Intro to Ember.js – Day three cheatsheet

Templating

Templating guide

Escaping/unescaping strings in templates

Marking a string as html safe, to allow HTML output:

```
"my <html> string".htmlSafe()
```

Escaping a string, to prevent HTML output:

```
Ember.Handlebars.Utils.escapeExpression("my <html> string")
```

Logging to the console

```
{{log <path>}}
```

Creating bound input fields / textareas

Input helpers guide

There are `{{input}}` and `{{textarea}}` helpers you can use as if they were standard HTML5 `<input>` or `<textarea>` elements

```
{{input type="text" value=<path>}}
```

```
{{input type="password" value=<path>}}
```

```
{{input type="checkbox" checked=<path>}}
```

```
{{textarea value=<path> rows=<number> cols=<number>}}
```

Controllers

Controllers guide

Generating a controller

`ember generate controller <name>` (name should correlate to a route)

Transitioning to another route from in a controller

`this.transitionToRoute(<route name>, <dynamic segments>)`

Resetting a controller's state when leaving it's route

Inside your controller's route, use the `resetController` method:

```
export default Ember.Route.extend({
  resetController(controller) {
    // your code here
  }
})
```

Getting a controller reference inside of a route

`this.controllerFor(<name of controller>)`

Actions

Actions guide

Defining an action

```
export default Ember.Controller.extend({
  actions: {
    <action-name>: function() {
      <your code here>
    }
  }
})
```

Triggering an action from a template

`<button {{action "<name>"}}>fire the action</button>`

Computed properties

Getter shorthand

```
foo: Ember.computed(<zero or more dependent keys>, function()
{
  // your code here
})
```

Getter/setter

```
foo: Ember.computed(<zero or more dependent keys>, {
  get() {
    // your code here
  },
  set(key, value) {
    // your code here
  }
})
```

Services

Services guide

Generating a service

`ember generate service <name>`

Injecting a service

```
export default Ember.Component.extend({
  cart: Ember.inject.service(<name>)
});
```

Comparison matrix of components, controllers, and services

	Components	Controllers	Services
State:	Transient	Persistent	Persistent
Lifespan:	Short-lived	Long-lived	Long-lived
Singleton?	No	Yes	Yes
Template context?	Yes	Yes	No

Object system

[Classes and instances guide]

Creating a new subclass

To subclass an existing class object, just call `.extend()` on it. For example:

```
Ember.Route.extend({
  // your code here
})
```

Generating a new mixin

`ember generate mixin <name>`

Using a mixin

Pass the mixin to your class's `extend()`, for example:

```
import MyMixin from '../mixins/my-mixin';

export default Ember.Object.extend(MyMixin, {
  // class definition
})
```

Addons

[Developing addons and blueprints guide]

Creating an addon

Outside of your app directory:

`ember addon <name>`

The addon project created follows these structure conventions. Some of these directories may be created upon the generation of your first file of that type.

```
app/ - merged with the application’s namespace.
addon/ - part of the addon’s namespace.
blueprints/ - contains any blueprints that come with the addon
, each in a separate directory
public/ - static files which will be available in the applicat
ion as /your-addon/*
test-support/ - merged with the application’s tests/
tests/ - test infrastructure including a “dummy” app and accep
tance test helpers.
vendor/ - vendor specific files, such as stylesheets, fonts, e
xternal libs etc.
ember-cli-build.js - Compilation configuration
package.json - Node meta-data, dependencies etc.
index.js - main Node entry point (as per npm conventions)
```

Generating files in an addon

Works the same as inside an app:

`ember generate <type> <name>`

Developing your addon

`ember serve`

Installing your addon in an app

`ember install <local file path to addon dir>`

Adding bower dependencies to your addon

First, create your addons' "default blueprint." This is ran automatically when your addon is installed using `ember install`. A default blueprint is simply a blueprint with the same name as your addon.

`ember generate blueprint <addon-name>`

Second, open up the generated blueprint's `index.js` file and addon the following code:

```
module.exports = {
  normalizeEntityName: function() {}, // no-op since we're jus
t adding dependencies

  afterInstall: function() {
    return this.addBowerPackageToProject('x-button'); // is a
promise
  }
};
```

Third, wire up the Bower dependency's imports inside of your addon's `index.js` file, which will be at the very top level of your addon directory. Note, an `app.bowerDirectory` property is available for usage in addons. Bower's install directory is configurable, so addons should be agnostic to their configuration.

```
module.exports = {
  name: 'ember-cli-x-button',

  included: function(app) {
    this._super.included(app);

    app.import(app.bowerDirectory + '/x-button/dist/js/x-butto
n.js');
    app.import(app.bowerDirectory + '/x-button/dist/css/x-butt
on.css');
  }
};
```

Developing your addon while it's installed in an app

Normally, when an addon is installed into an app, it's copied from the original location. Modifying the addon's code in it's original location will not update the installed copy. Addons are built on top of npm packages, so we can utilize npm's `link` functionality to make developing our addons inside of an app easier.

Inside of your addon, register it with npm:

`npm link`

This will setup up a global symlink.

Then inside of your app, declare your intent to use the addon via a symlink:

`npm link <addon-name>`

You should now have the addon installed in your app via a symlink, rather than it being a copy.

To enable LiveReload / watching of your addon's code, return `true` from the `isDevelopingAddon` method in your addon's top level `index.js` file:

```
isDevelopingAddon: function() {
  return true;
}
```