloudflare.com/an-introduction-to-javascript-based-ddos/>)
 - The latest GitHub DDoS attack (<http://googleonlinesecurity.blogspot.co.uk/2015/04/a-javascript-based-ddos-attack-as-seen.html>)
- Protection against corrupted code on less trusted servers

Customize

To customize SRI generation see: `ember-cli-sri` (<https://github.com/JonathanKingston/ember-cli-sri>)

 (https://github.com/ember-cli/ember-cli.github.io/blob/master/_posts/2013-04-08-managing-dependencies.md)

Managing Dependencies

NPM and Bower Configuration

Ember CLI uses NPM (<https://www.npmjs.com>) and Bower (<http://bower.io/>) for dependency management. Both configuration files (`package.json` for NPM and `bower.json` for Bower) are located at the root of your Ember CLI project, and together they list all the dependencies for your project. Changes to your dependencies should be managed through these files, rather than manually installing packages individually.

Executing `npm install` will install all of the dependencies listed in `package.json` in one step. Similarly, executing `bower install` will install all of the dependencies listed in `bower.json` in one step.

Ember CLI is configured to have git ignore your `bower_components` and `node_modules` directories by default. Using the Bower and NPM configuration files allows collaborators to fork your repo and get their dependencies installed locally by executing `npm install` and `bower install` themselves.

Ember CLI watches `bower.json` for changes. Thus it reloads your app if you install new dependencies via `bower install <dependencies> --save`. If you install NPM dependencies via `npm install <dependencies> --save`, you will need to restart your Ember CLI server session manually.

Further documentation about NPM and Bower is available at their official documentation pages:

- Bower (<http://bower.io/>)
- NPM (<https://www.npmjs.com>)

Note that it is often easiest to install Ember addon dependencies using the `ember install` command, which will save all dependencies to the correct configuration files and run any further setup steps required.

Compiling Assets

Ember CLI uses the Broccoli (<https://github.com/broccolijs/broccoli>) assets pipeline.

The assets manifest is located in the `ember-cli-build.js` file in your project root (not the default `ember-cli-build.js`).

To add an asset specify the dependency in your `ember-cli-build.js` before calling `app.toTree()`. You can only import assets that are within the `bower_components` or `vendor` directories. The following example scenarios illustrate how this works.

Javascript Assets

Standard Non-AMD Asset

First, provide the asset path as the first and only argument:

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

From here you would use the package as specified by its documentation, usually a global variable. In this case it would be:

```
import Ember from 'ember';
/* global moment */
// No import for moment, it's a global called `moment`

// ...
var day = moment('Dec 25, 1995');
```

Note: Don't forget to make JSHint happy by adding a `/ global MY_GLOBAL */` to your module, or by defining it within the `predefs` section of your `.jshintrc` file.*

Alternatively, you could generate an ES6 shim to make the library accessible via `import`.

First, generate the shim:

```
ember generate vendor-shim moment
```

Next, provide the vendor asset path:

```
app.import('vendor/shims/moment.js');
```

Finally, use the package by adding the appropriate `import` statement:

```
import moment from 'moment';

// ...
var day = moment('Dec 25, 1995');
```

Standard Named AMD Asset

Provide the asset path as the first argument, and the list of modules and exports as the second:

```
app.import('bower_components/ic-ajax/dist/named-amd/main.js');
```

To use this asset in your app, import it. For example, with `ic-ajax`, when to use `ic.ajax.raw`:

```
import { raw as icAjaxRaw } from 'ic-ajax';
//...
icAjaxRaw( /* ... */ );
```

Standard Anonymous AMD Asset

Provide the asset path as the first argument, and the desired module name in the second:

```

Ember CLI (7)
app.import('/bower_components/ic-ajax/dist/amd/main.js', {
  using: [
    { transformation: 'amd', as: 'ic-ajax' }
  ]
});

```

To use this asset in your app, import it. For example, with `ic-ajax`, when to use `ic.ajax.raw`:

```

import { raw as icAjaxRaw } from 'ic-ajax';
//...
icAjaxRaw( /* ... */ );

```

Environment Specific Assets

If you need to use different assets in different environments, specify an object as the first parameter. That object's key should be the environment name, and the value should be the asset to use in that environment.

```

app.import({
  development: 'bower_components/ember/ember.js',
  production: 'bower_components/ember/ember.prod.js'
});

```

If you need to import an asset in one environment but not import it or any alternatives in other environments then you can wrap `app.import` in an `if` statement.

```

if (app.env === 'development') {
  app.import('vendor/ember-renderspeed/ember-renderspeed.js');
}

```

Customizing a built-in Asset

This is somewhat non-standard and discouraged, but suppose that due to a requirement in your application that you need to use the full version of Handlebars even in the production environment. You would simply provide the path to the `EmberApp` constructor:

```

var app = new EmberApp({
  vendorFiles: {
    'handlebars.js': {
      production: 'bower_components/handlebars/handlebars.js'
    }
  }
});

```

Alternatively, if you want to exclude the built-in asset from being automatically included in `vendor.js`, you can set its value to `false`:

```

var app = new EmberApp({
  vendorFiles: {
    'handlebars.js': false
  }
});

```

Note: The built-in assets are required dependencies needed by the environment to run your app. If you use the above method to specifically exclude some, you should still be including them in some other way.

Whitelisting and Blacklisting Assets

You can limit which dependencies in your `package.json` file get imported into your Ember application by using the `addon` option of the `EmberApp` constructor. A `whitelist` parameter allows you to restrict modules to a specific list. A `blacklist` parameter excludes specific modules from being imported into your app:

```

var app = new EmberApp({
  addon: {
    blacklist: [
      'fastboot-app-server'
    ]
  }
});

```

Test Assets

You may have additional libraries that should only be included when running tests (such as `qunit-bdd` or `sinon`). These can be imported into your app in your `ember-cli-build.js`:

```
// ember-cli-build.js
var EmberApp = require('ember-cli/lib/broccoli/ember-app'),
    isProduction = EmberApp.env() === 'production';

var app = new EmberApp();

if ( !isProduction ) {
  app.import( app.bowerDirectory + '/sinonjs/sinon.js', { type: 'test' } );
  app.import( app.bowerDirectory + '/sinon-qunit/lib/sinon-qunit.js', { type: 'test' } );
}

module.exports = app.toTree();
```

Notes: - Be sure to pass { type: 'test' } as the second argument to app.import . This will ensure that your libraries are compiled into the test-support.js file.

Styles

Static CSS

Provide the asset path as the first argument:

```
app.import('bower_components/foundation/css/foundation.css');
```

All style assets added this way will be concatenated and output as /assets/vendor.css .

Dynamic Styles (SCSS, LESS, etc)

The vendor trees that are provided upon instantiation are available to your dynamic style files. Take the following example (in app/styles/app.scss):

```
@import "bower_components/foundation/scss/normalize.scss";
```

Other Assets

Using app.import()

All other assets like images or fonts can also be added via import() . By default, they will be copied to dist/ as they are.

```
app.import('bower_components/font-awesome/fonts/fontawesome-webfont.ttf');
```

This example would create the font file in dist/font-awesome/fonts/fontawesome-webfont.ttf .

You can also optionally tell import() to place the file at a different path. The following example will copy the file to dist/assets/fontawesome-webfont.ttf .

```
app.import('bower_components/font-awesome/fonts/fontawesome-webfont.ttf', {
  destDir: 'assets'
});
```

If you need to load certain dependencies before others, you can set the prepend property equal to true on the second argument of import() . This will prepend the dependency to the vendor file instead of appending it, which is the default behavior.

```
app.import('bower_components/es5-shim/es5-shim.js', {
  type: 'vendor',
  prepend: true
});
```

If you need some of your assets to be included into specific file you can provide an outputFile option for your import:

```
// ember-cli-build.js
app.import('vendor/dependency-1.js', { outputFile: 'assets/additional-script.js'});
app.import('vendor/dependency-2.js', { outputFile: 'assets/additional-script.js'});
```

As a result both dependencies will end up in dist/assets/additional-script.js in the same order they were specified.

Note: outputFile works only for javascript and css files.

Using broccoli-funnel

With the broccoli-funnel (<https://github.com/broccolijs/broccoli-funnel>) package, (parts of) a bower-installed package can be used as assets as-is. First ensure that the Broccoli package needed to build is installed:

```
npm install broccoli-funnel --save-dev
```

Add this import to the top of ember-cli-build.js , just below the EmberApp require:

```
var Funnel = require('broccoli-funnel');
```

Within ember-cli-build.js , we merge assets from a bower dependency with the main app tree:

```

module.exports = function(defaults) {
  ...

  // Copy only the relevant files. For example the WOFF-files and stylesheets for a webfont:

  var extraAssets = new Funnel('bower_components/a-lovely-webfont', {
    srcDir: '/',
    include: ['**/*.woff', '**/stylesheet.css'],
    destDir: '/assets/fonts'
  });

  // Providing additional trees to the `toTree` method will result in those
  // trees being merged in the final output.

  return app.toTree(extraAssets);
}

```

In the above example the assets from the fictive bower dependency called `a-lovely-webfont` can now be found under `/assets/fonts/`, and might be linked to from `index.html` like so:

```
<link rel="stylesheet" href="assets/fonts/lovelyfont_bold/stylesheet.css">
```

You can exclude assets from the final output in a similar fashion. For example, to exclude all `.gitkeep` files from the final output:

```

// Again, add this import to the top of `ember-cli-build.js`, just below the `EmberApp` require:
var Funnel = require('broccoli-funnel');

// Normal ember-cli-build contents

// Filter toTree()'s output
var filteredAssets = new Funnel(app.toTree(), {
  // Exclude gitkeeps from output
  exclude: ['**/.gitkeep']
});

// Export filtered tree
module.exports = filteredAssets;

```

Note: `broccoli-static-compiler` (<https://github.com/joliss/broccoli-static-compiler>) is deprecated. Use `broccoli-funnel` (<https://github.com/broccolijs/broccoli-funnel>) instead.

✍ (https://github.com/ember-cli/ember-cli.github.io/blob/master/_posts/2013-04-05-environments.md)

Environments

Ember-CLI ships with support for managing your application's environment. Ember-CLI will build an environment config file at `config/environment`. Here, you can define an ENV object for each environment (development, test and production). For now, this is limited to the three environments mentioned.

The ENV object contains two important keys: 1) `EmberENV`, and 2) `APP`. The first can be used to define Ember feature flags (see the [Feature Flags guide](http://emberjs.com/guides/configuring-ember/feature-flags/) (<http://emberjs.com/guides/configuring-ember/feature-flags/>)). The second can be used to pass flags/options to your application instance.

You can access these environment variables in your application code by importing from `../config/environment` or `your-application-name/config/environment`.

For example:

```

import ENV from 'your-application-name/config/environment';

if (ENV.environment === 'development') {
  // ...
}

```

Ember-CLI assigns `ENV.EmberENV` to `window.EmberENV`, which Ember reads on application initialization.

Additionally, Ember-CLI contains a number of environment-dependent helpers for assets:

- Env specific assets
- Env specific asset fingerprinting

It is now also possible to override command line options by creating a file in your app's root directory called `.ember-cli` and placing desired overrides in it.

For example, a common desire is to change the port (<http://stackoverflow.com/questions/24003944/save-port-number-for-ember-cli-in-a-config-file>) that ember-cli serves the app from. It's possible to pass the port number directly to ember server in the command line, e.g. `ember server --port 8080`. If you wish to make this change a permanent configuration change, make the `.ember-cli` file and add the options you wish to pass to the server in a hash.

```

{
  "port": 8080
}

```

✍ (https://github.com/ember-cli/ember-cli.github.io/blob/master/_posts/2013-04-04-deployments.md)

Deployments

You can easily deploy your Ember CLI application to a number of places using `ember-cli-deploy` (<http://ember-cli-deploy.com/>). Or you can follow some of the recipes below.

Heroku

Prerequisites:

- An Ember CLI application
- Heroku Account (<https://www.heroku.com>)
- Heroku Toolbelt (<https://toolbelt.heroku.com>)

Official Buildpack and instructions can be found here (<https://github.com/heroku/heroku-buildpack-ember-cli>).

Azure

Continuous deployment with Azure Websites (<http://www.azure.com>) is enabled through Microsoft's module `ember-cli-azure-deploy` (<https://github.com/felixrieseberg/ember-cli-azure-deploy>). The installation is simple just run the following commands in your Ember CLI app's root directory:

```
npm install --save-dev -g ember-cli-azure-deploy
azure-deploy init
```

Next, set up your Azure Website's source control to point to your repo - either via GitHub, BitBucket, VSO or any of the other available options (<http://azure.microsoft.com/en-us/documentation/articles/web-sites-publish-source-control/#Step4>). As soon as you push a new commit to your repository, Azure Websites will automatically run `ember build` and deploy the contents of the created `dist` directory to your website's `wwwroot`.

Firebase

To deploy your Ember CLI application to Firebase, you'll first need to enable hosting from your Firebase's Dashboard. Then, install the Firebase Tools (<https://github.com/firebase/firebase-tools>):

```
npm install -g firebase-tools
```

You can then configure your application for deployment by running the following in your app's root directory and following the prompts:

```
firebase init
```

Finally, to deploy your application, run:

```
firebase deploy
```

For more configuration options, check out Firebase's Hosting Guide (<https://www.firebase.com/docs/hosting/guide/>).

History API and Root URL

If you are deploying the app to somewhere other than the `rootURL` (`/`), you will need to configure the value of `rootURL` in `config/environment.js`. This is required for the History API, and thus also the Router, to function correctly.

For example

```
// config/environment.js
if (environment === 'production') {
  ENV.rootURL = '/path/to/ember/app/';
}
```

This will also be used as a prefix for assets, eg `/path/to/ember/app/assets/vendor.js`. However when building for production the value of `prepend` for `fingerprint` will be used instead. So for

```
ember build --prod
```

with

```
// ember-cli-build.js
module.exports = function(defaults) {
  var app = new EmberApp(defaults, {
    // Add options here
    fingerprint: {
      prepend: 'https://cdn.example.com/'
    }
  });
};
```

the asset URLs will not use `rootURL` and will instead be: `https://cdn.example.com/assets/vendor-3b1b39893d8e34a6d0bd44095afcd5c4.js`.

As of version 2.7, `baseURL` is deprecated and `rootURL` should be used instead. See this blog post (<http://emberjs.com/blog/2016/04/28/baseURL.html>) for more details.

Content Security Policy

Ember CLI (7)

For those interested in enhanced security for their web application, they should consider the setting up a content-security policy even for development. That way security violations can be discovered immediately, rather than in production.

For more information, see the `ember-cli-content-security-policy` README. (<https://github.com/rwjblue/ember-cli-content-security-policy>)

Deploying an HTTPS server using Nginx on a Unix/Linux/MacOSx machine

The following is a simple deployment with https using nginx. Http just redirects to the https server here. Don't forget to include your ssl keys in your config.

Before deployment make sure you run this command to populate the dist directory:

```
ember build --environment="production"
```

File: nginx.conf

```
## Nginx Production Https Ember Server Configuration

## https site##
server {
    listen      443 default;
    server_name <your-server-name>;
    #root       /usr/share/nginx/html;
    root        <root path to an ember /dist directory>;
    index       index.html index.htm;

    # log files
    access_log  /var/log/nginx/<your-server-name>.access.log;
    error_log   /var/log/nginx/<your-server-name>.error.log;

    # ssl files
    ssl on;
    keepalive_timeout 60;

    # include information on SSL keys, cert, protocols and ciphers
    # SSL Labs.com is a great resource for this, along with testing
    # your SSL configuration: https://www.ssllabs.com/projects/documentation/

    # proxy buffers
    proxy_buffers 16 64k;
    proxy_buffer_size 128k;

    ## default location ##
    location / {
        include /etc/nginx/mime.types;
        try_files $uri $uri/ /index.html?$request_uri;
    }
}

## http redirects to https ##
server {
    listen      80;
    server_name <your-server-name>;

    # Strict Transport Security
    add_header Strict-Transport-Security max-age=2592000;
    rewrite ^/.*$ https://$host$request_uri? permanent;
}
```

 (https://github.com/ember-cli/ember-cli.github.io/blob/master/_posts/2013-04-03-upgrading.md)

Upgrading

Upgrading an Ember CLI App

Steps to upgrade to the latest version of Ember CLI are included with the release notes for each release (<https://github.com/ember-cli/ember-cli/releases>).

 (https://github.com/ember-cli/ember-cli.github.io/blob/master/_posts/2013-04-03-common-issues.md)

Common Issues

npm Package Management with `sudo`

Installing packages such as `bower` with `sudo` powers can lead to permissions issues and ultimately to problems installing dependencies. See <https://gist.github.com/isaacs/579814> (<https://gist.github.com/isaacs/579814>) for a collection of various solutions.

Installing From Behind a Proxy

Ember CLI (/)

If you're behind a proxy, you might not be able to install because Ember CLI—or some of its dependencies—tries to `git clone` a `git://` URL. (In this scenario, only `http://` URLs will work).

You'll probably get an error like this:

```
npm ERR! git clone git://github.com/jgable/esprima.git Cloning into bare repository '/home/<username>/npm/_git-remotes/git-github-com-jgable-esprima-git-d221af32'...
npm ERR! git clone git://github.com/jgable/esprima.git
npm ERR! git clone git://github.com/jgable/esprima.git fatal: unable to connect to github.com:
npm ERR! git clone git://github.com/jgable/esprima.git github.com[0: 192.30.252.129]: errno=Connection timed out
npm ERR! Error: Command failed: fatal: unable to connect to github.com:
npm ERR! github.com[0: 192.30.252.129]: errno=Connection timed out
```

As a workaround you can configure `git` to make the translation:

```
git config --global url."https://".insteadOf git://
```

Using Canary Build instead of release

For Ember: `bower install ember#canary --resolution canary` For ember-data: `npm install --save-dev emberjs/data#master`

Windows Build Performance Issues

See The Windows Section for more details.

PhantomJS on Windows

When running tests on Windows via PhantomJS the following error can occur:

```
events.js:72
throw er; // Unhandled 'error' event
^
Error: spawn ENOENT
at errnoException (child_process.js:988:11)
at Process.ChildProcess._handle.onexit (child_process.js:779:34)
```

In order to fix this ensure the following is added to your `PATH` :

```
C:\Users\USER_NAME\AppData\Roaming\npm\node_modules\phantomjs\lib\phantom
```

Cygwin on Windows

Node.js on Cygwin is no longer supported more details (<https://github.com/nodejs/node/wiki/Installation#building-on-cygwin>) Rather than using Cygwin, we recommend running Ember CLI natively on windows, or via the new Windows Subsystem Linux (https://msdn.microsoft.com/en-us/commandline/wsl/install_guide).

Usage with Docker

When building your own Docker (<http://docker.com>) image to build Ember applications and run tests, there are a couple of pitfalls to avoid. * PhantomJS requires `bzip2` and `fontconfig` to already be installed. * After installing PhantomJS, you will need to manually link PhantomJS to `/usr/local/bin` if that is not done by the install process. * Testem uses the `which` command to locate PhantomJS, so you must install `which` if it is not included in your base OS.

Usage with Vagrant

Vagrant (<http://vagrantup.com>) is a system for automatically creating and setting up development environments that run in a virtual machine (VM).

Running your Ember CLI development environment from inside of a Vagrant VM will require some additional configuration and will carry a few caveats.

Ports

In order to access your Ember CLI application from your desktop's web browser, you'll have to open some forwarded ports into your VM. Ember CLI by default uses two ports.

- For serving assets the default is `4200` . Can be configured via `--port 4200` .
- For live reload there is no default. Can be configured via `---live-reload-port=9999` .

To make Vagrant development seamless these ports will need to be forwarded.

```
Vagrant.configure("2") do |config|
  # ...
  config.vm.network "forwarded_port", guest: 4200, host: 4200
  config.vm.network "forwarded_port", guest: 9999, host: 9999
end
```

Watched Files

The way Vagrant syncs directories between your desktop and vm may prevent file watching from working correctly. This will prevent rebuilds and live reloads from working correctly. There are several work arounds:

1. Watch for changes by polling the file system via: `ember serve --watcher polling`.
2. Use NFS for synced folders (<https://docs.vagrantup.com/v2/synced-folders/nfs.html>).

VM Setup

When setting up your VM, install Ember CLI dependencies as you normally would. Some of these dependencies (such as broccoli-sass) may have native dependencies that may require recompilation. To do so run:

```
npm rebuild
```

Provider

The two most common Vagrant providers, VirtualBox and VMware Fusion, will both work. However, VMware Fusion is substantially faster and will use less battery life if you're on a laptop. As of now, VirtualBox will use 100% of a single CPU core to poll for file system changes inside of the VM.

✍️ (https://github.com/ember-cli/ember-cli.github.io/blob/master/_posts/2013-04-02-practices-windows.md)

Windows

Windows

Windows Vista and newer windows versions are fully supported.

To get started ensure the following dependencies are installed:

- Node.js - <https://nodejs.org/en/> (<https://nodejs.org/en/>)
- Git - <https://git-scm.com/> (<https://git-scm.com/>)
- Phantom.js - <http://phantomjs.org/> (<http://phantomjs.org/>)

Performance

Although supported, Windows performance, at least by default, isn't as good as on Linux or MacOS. On a positive note, this story continues to improve. Both Microsoft, and the Ember CLI team continue to work to improve these developer ergonomics.

What causes the build slowdown?

The two primary reasons are:

- Lack of enabled-by-default symlinks
- Generally slower FS operations on NTFS

For the best possible Windows experience

- Windows 10, insiders release with development mode enabled. Symlinks are enabled by default) Details from Microsoft (<https://blogs.windows.com/buildingapps/2016/12/02/symlinks-windows-10/>)
- or, Windows Subsystem Linux Installation Guide (https://msdn.microsoft.com/en-us/commandline/wsl/install_guide)

Improving your Windows experience

Ensure Search and Defender ignore your project's `tmp` directory:

```
npm install --save-dev ember-cli-windows-addon
```

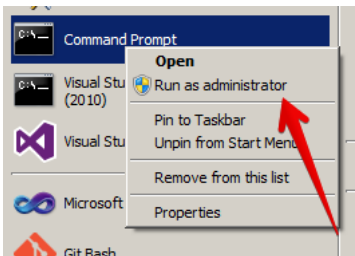
Then, to start the automatic configuration, run:

```
ember windows
```

Read more about this from the Microsoft DX Open Source team (<http://felixrieseberg.com/improved-ember-cli-performance-with-windows/>)

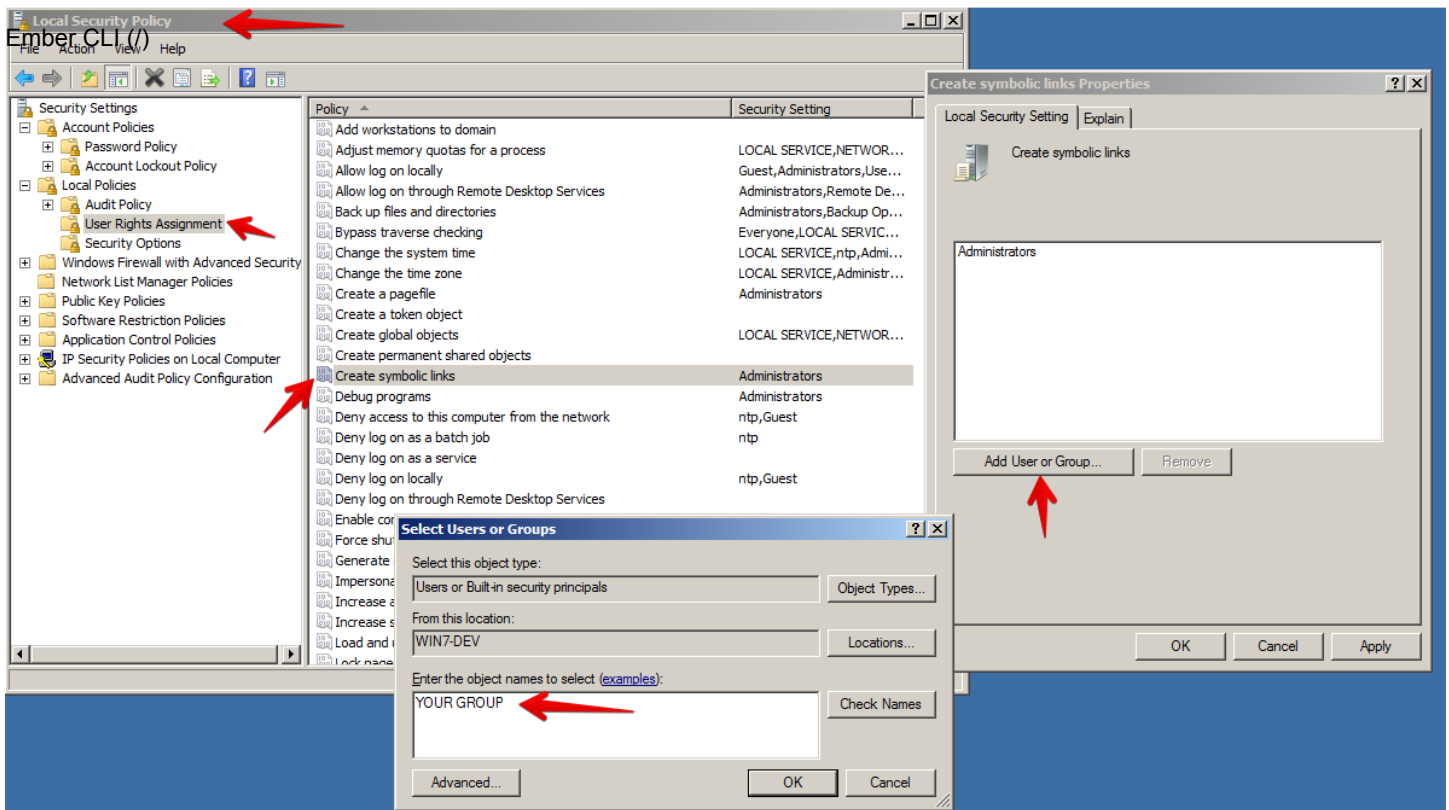
Enabling symlinks

To create symlinks the account running Ember CLI must have the `SeCreateSymbolicLinkPrivilege`. Users in the Administrators group have this permission already. However, if UAC (User Access Control) is enabled, users in the Administrators group must run their shell using Run As Administrator because UAC strips away certain permissions from the Administrators +group, including `SeCreateSymbolicLinkPrivilege`.



If the user account is not part of the Administrators group you will need to add the `SeCreateSymbolicLinkPrivilege` to allow the creation of symlinks. To do this open the Local Security Policy by typing Local Security Policy in the Windows Run Box.

Under Local Policies -> User Rights Assignment find the Create symbolic links policy and double click it to add a new user or group. Once your user or group has been added, your user should be able to create symlinks. Keep in mind if your user is part of the Administrators group and UAC is enabled you will still need to start your shell using Run as Administrator.



Issues With npm: EEXIST, Path too Long, etc

There were always two major issues with running Node.js on Windows: first and foremost, the operating system maintains a maximum length for path names, which clashes with Node's traditional way of nesting modules in `node_modules`. The second issue was a bit more subtle: The npm installer had a set of steps it executed for each package and it would immediately start executing them as soon as it decided to act on a package resulting in hard-to-debug race conditions.

`npm@3` is a nearly complete rewrite of `npm`, fixing both issues. Windows users of Ember CLI might want to make the switch to `npm@3` to benefit from its flat module installation (solving most issues involving long path names) as well as its multi-stage installer.

✍️ (https://github.com/ember-cli/ember-cli.github.io/blob/master/_posts/2013-04-02-editors.md)

Editors

Atom

If you are using Atom (<https://atom.io>) with `ember-cli`, there are some packages available specific to Ember development.

Atom -> Preferences -> Install

- `ember-cli-helper` (<https://atom.io/packages/ember-cli-helper>) - ember-cli integration in Atom
- `ember-tabs` (<https://atom.io/packages/ember-tabs>) - Makes atom.io work better with Ember pods

Emacs

If you are using Emacs (<https://www.gnu.org/software/emacs/>) with `ember-cli`, Emacs creates temporary backup, autosave, and lockfiles that interfere with broccoli watcher, so they need to either be moved out of the way or disabled. To do that, ensure your emacs configuration contains the following:

```
(setq backup-directory-alist `((".*" . ,temporary-file-directory)))
(setq auto-save-file-name-transforms `((".*" ,temporary-file-directory t)))
(setq create-lockfiles nil)
```

An `ember-mode` (<https://github.com/madnificent/ember-mode>) package is also available. It has shortcuts for quickly navigating files in ember projects, running generators, and running build, serve, and test tasks. It also includes support for linking build errors to files and minibuffer notifications of `ember serve` status. It can be installed from MELPA (<http://melpa.org/>). To use MELPA, ensure your configuration contains the following:

```
(require 'package)
(add-to-list 'package-archives
  '("melpa" . "http://melpa.org/packages/") t)
(package-initialize)
```

Then `ember-mode` can be installed from the package menu at `M-x package-list-packages`. After it is installed, add a file named `.dir-locals.el` to the root of your ember projects with the contents:

```
((nil . ((mode . ember))))
```

to enable it inside those projects.
Ember CLI (/)

Sublime Text

If you are using Sublime Text (<http://www.sublimetext.com>) with `ember-cli`, by default it will try to index all files in your `tmp` directory for its GoToAnything functionality. This will cause your computer to come to a screeching halt @ 90%+ CPU usage, and can significantly increase build times. Simply remove these directories from the folders Sublime Text watches:

Sublime Text -> Preferences -> Settings - User

```
// folder_exclude_patterns and file_exclude_patterns control which files
// are listed in folders on the side bar. These can also be set on a per-
// project basis.
"folder_exclude_patterns": [".svn", ".git", ".hg", "CVS", "tmp/class-*", "tmp/es_*", "tmp/jshinter*", "tmp/replace_*", "tmp/static_compiler*", "tm
p/template_compiler*", "tmp/tree_merger*", "tmp/coffee_script*", "tmp/concat-tmp*", "tmp/export_tree*", "tmp/sass_compiler*"]
```

WebStorm

If you are using WebStorm (<https://www.jetbrains.com/webstorm/>) with `ember-cli`, you will need to modify your `.gitignore` file, enable ECMAScript6 settings, and mark certain directories.

First, add the following line to `.gitignore`:

```
.idea
```

Next, from the WebStorm menu:

File > Settings -> Languages & Frameworks -> JavaScript -> ECMAScript6

Click 'OK' to close the Settings modal window.

Next, in Webstorm's Project window right-click on each of the following directories, go to 'Mark Directory As' and mark as indicated:

Mark as Excluded:

```
/tmp
/dist
```

Mark as Resource Root:

```
/
/bower_components
/bower_components/ember-qunit/lib
/public
```

Mark as Test Sources Root:

```
/tests
```

Intellij-emberjs

This plugin (<https://github.com/Turbo87/intellij-emberjs>) provides excellent Ember.js support for all JetBrains IDEs that support JavaScript, including WebStorm.

In order to install it, go to File | Settings... | Plugins | Browse repositories... search for `Ember.js` and hit the Install button.

Vim

If you are using Vim (<http://www.vim.org/>) with `ember-cli`, Vim creates temporary backups and autosaves which interfere with broccoli, so they need to either be moved out of the way or disabled. To do that, ensure your `.vimrc` contains the following:

```
set backupdir=~/.vim/backup//
set directory=~/.vim/swap//
set undodir=~/.vim/undo//
```

And make sure to create the directories- `mkdir -p ~/.vim/backup`; `mkdir -p ~/.vim/swap`; `mkdir -p ~/.vim/undo`

 (https://github.com/ember-cli/ember-cli.github.io/blob/master/_posts/2012-05-01-tutorials.md)

tutorials

Getting Started

- Discover Ember (online course) (<https://www.ludu.co/course/ember>)
- ember-cli-101 (book) (<http://leanpub.com/ember-cli-101>)
- Rails + Ember.js (with the Ember CLI) (<https://www.devmynd.com/blog/2014-7-rails-ember-js-with-the-ember-cli-redux>)
- Rock and Roll with EmberJS (book) (<http://balinterdi.com/rock-and-roll-with-emberjs/>)
- Ember.js tutorial (<http://yoember.com>) (Free, with the latest Ember 2 and Ember CLI)

Testing

Ember CLI (/)

- Creating an Integration Test with Ember.js (Video) (https://www.youtube.com/watch?v=2O24ltr0pPU&feature=youtu.be&list=PLxP_o-ABjKLFuDpuJ2Tw_3__OzxE7kFnh)

Addons

- Introducing Ember CLI Addons (http://reefpoints.dockyard.com/2014/06/24/introducing_ember_cli_addons.html)
- Building Ember CLI Addons Simply (<http://hashrocket.com/blog/posts/building-ember-addons>)
- Updating EmberCLI addons to the latest ember-cli (Video) (https://www.youtube.com/watch?v=XOMGGJ_d4wl)

Deploying

- Lightning Fast Deployments With Rails (in the Wild). (<http://blog.abuiles.com/blog/2014/07/08/lightning-fast-deployments-with-rails>)

Tooling

- Installing Node.js/NPM Without Sudo (<http://www.wenincode.com/installing-node-jsnpm-without-sudo>)

Example Apps

- ember-cli/ember-cli-todos (<https://github.com/ember-cli/ember-cli-todos>)
- stefanpenner/ember-jobs (firebase example) (<https://github.com/stefanpenner/ember-jobs>)
- abuiles/facturas-client (<https://github.com/abuiles/facturas-client>)

Made by Leo (<http://leo.im>), jhdesign (<http://jonharmondesign.com/>) and contributors (<https://github.com/ember-cli/ember-cli.github.io/graphs/contributors>)

