

[Submission](#)[Next→](#)

1 Introduction

Overview and Purpose

In this project you'll rework Bloc Jams to use the Angular JavaScript framework.

Objectives

After this project, you should be able to:

- Bootstrap Angular to an application and create an Angular module.
- Configure routing and states for an application.
- Implement controllers for an application's views.
- Create a service that controls song playback.
- Write a custom directive that controls song and volume sliders.
- Create a custom time code filter.

Use Case

jQuery is a great tool for adding animations and effects to a page, but it's difficult to build a sophisticated frontend application with jQuery alone. In this project, you'll refactor Bloc Jams using **AngularJS**, commonly referred to as Angular.

What Is Angular?

Angular is a JavaScript framework for building dynamic web applications, primarily **CRUD (Create, Read, Update, Delete)** applications. Although its purpose is to simplify application development, Angular is not an all-purpose framework. In particular, it's not suitable for static, content-heavy sites.

Instead, Angular helps build **single-page applications (SPAs)**, which are web applications that do not require page loads when navigating between pages as most web sites do. SPAs aim to provide a user experience akin to desktop applications. Some prominent SPAs include **Gmail**, **Medium**, and **Virgin America**.

Angular allows us to extend HTML syntax (tags and attributes) with *directives*. We'll learn more about directives later in the project, but for now, know that they attach particular behavior(s) to DOM elements. For example, in this HTML, we've added a directive to the `<div>`:

```
<div ng-click="doSomething()">
```

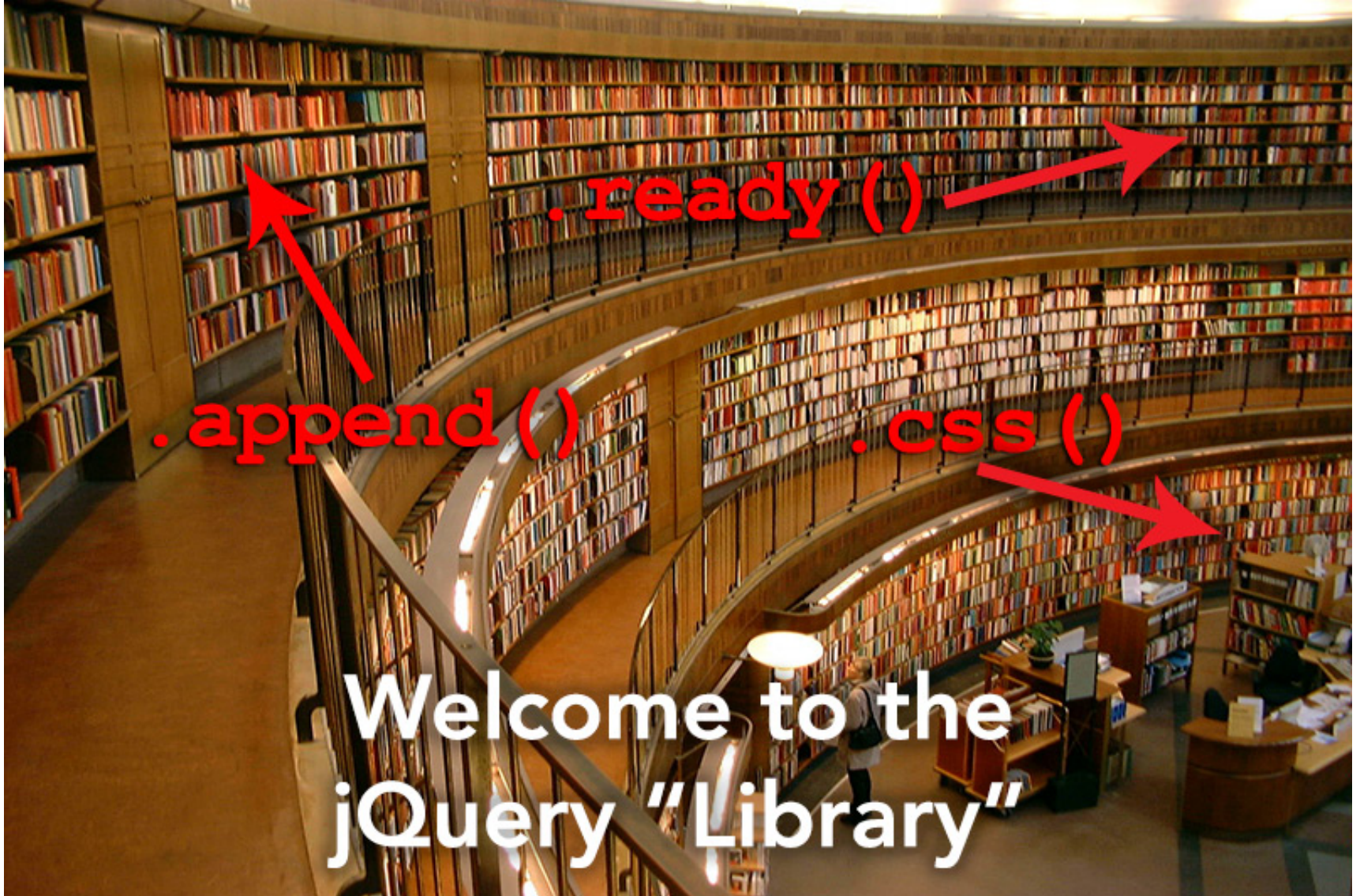
`ng-click` is the directive that tells Angular to execute `doSomething()` when a user clicks the `<div>`. Angular has some built-in directives like `ngClick`, but, more excitingly, we can write our own.

Another key feature of Angular is its **two-way data binding**. As a user interacts with an application, the data populated in the view updates in real time to reflect the changes made by the user. **Visit this example** to see data binding in action.

Framework vs. Library

The terms *framework* and *library* are sometimes used interchangeably and can cause confusion. The use of these terms often depends on the context. We distinguish the two below:

A **library**, such as jQuery, is a collection of prewritten code consisting of common tasks that simplify development. Our code is "in charge" and uses a library to retrieve specific functions from the collection:



A **framework**, like Angular, provides the basic structure of an application. The framework is "in charge" and our code fills in the details:



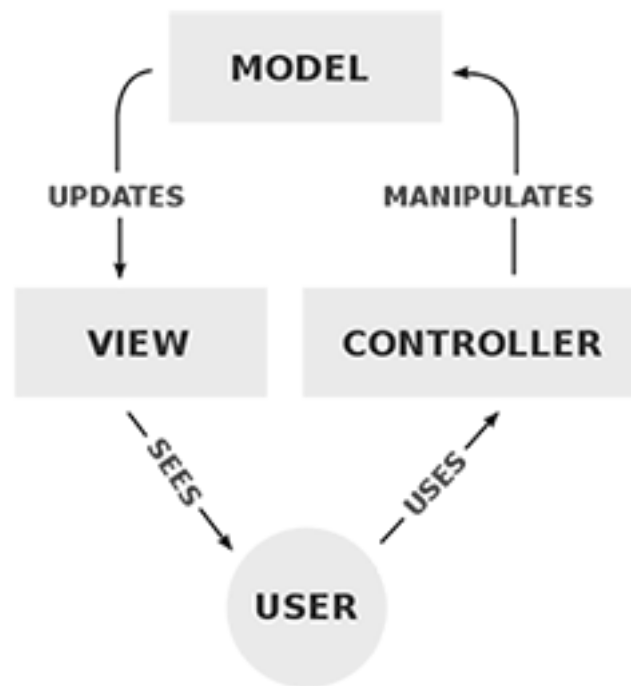
Architectural Patterns

Architectural patterns are reusable solutions to common, recurring problems in software architecture. Two patterns developers most often use with Angular are **Model-View-Controller**, or MVC, and **Model-View-ViewModel**, or MVVM.

For both, the Model is the data of an application. The View is an output representation of that data, or the UI.

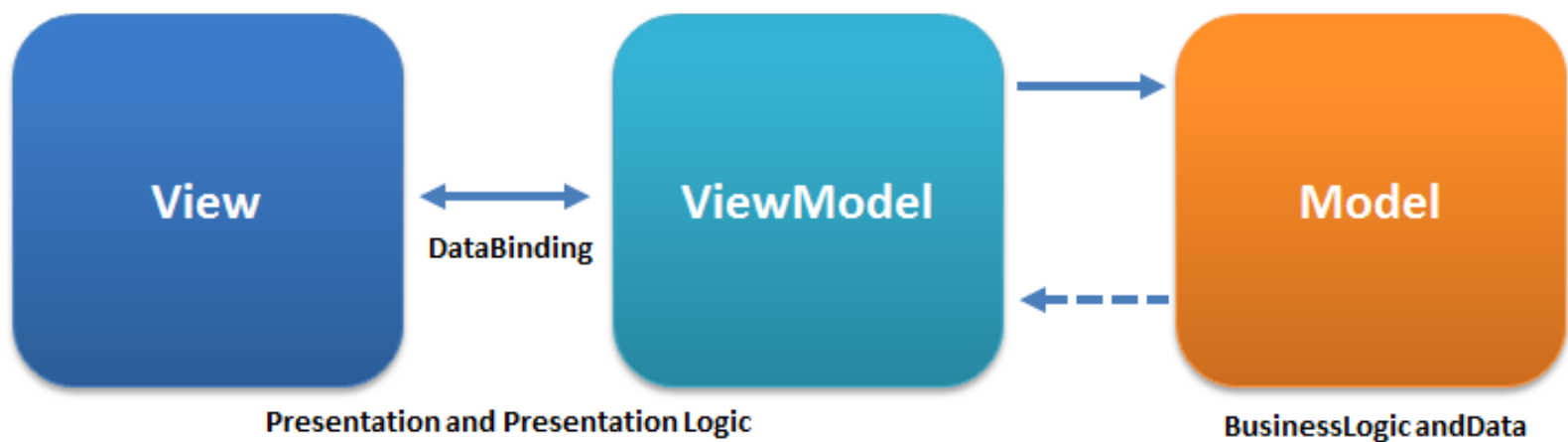
With MVC, a user interacts with the Controller, and based on the user's input, the Controller conducts transactions on behalf of the Model. The Controller manipulates the Model and passes the Model to the View. The View renders the Model into an appropriate output format.

A simplified diagram of these interactions looks like this:



With MVVM, a user interacts with the View which is always in sync with the ViewModel. The ViewModel represents the View and exposes the data objects from the Model in a way that makes them manageable; that is, the ViewModel provides data binding between the View and the Model. MVVM was designed to support data-binding functions with less effort.

To illustrate MVVM, it might look like this:



Source: **Wikimedia Foundation**

Angular began as an MVC framework, but is now closer to MVVM. Don't concern yourself with semantics, however. Most often your architecture will not be purely MVC or MVVM, and that's okay. Although Angular has "shifted" architecture, Angular is often considered **an MV* framework**, meaning Model-View-*Whatever*. Whether you choose to use MVC or MVVM or a combination of the two, do *whatever works for you*.

Modules

An **Angular module** acts as a container for the different parts – the services, directives, controllers, filters, and configuration information (all of which we'll learn about soon) – of an application.

Modules are about *encapsulation*, a tenet of object-oriented programming. Recall the concept of scope, whereby an object created within a function exists in the function's local scope, not the global scope. Functions created in the global scope can be overwritten by other scripts. For this reason, avoid declaring functions in the global **namespace**.

Note: **Linters** are tools that check code quality. JSHint, a JavaScript linter, **can check variable shadowing** to see if a variable is overwriting another.

With modules, Angular alleviates unintentional shadowing by keeping functions and values local to an application's module.

Advice

AngularJS introduces a lot of new terminology which can make it difficult to absorb the framework's concepts. If you struggle with the terms or concepts, don't fret. We encourage you to do some things during this project:

- Read the documentation. In particular, **version 1.6.4** because it's the version of Angular this project will use to refactor Bloc Jams.
- Click the links provided throughout a checkpoint for additional, relevant information on a concept. Most of the links take you to specific spots in the documentation. Bookmark them and refer to them often.
- Take notes. Keep track of new terms either in a file on your computer or **by hand in a notebook** (it's effective). We support **learning by writing**.
- Communicate with your mentor. Seek their guidance when a concept becomes difficult to understand.

Resources

If you want more practice or resources to refer to while you learn Angular, here are some recommendations:

- **AngularJS Tutorial**
- **Learn Angular**
- **Todd Motto: AngularJS Tutorial**



User Stories




User Story	Difficulty Rating
As a developer, I want to bootstrap Angular to my application.	1
As a developer, I want to configure routing and states for my application.	2
As a developer, I want to implement controllers for my application's views.	2
As a developer, I want to create a service that handles song playback.	3

As a developer, I want to write a directive that controls song and volume sliders.	4
As a developer, I want to add a time code filter to display time properly.	1

Later user stories often rely on the completion of the former, therefore, work on them in the order prescribed.

How would you rate this checkpoint and assignment?



1. Introduction		
 Assignment	 Discussion	 Submission



2 Configuration

The Bloc Starter Application

Because Angular lends itself to the single-page application architecture, it is better suited to a different application structure than Bloc Jams in the Foundation. SPAs that change the URLs of an application require a server to intercept their requests and allow the frontend of the application to handle what's displayed in the browser.

To accommodate these differences, clone the Bloc Frontend Starter Project that includes a NodeJS server:

Install NodeJS

Open [this link](#) and follow the instructions to learn more about NodeJS and NPM.

Setup the Application

```
$ git clone -b version-0.0.3-without-grunt https://github.com/Bloc/bloc-frontend-project-starter.
```

1. `$ mv bloc-frontend-project-starter bloc-jams-angular`
2. Change into the directory: `$ cd bloc-jams-angular`
3. Run `$ npm install` to download/install the needed dependencies
4. Start the node server: `npm start`
5. Open a web browser and go to `http://localhost:3000`
6. Press `cntrl + c` to stop the server

Reset Git and Push To Github

This project was downloaded from Bloc's Github repository. That means git has already been initialized. The problem is that this repository is stored on Bloc's Github account and

we want our work to be on OUR Github account; not Bloc's. So we need to reset Git and push this code up to our own repository.

1. Remove the existing Git local repository and history: `$ rm -r -f .git`
2. Reinitialize Git: `$ git init`
3. Create a new GitHub repository at <http://www.github.com>
4. Add that remote to your project:

```
$ git remote add origin <link to your github repository>
```

Add the remaining parts of your application structure

Use either brackets or the terminal to create the following empty files and folders for your application.

```
├─ app
│   ├── assets
│   │   └─ images
│   │
│   ├── scripts
│   │   └─ app.js
│   ├── styles
│   │
│   └─ templates
│       └─ home.html
└─ index.html
```

We will go over all of these different parts as we go through the curriculum, but for now, it is good to have the skeleton in place.

Migrate CSS and Assets

Throughout this project we'll migrate files and code from the Foundation Bloc Jams to Angular Bloc Jams. Start copying the six CSS files in `bloc-jams/styles` into the `bloc-jams-angular/app/styles` directory.

Once copied, link to the files in `app/index.html`, along with the font stylesheets:

```
~/bloc/bloc-jams-angular/app/index.html
```

```

...
<head lang="en">
  <meta charset="UTF-8">
  <title>Bloc Jams Angular</title>
+   <link rel="stylesheet" type="text/css" href="http://fonts.googleapis.com/css?fam:
+   <link rel="stylesheet" type="text/css" href="http://code.ionicframework.com/ionic
+   <link rel="stylesheet" type="text/css" href="styles/normalize.css">
+   <link rel="stylesheet" type="text/css" href="styles/main.css">
+   <link rel="stylesheet" type="text/css" href="styles/landing.css">
+   <link rel="stylesheet" type="text/css" href="styles/collection.css">
+   <link rel="stylesheet" type="text/css" href="styles/album.css">
+   <link rel="stylesheet" type="text/css" href="styles/player_bar.css">
</head>
...

```

Next, update these lines to tailor the `<head>` content for Bloc Jams Angular:

```
~/bloc/bloc-jams-angular/app/index.html
```

```

...
<head lang="en">
  <meta charset="UTF-8">
+   <title>Bloc Jams Angular</title>
+   <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" type="text/css" href="http://fonts.googleapis.com/css?fam:
...

```

Lastly, copy the contents of `bloc-jams/assets` into the `bloc-jams-angular/app/assets` directory. Once copied, remove the `body.collection` style from `collection.css` and the `body.album` style from `album.css`. Instead, add a background image to `main.css`:

```
~/bloc/bloc-jams-angular/app/styles/main.css
```

```
...
body {
+   background-image: url(../assets/images/blurred_backgrounds/blur_bg_3.jpg);
+   background-repeat: no-repeat;
+   background-attachment: fixed;
+   background-position: center center;
+   background-size: cover;
    font-family: 'Open Sans';
    color: white;
    min-height: 100%;
+   padding-bottom: 200px;
}
...
```

Include Angular

Follow the steps below to initialize an Angular application.

1. Reference the Angular Script File

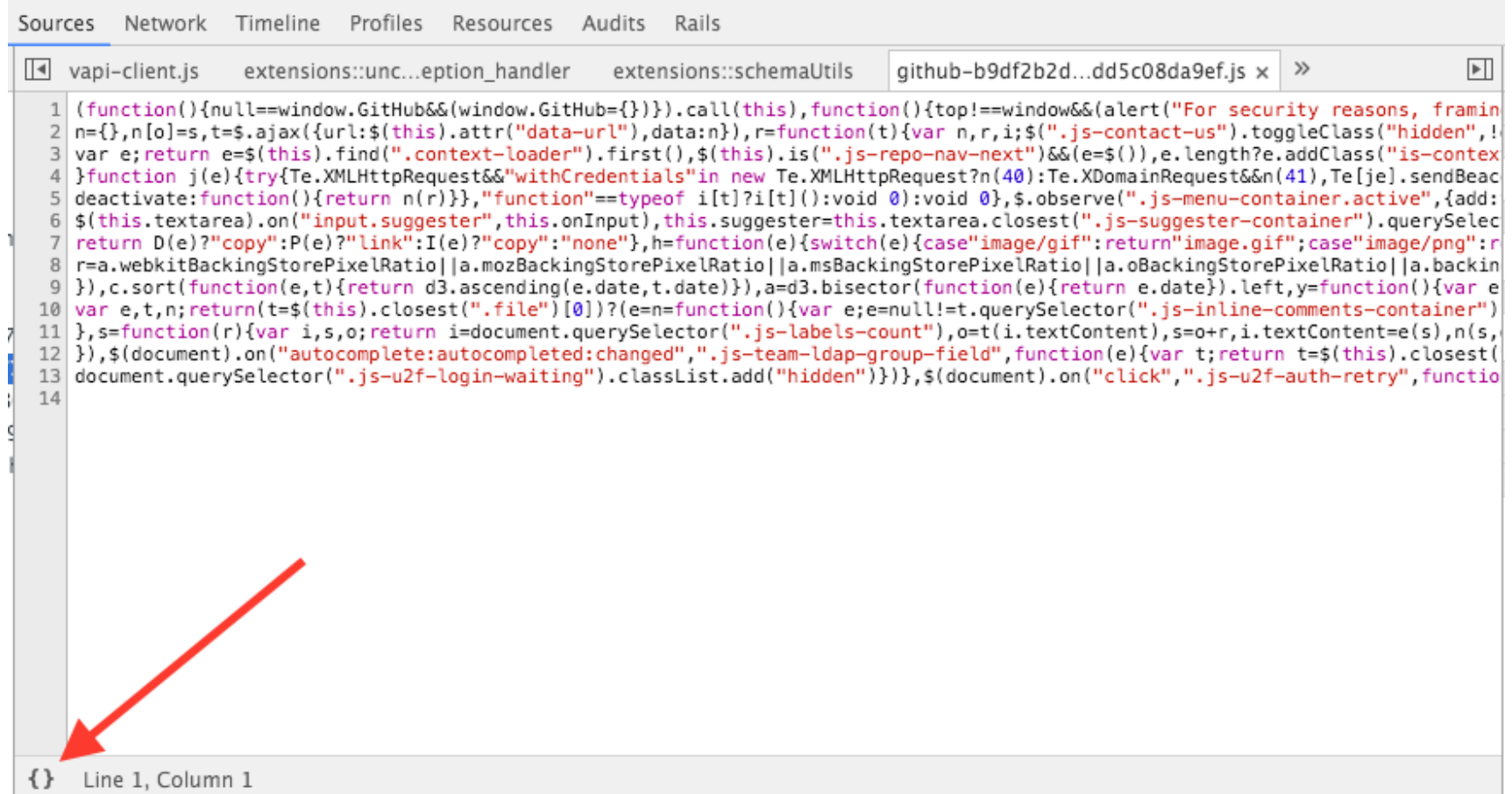
Include Angular the same way we include a library like jQuery, by adding a script source to the HTML document. Add a link to the Angular source in `index.html` and make sure that the link is above your other script tags:

~/bloc/bloc-jams-angular/app/index.html

```
...
+   <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.4/angular.min.js"></script>
    <script src="scripts/app.js"></script>
</body>
...
```

The source we've included above is the *minified* version of the script, denoted by the `.min` in the file name. **Minification** removes all the unnecessary characters, such as spaces and new lines, from source code and shortens variable names to the smallest number of letters possible (usually one) in order to reduce the file size. Alternatively, we can use the un-minified version of the script (`https://ajax.googleapis.com/ajax/libs/angularjs/1.6.4/angular.js`) to help when debugging issues, as the source is easier to read and dissect.

Chrome does, however, have a "pretty print" tool to un-minify JavaScript via a button in the bottom lefthand corner of a minified file in the Source tab:



Note that variable names will remain shortened even when using the Pretty Print tool.

When should I use minified source code? In a production application, because it optimizes the file size and loading speed, while reducing the bandwidth consumption of your site/application.

2. Declare an Angular Module

The root Angular module will act as a container for different parts of our application. In

`app.js`, define **a module** with `angular.module`:

```
~/bloc/bloc-jams-angular/app/scripts/app.js
```

```
angular.module('blocJams', []);
```

The first argument passed, `blocJams`, is the prescribed name of the module. The empty array, passed as the second argument, injects dependencies into an application. For now, there are no dependencies to inject, but we'll cover dependency injection in the next checkpoint.

3. Bootstrap the Application

To link the `blocJams` module to the application, Angular needs to know the root element of the application, which is typically the `<html>` or `<body>` element. In `index.html`, link the root module to the `<html>` tag:

```
~/bloc/bloc-jams-angular/app/index.html
```



```
...
<!DOCTYPE html>
- <html>
+ <html ng-app="blocJams">
  <head lang="en">
...

```

`ngApp` is a built-in Angular *directive*. This particular directive tells Angular where to bootstrap the defined application. For Bloc Jams, it's `<html>`, or the entire document. The `ng-app` activates the `blocJams` module for this part of the page.

What does `ng` stand for? `ng` has no particular significance. The [Angular FAQ](#) says that "ng" sounds like "Angular."

At this point, we've wired Angular to the application. The next steps are to configure the module and create a controller, which we'll cover in the following checkpoints.

Check your work

To ensure that Angular is up and running, add a simple Javascript expression inside double curly brackets `` to your index.html file.

~/bloc/bloc-jams-angular/app/index.html

```
<!DOCTYPE html>
<html>
...
<body>
  4
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.4/angular.min.js">
</body>
</html>

```

Then switch to your terminal. Start the server with: `$ npm start`. Open a new web browser window and navigate to: `http://localhost:3000`. You should see the expression evaluated on the webpage. So assuming you did `2 + 2` inside double curly brackets, you would see a white webpage with just the number 4 on it.

Once you have verified that Angular is up and running on your app, you can remove the `4` from index.html

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details.

How would you rate this checkpoint and assignment?



2. Configuration

 **Assignment**

 Discussion

 Submission



3 Routing and States

Most web applications refresh when we navigate to a different page on the site. When a user clicks a link, the application sends a request to the server for a new HTML file. The browser then reloads the entire page with the newly received information.

An alternative to this request/response process is to use single-page applications (SPA). They have engineering benefits as well as a snappier user interface that feels like a native desktop application.

Some disadvantages of SPAs include browser history management and client-side state. For example, when a user bookmarks a page, the application should reinitialize the page in the same state as when the user bookmarked it. With SPAs, implementing this ability is complicated.

Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

Templates

Let's look at a simple, conceptual example of an application. Suppose the `index.html`, `album.html`, and `collection.html` pages look like this:

```
index.html
```

```
<html>
  <head>
    <title>Bloc Jams</title>
  </head>
  <body>
    <h1>Landing</h1>
  </body>
</html>
```

album.html

```
<html>
  <head>
    <title>Bloc Jams</title>
  </head>
  <body>
    <h1>Album</h1>
  </body>
</html>
```

collection.html

```
<html>
  <head>
    <title>Bloc Jams</title>
  </head>
  <body>
    <h1>Collection</h1>
  </body>
</html>
```

There is a lot of redundant code because the only unique part of each page is the `<h1>` tag. This is the case with Bloc Jams, where each of the pages shares the same basic structure and navigation.

The browser also has to load a new HTML document when a user navigates between the pages, which can result in a slow experience. With Angular, we can pull the shared HTML into one **global file**, and each of the unique fragments into separate **templates**.

Templates, in this context, are HTML documents with Angular markup.

Using the example presented above, `index.html` could become the global file that works with three templates: `landing.html`, `album.html`, and `collection.html`:

index.html

```
<html>
  <head>
    <title>Bloc Jams</title>
  </head>
  <body>
    <ui-view></ui-view>
  </body>
</html>
```

templates/landing.html

```
<h1>Landing</h1>
```

templates/album.html

```
<h1>Album</h1>
```

templates/collection.html

```
<h1>Collection</h1>
```

Note the `ui-view` **directive** in the global file, `index.html`. This directive (with the help of an external module known as UI-Router) tells Angular where to place the templates when they're requested.

Create Templates for Bloc Jams

`index.html` will act as the global file for Bloc Jams. Add `<ui-view></ui-view>` to the global file:

~/bloc/bloc-jams-angular/app/index.html

```
...
<body>
  <nav class="navbar">
    ...
  </nav>

+   <ui-view></ui-view>

  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.4/angular.min.js"></script>
  <script src="/scripts/app.js"></script>
</body>
</html>
```

Remove the `home.html` file from the starter project `app/templates` directory – we won't use it for this project.

The Landing, Collection, and Album templates will consist of content unique to each template.

Create a `landing.html` file in the `app/templates` directory:

```
~/bloc/bloc-jams-angular/app/templates/landing.html
```

```

+ <section class="hero-content">
+   <h1 class="hero-title">Turn the music up!</h1>
+ </section>
+
+ <section class="selling-points container clearfix">
+   <div class="point column third">
+     <span class="ion-music-note"></span>
+     <h5 class="point-title">Choose your music</h5>
+     <p class="point-description">The world is full of music; why should you have
+   </div>
+   <div class="point column third">
+     <span class="ion-radio-waves"></span>
+     <h5 class="point-title">Unlimited, streaming, ad-free</h5>
+     <p class="point-description">No arbitrary limits. No distractions.</p>
+   </div>
+   <div class="point column third">
+     <span class="ion-iphone"></span>
+     <h5 class="point-title">Mobile enabled</h5>
+     <p class="point-description">Listen to your music on the go. This streaming s
+   </div>
+ </section>

```

When we view the landing page, we see that the selling points don't display. This is because `bloc-jams/scripts/landing.js` animated those elements to display with jQuery. In this project, we won't refactor the animation. In `bloc-jams-angular/app/styles/landing.css`, change the `opacity` of `.point` from `0` to `1` and remove the transition properties.

To display the Collection view with Angular, we'll need to move the template from `collection.js` back to HTML. Create a `collection.html` file in the `app/templates` directory:

```
~/bloc/bloc-jams-angular/app/templates/collection.html
```

```
+ <section class="album-covers container clearfix">
+   <div class="collection-album-container column fourth">
+     
+     <div class="collection-album-info caption">
+       <p>
+         <a class="album-name" href="album.html">The Colors</a>
+         <br/>
+         <a href="album.html">Pablo Picasso</a>
+         <br/>
+         X songs
+         <br/>
+       </p>
+     </div>
+   </div>
+ </section>
```

For the Album view, create an `album.html` file in the `app/templates` directory. Similar to the Collection view, move the song row template from `album.js` and place it back in the HTML:

~/bloc/bloc-jams-angular/app/templates/album.html

```
+ <main class="album-view container narrow">
+   <section class="clearfix">
+     <div class="column half">
+       
+     </div>
+     <div class="album-view-details column half">
+       <h2 class="album-view-title">The Colors</h2>
+       <h3 class="album-view-artist">Pablo Picasso</h3>
+       <h5 class="album-view-release-info">1909 Spanish Records</h5>
+     </div>
+   </section>
+   <table class="album-view-song-list">
+     <tr class="album-view-song-item">
+       <td class="song-item-number">1</td>
+       <td class="song-item-title">Blue</td>
+       <td class="song-item-duration">3:31</td>
+     </tr>
+   </table>
+ </main>
```


Note that the Album view does not include the player bar. You'll add that in the next checkpoint.

Routing

To display these templates in the view, Angular uses **routing**, which is organized around URL routes. Angular has a built-in router, but **many developers don't use it**.

Instead, developers use **UI-Router** because it is more flexible and features behaviors not found in the Angular tools. With UI-Router, an application can be in different **states** that determine what to display when a user navigates to a specific route.

UI-Router will take care of replacing the contents of `<ui-view></ui-view>` with a template when a user navigates to the proper route. Each template can be unique, while the shared code is kept in the global file. Since UI-Router uses JavaScript to switch the views, the browser won't load a new HTML document when a user navigates to a new route.

Add an External Module to an Angular Application

The Angular development community has useful external modules that makes adding more robust functionality to Angular applications easy. To add these external modules, we must do at least two things:

1. Include the script source in the application, below the Angular source.
2. Inject the module into the application's declaration.

Include the UI-Router Source

UI-Router is a separate module from the Angular source script. To include it in an application, add its source script after the Angular source. With a global file that contains all the subviews, we only need to add the source once:

```
~/bloc/bloc-jams-angular/app/index.html
```

```
...
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.4/angular.min.js"></script>
+ <script src="https://cdnjs.cloudflare.com/ajax/libs/angular-ui-router/0.4.2/angular-ui-router.min.js"></script>
<script src="/scripts/app.js"></script>
...
```

UI-Router is an extension of Angular so it must come *after* the Angular source or the application will throw a **Reference Error**.

Inject the UI-Router Module

Recall the `blocJams` module definition in `app.js`:

```
~/bloc/bloc-jams-angular/app/scripts/app.js
```

```
angular.module('blocJams', []);
```

The second argument, the empty array, is the list of external modules that `blocJams` depends on, known as **dependency injection**. After we add an external module's script source, we can inject the module into the application by adding it to the array:

```
~/bloc/bloc-jams-angular/app/scripts/app.js
```

```
- angular.module('blocJams', []);
+ angular.module('blocJams', ['ui.router']);
```

The `blocJams` module can now make use of the UI-Router module. We can add additional modules as strings in the array.

Configure the Module with Providers

With UI-Router, we need to set up state configuration using an Angular **provider**. **Providers** are services used by Angular modules to either configure or define default behavior for a certain Angular module. For Bloc Jams, we'll use two providers:

- `$stateProvider`: to configure the state behavior
- `$locationProvider`: to configure how the application handles URLs in the browser

To make sure the providers are accessible throughout the application, inject them using

the `config` block on the application's root module. Write a `config` function to pass into the `config()` function:

~/bloc/bloc-jams-angular/app/scripts/app.js

```
+ (function() {  
+     function config($stateProvider, $locationProvider) {  
+     }  
+  
- angular.module('blocJams', ['ui.router']);  
+     angular  
+         .module('blocJams', ['ui.router'])  
+         .config(config);  
+ })();
```

This style of module declaration is based on the [Opinionated Angular Styleguide](#).

Configure Paths with `$locationProvider`

`$locationProvider`, which is part of Angular's core, configures an **application's paths**. By default, Angular prefixes routes with `#!`, known as **Hashbang mode**. This is a convention for showing that a page load is triggered by JavaScript.

For example, if we navigate to a state with the path `/album`, the full URL will read `localhost:3000/#!/album` instead of `localhost:3000/album`. It doesn't look nice, but we can disable it. Add the following code to the `config` function:

~/bloc/bloc-jams-angular/app/scripts/app.js

```
function config($stateProvider, $locationProvider) {  
+     $locationProvider  
+         .html5Mode({  
+             enabled: true,  
+             requireBase: false  
+         });  
+  
+     ...  
}
```

By setting the `html5Mode` method's `enabled` property to `true`, the hashbang URLs are disabled; that is, users will see clean URLs without the hashbang. Setting the `requireBase` property to `false` is unrelated to the hashbang issue, but is one way to avoid **a common**

`$location` **error**.

Configure States with `$stateProvider`

`$stateProvider`, a component of UI-Router, will determine a number of properties for a state. For Bloc Jams, we'll need to know how to configure at least four aspects of a state: its name, URL route, controller, and template.

`$stateProvider` calls `.state()`, which takes two arguments: `stateName` and `stateConfig`. For example:

```
$stateProvider.state(stateName, stateConfig)
```

`stateName` is a *unique* string that identifies a state and `stateConfig` is an object that defines specific properties of the state. For Bloc Jams, create a state named `landing` and add an accompanying URL and template to the `stateConfig` object:

~/bloc/bloc-jams-angular/app/scripts/app.js

```
function config($stateProvider, $locationProvider) {
  $locationProvider
    .html5Mode({
      enabled: true,
      requireBase: false
    });

+   $stateProvider
+     .state('landing', {
+       url: '/',
+       templateUrl: '/templates/landing.html'
+     });
}
```

...

With this state configuration, when we navigate to `localhost:3000` or `localhost:3000/`, the `ui-view` directive in the global file (`index.html`) will load the template associated with the `landing` state.

Because `$stateProvider.state()` returns `$stateProvider`, we are able to call `state()` again without having to reference the `$stateProvider` variable. With no arguments passed to the `state()` call, this would look like `$stateProvider.state().state()`. This is called

method chaining. It's common to chain `state()` calls instead of calling them individually on `$stateProvider`. Add another state, named `album`:

~/bloc/bloc-jams-angular/app/scripts/app.js

```
...
    $stateProvider
      .state('landing', {
        url: '/',
        templateUrl: '/templates/landing.html'
-      });
+      })
+      .state('album', {
+        url: '/album',
+        templateUrl: '/templates/album.html'
+      });
...
```

Note that chained calls should not have a semicolon at the end of each call. If you include them, you will receive an error. Chained calls require only a single semicolon after the last call, which signifies the termination of the statement.

You may notice that we've put the `state()` calls on their own line. Because we removed the semicolon from the first `state()` call, JavaScript will look to the next line for a continuation. When chaining method calls, it is common to see each call happen on its own line.

With this configuration the `album` state, when we navigate to `localhost:3000/album`, the `ui-view` directive in the global file will load the `album` template.

Trigger a State

Instead of using anchor tags (`<a>`) with an `href` (short for "hyperlink reference"), UI-Router triggers states by attaching a `ui-sref` directive (short for "user interface state reference") in place of the `href`.

For example, instead of ``, a link refers to a state name like so: `<a ui-sref="album">`, where `album` is the name of the state to trigger.

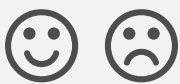
To trigger a state change (display a different view) when using UI-Router, use `ui-sref`




instead of `href`. For external page links or an internal anchor, continue to use `href`.

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details.

How would you rate this checkpoint and assignment?



3. Routing and States		
 Assignment	 Discussion	 Submission



4 Templates

We've established the basic views for Bloc Jams. The song list in the Album view, however, is missing the `onHover` and `offHover` ability that we wrote in the Foundation. We're also missing a crucial aspect of our application: the player bar. Before we fine tune these templates, let's learn a bit about "the Angular way".

Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

The Angular Way

An important difference between Angular and jQuery is the distinction between **"imperative" and "declarative" view manipulation**. Angular is designed so that views are open books which "declare" the functionality behind them. In a jQuery-powered view, any element could have "hidden" functionality attached to it, obscuring the view manipulation unless we dive into the JavaScript. In Angular, the associated JavaScript behavior of any element is declared clearly in the HTML.

Consider this jQuery code that we wrote for Bloc Jams in the Foundation:

```

var $previousButton = $('#.main-controls .previous');
var $nextButton = $('#.main-controls .next');

$(document).ready(function() {
    setCurrentAlbum(albumPicasso);
    $previousButton.click(previousSong);
    $nextButton.click(nextSong);
});

```

This code designates what should happen when a user clicks the `$previousButton` and `$nextButton` objects. The DOM elements are identified in the JavaScript. Angular, on the other hand, declares this code in the HTML:

```

<div class="main-controls">
  <a class="previous" ng-click="previousSong()">
    <span class="ion-skip-backward"></span>
  </a>
  <a class="next" ng-click="nextSong()">
    <span class="ion-skip-forward"></span>
  </a>
</div>

```

The functions `previousSong` and `nextSong` exist in the JavaScript, but Angular's `ng-click` syntax more clearly shows the elements that trigger some action when a user interacts with the application.

ngClick and ngShow Directives

The section above shows a use case for the `ngClick` **directive**. It is Angular's declarative equivalent of jQuery's `.click()` method.

Similarly, the `ngShow` **directive** is Angular's declarative equivalent of jQuery's `.show()` method. (Angular has an inverse of `ngShow` named `ngHide`.) From the documentation:

The `ngShow` directive shows or hides the given HTML element based on the expression provided to the `ngShow` attribute. The element is shown or hidden by removing or adding the `.ng-hide` CSS class onto the element. The `.ng-hide` CSS class is predefined in AngularJS and sets the `display` style to `none` (using an !important flag).

Here is an `ngShow` example from the Angular documentation:

```
<!-- when $scope.myValue is truthy (element is visible) -->
<div ng-show="myValue"></div>

<!-- when $scope.myValue is falsy (element is hidden) -->
<div ng-show="myValue" class="ng-hide"></div>
```

Refer to the MDN documentation for a refresher on **truthy** and **falsy**.

Update the Album View Template

Instead of using jQuery's imperative view manipulation to create `onHover` and `offHover` functions (**as shown here**), we'll declare this functionality in the view using Angular's `ngShow` directive.

Before we can add this declarative code, however, we need to decide what to show and when to show it:

What to Show	When to Show It
Song number	When the song is not playing <i>and</i> the mouse is off hover
Play button	When the song is not playing <i>and</i> the mouse is on hover
Pause button	When the song is playing

When a user first visits the Album view, each song row should display the song's number. When a user hovers over a song row, the song number should change to a play button if the song is not already playing. When a user clicks the play button, it should change to a pause button.

Using `ngShow`, we'll write some "wishful coding" with variables that we haven't yet declared. Update `album.html` with the following changes:

```
~/bloc/bloc-jams-angular/app/templates/album.html
```

```

...
- <td class="song-item-number">1</td>
+ <td class="song-item-number">
+   <span ng-show="!playing && !hovered">1</span>
+   <a class="album-song-button" ng-show="!playing && hovered"><span class="ion-play"></span></a>
+   <a class="album-song-button" ng-show="playing"><span class="ion-pause"></span></a>
+ </td>
...

```

We can incorporate the `hovered` variable using the straightforward `ngMouseover` and `ngMouseleave` directives. To replicate the Foundation behavior, add these directives to the table row:

```
~/bloc/bloc-jams-angular/app/templates/album.html
```

```

...
- <tr class="album-view-song-item">
+ <tr class="album-view-song-item" ng-mouseover="hovered = true" ng-mouseleave="hovered = false">
...

```

Both directives evaluate an expression. In this case, we assign the `hovered` variable a value of `true` or `false`.

At this stage, we should be able to see the play button appear on mouseover. We'll be able to display the pause button once we start to work on playing music in a later checkpoint.

Create a Template for the Player Bar

In the Foundation, we implemented the player bar using jQuery by adding the HTML to the bottom of the Album view. When we converted Bloc Jams to an SPA, we removed the player bar. Now, we'll use Angular to create a player bar template and include it in the Album view. For the time being, the player bar controls will not work – we'll implement the player bar functionality using Angular in a later checkpoint.

Create a file named `player_bar.html` in the `app/templates` directory. Copy the player bar markup from `bloc-jams/album.html` and paste it into the new file:

```
~/bloc/bloc-jams-angular/app/templates/player_bar.html
```

```

+ <section class="player-bar">
+   <div class="container">
+     <div class="control-group main-controls">
+       <a class="previous">
+         <span class="ion-skip-backward"></span>
+       </a>
+       <a class="play-pause">
+         <span class="ion-play"></span>
+       </a>
+       <a class="next">
+         <span class="ion-skip-forward"></span>
+       </a>
+     </div>
+     <div class="control-group currently-playing">
+       <h2 class="song-name"></h2>
+       <h2 class="artist-song-mobile"></h2>
+       <h3 class="artist-name"></h3>
+       <div class="seek-control">
+         <div class="seek-bar">
+           <div class="fill"></div>
+           <div class="thumb"></div>
+         </div>
+         <div class="current-time"></div>
+         <div class="total-time"></div>
+       </div>
+     </div>
+     <div class="control-group volume">
+       <span class="icon ion-volume-high"></span>
+       <div class="seek-bar">
+         <div class="fill"></div>
+         <div class="thumb"></div>
+       </div>
+     </div>
+   </div>
+ </section>

```

Include the Template in the Album View

Angular has a directive for including external templates called `ngInclude`. Like an `` tag, `ngInclude` has an `src` attribute that defines the path of the asset – in this case, a template.

Use the `ngInclude` like an element tag at the bottom of the Album view template:

```
~/bloc/bloc-jams-angular/app/templates/album.html

...
<main class="album-view container narrow">
  ...
</main>

+ <ng-include src="'/templates/player_bar.html'"></ng-include>



...
```

Because the player bar now has its own template, it could be added to any view using `ngInclude`. Add the player bar template to the Landing and Collection views to try it out. After we've gotten the player bar to display on multiple views, be sure to remove it from the Landing view – we don't actually want a player bar to display there.

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details.

How would you rate this checkpoint and assignment?



5 Controllers

Controllers control the flow of data in Angular applications and help keep code modularized. Each controller is automatically paired with **a scope** (`$scope`) that Angular uses to communicate between the controller and the view, making it possible to define JavaScript code for a particular DOM element.

Git

Create a new Git feature branch for this checkpoint. See **Git Checkpoint Workflow: Before Each Checkpoint** for details.

When to Use a Controller

A controller is a JavaScript object created by a constructor function. Controllers contain the **"business logic"** that apply functions and values to the scope. When Angular instantiates a new Controller object, a child scope is created and made available as an injectable parameter to the Controller's constructor function as `$scope`.

We should only use a controller for two things:

- 1. To initialize the state of the `$scope` object.
- 2. To add behavior to the `$scope` object.

Do not use controllers to manipulate the DOM. Instead, use directives, which we'll learn more about in a later checkpoint.

Define a Controller

Within the `scripts` directory, create a `controllers` directory. Within the `controllers` directory, create a file named `LandingCtrl.js` and define a controller for the Landing view:

```
~/bloc/bloc-jams-angular/app/scripts/controllers/LandingCtrl.js

+ (function() {
+   function LandingCtrl() {
+   }
+
+   angular
+     .module('blocJams')
+     .controller('LandingCtrl', LandingCtrl);
+ })();
```

The `.controller()` method has two parameters:

- 1. The name of the controller.
- 2. A callback function **or** an array that injects dependencies, with a callback function as the last item in the array.

We've named the controller `LandingCtrl` – the first argument. Recall that object constructors are capitalized by convention to distinguish them from other functions. The second argument is the callback function that executes when the controller is initialized.

Like `.config()`, we must call `.controller()` on an Angular module. Note that the `.module()` call does not have the second argument, the array of dependencies. Because we've set the dependencies in `app.js`, we only need to retrieve the already-defined module.

We may inject as many dependencies as our controller requires. In the case of `LandingCtrl`, we've injected no dependencies. With dependencies, however, the controller definition would require an array and would look like this example shown in [the documentation](#):

```
(function() {
  function MyCtrl($scope, dep1, dep2) {
    // controller logic
  }

  angular
    .module('myApp')
    .controller('MyCtrl', [$scope, dep1, dep2, MyCtrl]);
})();
```

The last item in the array must be the callback function that executes when the controller is initialized – in this case, `MyCtrl`.

Instantiate a Controller

Angular registers a new controller object via the `ngController` directive, which attaches a controller to a DOM element. For example:

```
<body ng-controller="MyCtrl">
```

The `ngController` directive tells Angular to instantiate the controller named `MyCtrl` for the `<body>` element. This directive creates a new scope tied to the controller's `$scope` object. When the application loads, Angular will read the HTML, see the `ng-controller="MyCtrl"` attribute, and execute the callback to initialize the controller.

With UI-Router, there's another way to register a controller. We can designate a controller for a particular state by adding a `controller` property to the state configuration. Add the `LandingCtrl` to the `landing` state in `app.js`:

```
~/bloc/bloc-jams-angular/app/scripts/app.js

...
stateProvider
  .state('landing', {
    url: '/',
+    controller: 'LandingCtrl as landing',
    templateUrl: '/templates/landing.html'
  });
...
```

Link to the `LandingCtrl.js` script source in `index.html`:

```
~/bloc/bloc-jams-angular/app/index.html

...
<script src="/scripts/app.js"></script>
+ <script src="/scripts/controllers/LandingCtrl.js"></script>
...
```

Important: Before continuing, read about the different ways to [assign a controller to a template](#). For the `landing` state, we use `controller as` syntax to handle nested scopes. Read about [the benefits](#) of `controller as` syntax and examine [the example](#) provided by the Angular documentation.

Add Logic to a Controller

To initialize the `$scope` object, a controller attaches properties to it. Add a `heroTitle` property to the `LandingCtrl`:

```
~/bloc/bloc-jams-angular/app/scripts/controllers/LandingCtrl.js

...
function LandingCtrl() {
+   this.heroTitle = "Turn the Music Up!";
}
...
```

Using the `this` keyword adds `heroTitle` as a property on the `LandingCtrl`'s `$scope` object. `$scope` properties contain **the model**, or data, that the view will present, and are available to the template at the point in the DOM where the controller is registered. The `LandingCtrl` for Bloc Jams is registered for the `landing.html` template.

Within this designated template, we can use `markup` to display properties in the view. Use `Use` to display the hero title in `landing.html`:

```
~/bloc/bloc-jams-angular/app/templates/landing.html

...
<section class="hero-content">
-   <h1 class="hero-title">Turn the music up!</h1>
+   <h1 class="hero-title"></h1>
</section>
...
```

The ```` are a declarative way of specifying data binding locations in the HTML. Angular automatically updates this text whenever the `as`` syntax used when the controller was assigned to the template.

What exists between the ```` markup is called an **Angular expression**, which is similar to a JavaScript expression but differs in a few ways. Refer to the **AngularJS Developer Guide on expressions** to view their differences.

Create a Controller for the Collection View

The Bloc Jams Foundation Collection view uses a `for` loop and jQuery's `append()` to add a specified number of albums to the Collection view. We will need to refactor this because the jQuery code will break within the Angular application.

To start, copy the `fixtures.js` file created in the Foundation and move it to the Angular project in the `scripts` directory. Add the script source to `index.html`:

```
~/bloc/bloc-jams-angular/app/index.html

...
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.4/angular.min.js"></script>
<script src="/scripts/app.js"></script>
+ <script src="/scripts/fixtures.js"></script>
...
```

Next, create a `CollectionCtrl.js` file and add a controller for the Collection view:

```
~/bloc/bloc-jams-angular/app/scripts/controllers/CollectionCtrl.js
```

```
+ (function() {
+     function CollectionCtrl() {
+     }
+
+     angular
+         .module('blocJams')
+         .controller('CollectionCtrl', CollectionCtrl);
+ })();
```

Register the `CollectionCtrl` to the `collection` state in `app.js`:

```
~/bloc/bloc-jams-angular/app/scripts/app.js

...
.state('collection', {
    url: '/collection',
+   controller: 'CollectionCtrl as collection',
    templateUrl: '/templates/collection.html'
});
...
```

And link to the `CollectionCtrl.js` script source in `index.html`:

```
~/bloc/bloc-jams-angular/app/index.html

...
<script src="/scripts/app.js"></script>
<script src="/scripts/controllers/LandingCtrl.js"></script>
+ <script src="/scripts/controllers/CollectionCtrl.js"></script>
...

```

Before refactoring, look at the Bloc Jams Foundation code that runs when the Collection view loads:

```
~/bloc/bloc-jams/scripts/collection.js

...
var $collectionContainer = $('.album-covers');
$collectionContainer.empty();
for (var i = 0; i < 12; i++) {
    var $newThumbnail = buildCollectionItemTemplate();
    $collectionContainer.append($newThumbnail);
}

```

With jQuery, we select a DOM element and, with the `for` loop, append a new album cover thumbnail to the element as many times as the loop specifies.

Instead of using jQuery to append images, bind the data from the `albumPicasso` object to the Collection template:

```
~/bloc/bloc-jams-angular/app/scripts/controllers/CollectionCtrl.js

...
function CollectionCtrl() {
+   this.albums = [];
+   for (var i=0; i < 12; i++) {
+       this.albums.push(angular.copy(albumPicasso));
+   }
}
...
```

`angular.copy` is one of **several global function components** on the `angular` object.

We add an `albums` property and set its value to an empty array. Within the `for` loop, we use `angular.copy` to make copies of `albumPicasso` and push them to the array.

Access the Data in the Template

Now that we've bound the album data to the `CollectionCtrl`, we need to update the Collection template to access the necessary information and display multiple albums. Start by adding an `ngRepeat` to the template:

```
~/bloc/bloc-jams-angular/app/templates/collection.html

<section class="album-covers container clearfix">
-   <div class="collection-album-container column fourth">
+   <div class="collection-album-container column fourth" ng-repeat="album in collection">
...

```

Similar to a `for..in` statement, the `ngRepeat` directive iterates through a collection. From [the AngularJS documentation](#):

`ngRepeat` instantiates a template once per item from a collection. Each template instance gets its own scope, where the given loop variable is set to the current collection item, and `$index` is set to the item index or key.

In this case, the collection is the `albums` array that we created to hold copies of `albumPicasso`. The `"album in collection.albums"` part of the code is what allows us to access the data from a single item in the collection. Update the Collection template to access the data from the `albums` property:

```
~/bloc/bloc-jams-angular/app/templates/collection.html

<section class="album-covers container clearfix">
    <div class="collection-album-container column fourth" ng-repeat="album in collection.albums">
-       
+       <img src="">
        <div class="collection-album-info caption">
            <p>
-               <a class="album-name" ui-sref="album">The Colors</a>
+               <a class="album-name" ui-sref="album"></a>
                <br/>
-               <a ui-sref="album">Pablo Picasso</a>
+               <a ui-sref="album"></a>
                <br/>
-               X songs
+               songs
            </p>
        </div>
    </div>
...



```

With `ngRepeat`, the `<div>` with the `collection-album-container` class will repeat as many times as set by the loop.

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details.

How would you rate this checkpoint and assignment?



6 Services: Part 1

Controllers have a specific role in an application and should not share code or state between each other. Instead, Angular has **services** for that purpose.

Angular **services** are objects that can share data and behavior across several components (controllers, directives, filters, even other services) throughout an application. To use a service, we **inject it as a dependency** for the component that depends on the service.

Git

Create a new Git feature branch for this checkpoint. See **Git Checkpoint Workflow: Before Each Checkpoint** for details.

Service Recipes

Angular has five service types, or "recipes":

	Recipe	Brief Description
1.	Value Recipe	The simplest service of the recipes. It returns a value.
2.	Factory Recipe	Simple like the Value recipe, but allows us to inject a function instead of a variable.
3.	Service	Produces a service just like the Value or Factory recipes, but

3.	Recipe	does so by invoking a constructor with the <code>new</code> operator.
4.	Provider Recipe	The most verbose recipe with the most abilities. Overkill for most services.
5.	Constant Recipe	Similar to the Value recipe, but differs in that it is available in both the configuration and run phases.

For most problems we encounter, there is an optimal recipe that will help us conquer it.

Note the third item in the list that goes by the regrettable name of "Service." As the group of recipes are called "service recipes," it can be confusing that one of the individual recipes itself is called the "Service" recipe.

Angular's creators are aware of the confusion this naming may cause, and they acknowledge their mistake in a humorous way:

Yes, we have called one of our service recipes 'Service'. We regret this and know that we'll be somehow punished for our misdeed. It's like we named one of our offspring 'Child'. Boy, that would mess with the teachers.

Read the [AngularJS Developer Guide on providers](#) to learn more about the differences between the recipes.

Create a Service

Angular provides some **built-in services**, but we will often need to create custom services. We register a service in the same way we've learned to register a controller, by calling a function on the application's module.

Angular's built-in services and objects, like `$scope` and `$interval`, start with a `$` as an identifier. This is a naming convention. Avoid naming custom services with this prefix.

Within the `scripts` directory, create a `services` directory. Within the `services` directory, create a file named `Fixtures.js` and register a `Fixtures` service using the Factory recipe:

```
~/bloc/bloc-jams-angular/app/scripts/services/Fixtures.js
```

```
+ (function() {  
+     function Fixtures() {  
+         var Fixtures = {};  
+         return Fixtures;  
+     }  
+  
+     angular  
+         .module('blocJams')  
+         .factory('Fixtures', Fixtures);  
+ })();
```

This style of factory declaration is based on the [Opinionated Angular Styleguide](#)

`.factory()` designates the use of the Factory recipe. For the Service recipe, we would use `.service()`, and so on for the other types.

Within the `Fixtures` function, we declare a variable and set it to an empty object. The factory will return this object and make its properties and methods available to other parts of our Angular application.

Open the `scripts/fixtures.js` file and copy the two album objects into the `scripts/services/Fixtures.js` file:

```
~/bloc/bloc-jams-angular/app/scripts/services/Fixtures.js
```

```

...
function Fixtures() {
    var Fixtures = {};

+   var albumPicasso = {
+       title: 'The Colors',
+       artist: 'Pablo Picasso',
+       label: 'Cubism',
+       year: '1881',
+       albumArtUrl: '/assets/images/album_covers/01.png',
+       songs: [
+           { title: 'Blue', duration: '161.71', audioUrl: '/assets/music/blue' },
+           { title: 'Green', duration: '103.96', audioUrl: '/assets/music/green' },
+           { title: 'Red', duration: '268.45', audioUrl: '/assets/music/red' },
+           { title: 'Pink', duration: '153.14', audioUrl: '/assets/music/pink' },
+           { title: 'Magenta', duration: '374.22', audioUrl: '/assets/music/magenta' },
+       ]
+   };

+   var albumMarconi = {
+       title: 'The Telephone',
+       artist: 'Guglielmo Marconi',
+       label: 'EM',
+       year: '1909',
+       albumArtUrl: '/assets/images/album_covers/20.png',
+       songs: [
+           { title: 'Hello, Operator?', duration: '1:01' },
+           { title: 'Ring, ring, ring', duration: '5:01' },
+           { title: 'Fits in your pocket', duration: '3:21' },
+           { title: 'Can you hear me now?', duration: '3:14' },
+           { title: 'Wrong phone number', duration: '2:15' }
+       ]
+   };

    return Fixtures;
}
...

```

Note that a forward slash (/) has been added to the beginning of each asset URL.

We'll use this service to pull the album data into our application. Delete the `scripts/fixtures.js` file. Add a public `getAlbum` method to the service:

~/bloc/bloc-jams-angular/app/scripts/services/Fixtures.js

```
...  
function Fixtures() {  
    var Fixtures = {};  
  
    var albumPicasso = { ... };  
  
    var albumMarconi = { ... };  
  
+   Fixtures.getAlbum = function() {  
+       return albumPicasso;  
+   };  
  
    return Fixtures;  
}  
...
```

This service is a "Plain Old JavaScript Object" (POJO). Components that inject this service as a dependency can access the public methods of the object – that is, the properties and methods that are `return`ed.

In `index.html`, remove the source link to `scripts/fixtures.js` and add one for the `Fixtures` service:

~/bloc/bloc-jams-angular/app/index.html

```
...  
- <script src="/scripts/fixtures.js"></script>  
+ <script src="/scripts/services/Fixtures.js"></script>  
...
```

Inject a Service

Inject the custom service into the `AlbumCtrl`:

~/bloc/bloc-jams-angular/app/scripts/controllers/AlbumCtrl.js

```

(function() {
-   function AlbumCtrl() {
+   function AlbumCtrl(Fixtures) {
        this.albumData = angular.copy(albumPicasso);
    }

    angular
        .module('blocJams')
-       .controller('AlbumCtrl', AlbumCtrl);
+       .controller('AlbumCtrl', ['Fixtures', AlbumCtrl]);
    })();

```

We add `Fixtures` to `AlbumCtrl`'s array of dependencies. Once injected, the service is available for use within the controller.

Update `AlbumCtrl` to use the `Fixtures` service's `getAlbum()` method:

~/bloc/bloc-jams-angular/app/scripts/controllers/AlbumCtrl.js

```

...
function AlbumCtrl(Fixtures) {
-   this.albumData = angular.copy(albumPicasso);
+   this.albumData = Fixtures.getAlbum();
}
...

```

`AlbumCtrl` uses `Fixtures`'s `getAlbum()` method to get the `albumPicasso` object.

Update `CollectionCtrl`

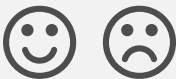
At this point you will see an error in the console if you were to visit `localhost:3000/collection` `albumPicasso` would be undefined. We haven't injected the `Fixture` service into `CollectionCtrl` yet. This is something you will do on your own in the assignment section.

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each](#)

Checkpoint for details.

How would you rate this checkpoint and assignment?



6. Services: Part 1

 Assignment

 Discussion

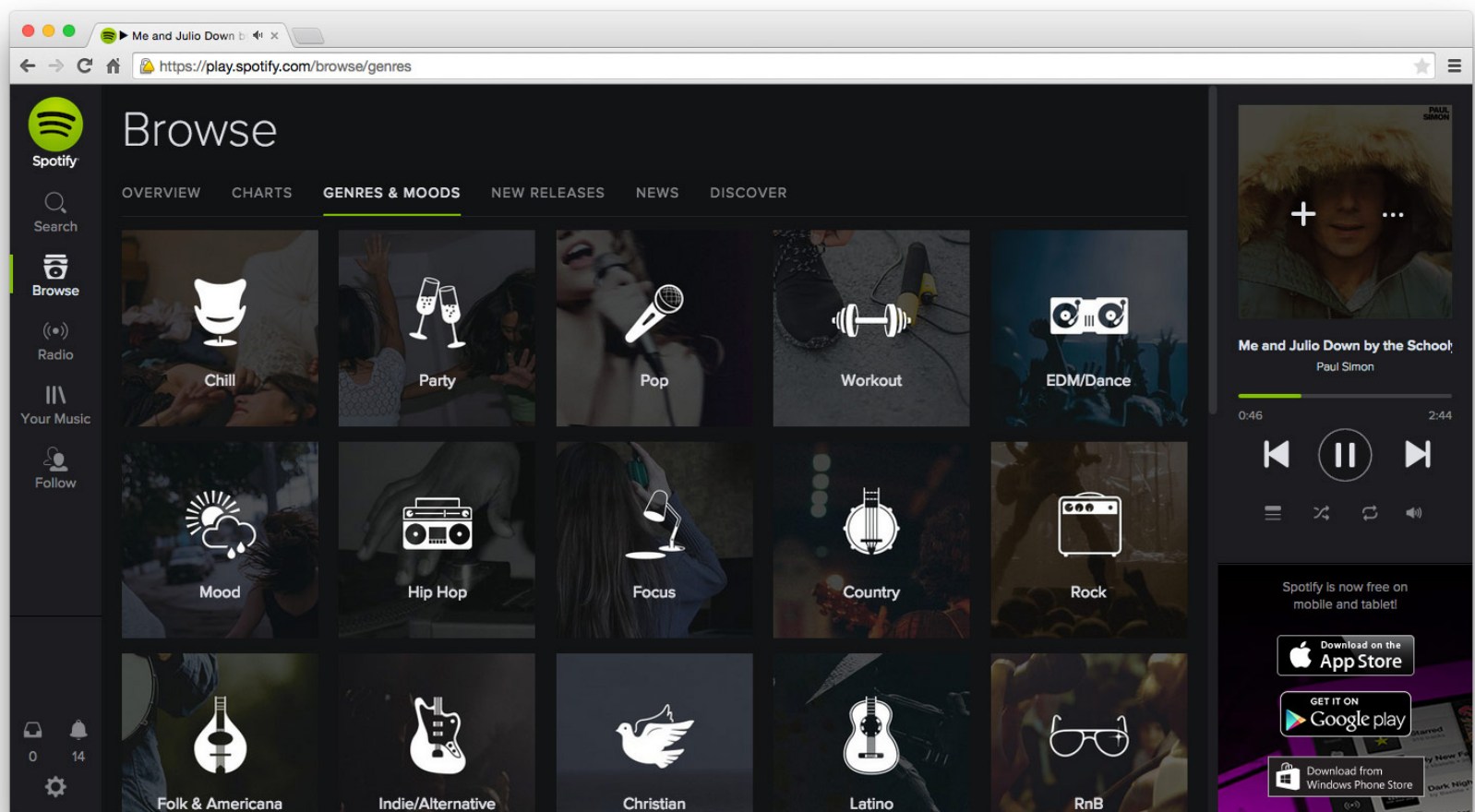
 Submission



7 Services: Part 2

An Angular service is a **singleton**, meaning regardless of how many components depend on the same service, each component gets the same instance of the service. If data in the service changes, any place where that service is injected will reflect the update(s) because the data references a *single* object (not multiple instances of an object).

Think of applications like Spotify, where the song we're listening to remains intact even as we navigate to different views. We view a playlist, then our profile, and then search results, and all the while the same song continues to play without needing to stop or reload.



If we were to build Spotify with Angular, we might create three separate controllers to govern the views described above (playlist, profile, and search results). These controllers could all have a service named `SongPlayer` injected as a dependency.

This service might hold data such as if a song is playing, and if so, the song's name, artist, album, length, and timestamp. It may also control behaviors such as pause, play, previous, next, and volume.

Since a service shares consistent data and behavior across components, Angular limits each service to a single instance: a singleton.

The example above is but one small use case for a service. Often, services are used to interact with application program interfaces, or APIs.

Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

Create a `SongPlayer` Service for Bloc Jams

Within the `services` directory, create a file named `SongPlayer.js` and register a `SongPlayer` service using the Factory recipe:

```
~/bloc/bloc-jams-angular/app/scripts/services/SongPlayer.js
```

```
+ (function() {  
+     function SongPlayer() {  
+         var SongPlayer = {};  
+         return SongPlayer;  
+     }  
+  
+     angular  
+         .module('blocJams')  
+         .factory('SongPlayer', SongPlayer);  
+ })();
```

Like the `Fixtures` service, within the `SongPlayer` service we create a variable and set it to an empty object. The service returns this object, making its properties and methods public to the rest of the application.

Add the source link for the `SongPlayer` service to `index.html`:

```
~/bloc/bloc-jams-angular/app/index.html
```

```
...
<script src="/scripts/services/Fixtures.js"></script>
<script src="/scripts/services/SongPlayer.js"></script>
...
```

Since the `SongPlayer` service will play music, we need to add the Buzz library to our application. Add the Buzz library source to `index.html`:

```
~/bloc/bloc-jams-angular/app/index.html
```

```
...
<script src="https://cdnjs.cloudflare.com/ajax/libs/angular-ui-router/0.4.2/angular-ui-router.min.js"></script>
+ <script src="https://cdnjs.cloudflare.com/ajax/libs/buzz/1.2.0/buzz.min.js"></script>
...
```

Inject the `SongPlayer` Service

To use the service, we need to decide where to inject it as a dependency. We'll play music from the Album view, so inject the service into `AlbumCtrl`:

```
~/bloc/bloc-jams-angular/app/scripts/controllers/AlbumCtrl.js
```

```
(function() {
-   function AlbumCtrl(Fixtures) {
+   function AlbumCtrl(Fixtures, SongPlayer) {
        this.albumData = Fixtures.getAlbum();
+       this.songPlayer = SongPlayer;
    }

    angular
        .module('blocJams')
-       .controller('AlbumCtrl', ['Fixtures', AlbumCtrl]);
+       .controller('AlbumCtrl', ['Fixtures', 'SongPlayer', AlbumCtrl]);
})();
```

The `songPlayer` property holds the service and makes the service accessible within the Album view.

Add a `play` Method

With the current state of the Album view, we can view a play button when we hover over a song row. Let's add a `play` method to the `SongPlayer` service so that we can play a song:

~/bloc/bloc-jams-angular/app/scripts/services/SongPlayer.js

```
...
function SongPlayer() {
  var SongPlayer = {};

+   SongPlayer.play = function(song) {
+     var currentBuzzObject = new buzz.sound(song.audioUrl, {
+       formats: ['mp3'],
+       preload: true
+     });
+
+     currentBuzzObject.play();
+   };

  return SongPlayer;
}
...
```

The `play` method takes an argument, `song`, which we'll get from the Album view when a user clicks the play button; the `ngRepeat` directive used in the Album view template will dictate which `song` to pass into the function. The `play` method creates a new Buzz object using the `song`'s `audioUrl` property and then calls Buzz's own `play` method on the object.

To trigger the `play` method, however, we need to add an `ngClick` directive to the play button anchor tag in `album.html`:

~/bloc/bloc-jams-angular/app/templates/album.html

```
...
- <a class="album-song-button" ng-show="hovered && !playing"><span class="ion-play"></span></a>
+ <a class="album-song-button" ng-show="hovered && !playing" ng-click="album.songPlayer.play(song)"><span class="ion-play"></span></a>
...
```

`album.songPlayer.play(song)` may seem verbose, but it's not. Let's break it down:

Object/Property	Description
<code>album</code>	Refers to the controller. We use "controller as" syntax: <code>AlbumCtrl as album</code> in our <code>config</code> block in <code>app.js</code> .
<code>.songPlayer</code>	A property on the <code>album</code> object: <code>this.songPlayer = SongPlayer;</code> , where <code>this</code> refers to the controller.
<code>.play(song)</code>	A method returned by the <code>SongPlayer</code> service, which we've injected and made available to <code>AlbumCtrl</code> .

View Bloc Jams locally and click to play a song. We can't pause a song yet, so refresh the page to stop the music.

Review `album.js`

Before we implement cohesive play and pause methods, let's review some code we implemented in `album.js` in the Foundation.

The `createSongRow` function handles a number of things. It:

1. Generates **the song row template**.
2. Declares and calls the `clickHandler` **function**, which plays or pauses a song.
3. Declares and calls the `onHover` **and** `offHover` **functions**, which shows either a play button or the song number.

We no longer need a `createSongRow` function because our Angular app handles these functions in other ways. For the song row template, we now declare it in the view. We've also replaced the `onHover` and `offHover` functions with `ngMouseover` and `ngMouseleave` directives in the Album view.

Now, we're working on playing and pausing music using a `SongPlayer` service instead of relying on a single `clickHandler` function. We'll use the `ngClick` directive in the view to tell Angular what to do when a user interacts with certain elements, such as play and pause buttons.

Note that `clickHandler` calls numerous other functions, such as `setSong`, `updateSeekBarWhileSongPlays`, and `updatePlayerBarSong`. When we dive into those functions, we see that they also call other functions. There are many pieces to making a seemingly simple music player work well, and we'll work our way through them in due time.

We'll start with playing and pausing music.

Refactor the `play` Method

With our current `play` method, we could play all the songs on the album simultaneously, but that's not the expected behavior of a music player. We need better logic if we want users to have a good experience.

We **already figured out the logic** in the foundation, but now we need to reimplement it using Angular. For example, the `clickHandler` function considers what to do if the currently playing song is or is not the same as the song the user clicks on.

Update the `play` method with a condition that checks if the currently playing song is not equal to the song the user clicks on:

```
~/bloc/bloc-jams-angular/app/scripts/services/SongPlayer.js
```

```

...
function SongPlayer() {
    var SongPlayer = {};

+   var currentSong = null;
+   var currentBuzzObject = null;

    SongPlayer.play = function(song) {
+       if (currentSong !== song) {
+           if (currentBuzzObject) {
+               currentBuzzObject.stop();
+           }

-           var currentBuzzObject = new buzz.sound(song.audioUrl, {
+           currentBuzzObject = new buzz.sound(song.audioUrl, {
                formats: ['mp3'],
                preload: true
            });

+           currentSong = song;

            currentBuzzObject.play();
+       }
    };

    return SongPlayer;
}
...

```

First, we declare new variables named `currentSong` and `currentBuzzObject` and set their values to `null`, which is **what we did in the foundation**. We've removed the `currentBuzzObject` variable declaration from the local scope of the `play` method because we anticipate needing to access this variable elsewhere in the service.

If the currently playing song is not the same as the song the user clicks on, then we want to:

1. Stop the currently playing song, if there is one.
2. Set a new Buzz sound object.
3. Set the newly chosen song object as the `currentSong`.
4. Play the new Buzz sound object.

Next, add a second conditional statement that checks if `currentSong` is equal to `song`:

~/bloc/bloc-jams-angular/app/scripts/services/SongPlayer.js

```
...
SongPlayer.play = function(song) {
  if (currentSong !== song) {
    ...
-   }
+   } else if (currentSong === song) {
+     if (currentBuzzObject.isPaused()) {
+       currentBuzzObject.play();
+     }
+   }
};
...
```

If the user can trigger the `play` method on a song that is already set as the `currentSong`, then the assumption is that the song must be paused. The conditional statement `if (currentBuzzObject.isPaused())` is a check to make sure our assumption is correct.

Revisit Wishful Coding

Before we can implement a `pause` method, we need to go back to the "wishful coding" we wrote in `album.html` earlier in the project, specifically the expressions for the `ngShow` directive:

~/bloc/bloc-jams-angular/app/templates/album.html

```
<td class="song-item-number">
  <span ng-show="!hovered && !playing">1</span>
  <a class="album-song-button" ng-show="hovered && !playing" ng-click="album.songPla
  <a class="album-song-button" ng-show="playing"><span class="ion-pause"></span></a>
</td>
```

We used a variable named `playing` that we hadn't yet implemented, but was intended to reflect whether or not a song is playing. We'll maintain this idea, though we'll set `playing` to be a property on the `song` object and track the state of the song that way. **Replace the three instances of `playing` in `album.html` (as shown above) with `song.playing`.**

In our `SongPlayer` service, every time we play, pause, or stop a song, we'll need to update this boolean. Add the following to the `play` method:

~/bloc/bloc-jams-angular/app/scripts/services/SongPlayer.js

```
...
SongPlayer.play = function(song) {
  if (currentSong !== song) {
    if (currentBuzzObject) {
      currentBuzzObject.stop();
+     currentSong.playing = null;
    }

    currentBuzzObject = new buzz.sound(song.audioUrl, {
      formats: ['mp3'],
      preload: true
    });

    currentSong = song;

    currentBuzzObject.play();
+    song.playing = true;
  }
  ...
}
```

When we click to play a song, we should now see the pause button.

Write a `pause` Method

Now that we can actually see the pause button, let's implement the method to call when a user clicks on it:

~/bloc/bloc-jams-angular/app/scripts/services/SongPlayer.js

```

...
SongPlayer.play = function(song) {
    ...
};

+ SongPlayer.pause = function(song) {
+     currentBuzzObject.pause();
+     song.playing = false;
+ };
...

```

The `pause` method requires less logic because we don't need to check for various conditions – a song must already be playing before the user can trigger it.

Add an `ngClick` directive to the pause button anchor tag in `album.html`:

```
~/bloc/bloc-jams-angular/app/templates/album.html
```

```

...
- <a class="album-song-button" ng-show="song.playing"><span class="ion-pause"></span></a>
+ <a class="album-song-button" ng-show="song.playing" ng-click="album.songPlayer.pause"></a>
...

```

Open Bloc Jams and test the play and pause buttons.

Refactor

We can play and pause songs without issue now, though we should refactor some parts of our code to make it reusable. For example, **review the** `setSong` **function** from the foundation. Now look at our new `play` method.

We'll extract parts of the code from the `play` method and create a `setSong` function to handle them:

```
~/bloc/bloc-jams-angular/app/scripts/services/SongPlayer.js
```

```

...
+ var setSong = function(song) {
+   if (currentBuzzObject) {
+     currentBuzzObject.stop();
+     currentSong.playing = null;
+   }
+
+   currentBuzzObject = new buzz.sound(song.audioUrl, {
+     formats: ['mp3'],
+     preload: true
+   });
+
+   currentSong = song;
+ };

SongPlayer.play = function(song) {
  if (currentSong !== song) {
-     if (currentBuzzObject) {
-       currentBuzzObject.stop();
-       currentSong.playing = null;
-     }
-
-     currentBuzzObject = new buzz.sound(song.audioUrl, {
-       formats: ['mp3'],
-       preload: true
-     });
-
-     currentSong = song;
+     setSong(song);
    currentBuzzObject.play();
    song.playing = true;
  } else if (currentSong === song) {
    if (currentBuzzObject.isPaused()) {
      currentBuzzObject.play();
    }
  }
};
...

```

Documentation

Our `SongPlayer` service should now contain:

- two private attributes: `currentSong` and `currentBuzzObject`,
- one private function: `setSong`,
- and two public methods: `SongPlayer.play` and `SongPlayer.pause`.

As the logic of the service grows, it's important to write good documentation for our own benefit as well as the benefit of other developers.

We'll continue to group our service logic into four groups and maintain them in this order:

1. private attributes
2. private functions
3. public attributes
4. public methods

We will also provide more details for each attribute or function. For example, add the following documentation for the `setSong` function:

~/bloc/bloc-jams-angular/app/scripts/services/SongPlayer.js

```
...
+ /**
+  * @function setSong
+  * @desc Stops currently playing song and loads new audio file as currentBuzzObject
+  * @param {Object} song
+  */
  var setSong = function(song) {
    ...
  };
  ...
```

We place the documentation immediately before the function and provide:

- `@function`: Name of the function
- `@desc`: A short description
- `@param`: A list of parameters and their type

If a function does not have a parameter, then we exclude that line.

If a function returns something, such as a number, then we would include a fourth line:

```
* @returns {Number}
```

For attributes, we include two lines of information:

- `@desc`: A short description
- `@type`: The type, such as `{Object}`, `{Array}`, `{Number}`, etc.

Add the following documentation for `currentBuzzObject`:

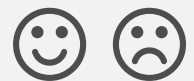
```
~/bloc/bloc-jams-angular/app/scripts/services/SongPlayer.js
```

```
...  
+ /**  
+ * @desc Buzz object audio file  
+ * @type {Object}  
+ */  
var currentBuzzObject = null;  
...
```

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details.

How would you rate this checkpoint and assignment?



7. Services: Part 2

 **Assignment**

 Discussion

 Submission



8 Services: Part 3

We can play and pause songs from the song rows in the Album view, but what about the player bar? The player bar is a separate entity from the Album view, so we'll create a controller for it.

Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

Create a Controller for the Player Bar

Create a new file for the `PlayerBarCtrl` and inject both the `Fixtures` and `SongPlayer` services into the controller:

```
~/bloc/bloc-jams-angular/app/scripts/controllers/PlayerBarCtrl.js
```

```
+ (function() {  
+   function PlayerBarCtrl(Fixtures, SongPlayer) {  
+     this.albumData = Fixtures.getAlbum();  
+     this.songPlayer = SongPlayer;  
+   }  
+  
+   angular  
+     .module('blocJams')  
+     .controller('PlayerBarCtrl', ['Fixtures', 'SongPlayer', PlayerBarCtrl]);  
+ })();
```

Add the source file to `index.html`:

```
~/bloc/bloc-jams-angular/app/index.html
```

```
...
<script src="/scripts/controllers/AlbumCtrl.js"></script>
+ <script src="/scripts/controllers/PlayerBarCtrl.js"></script>
...
```

Lastly, register `PlayerBarCtrl` for the player bar template:

```
~/bloc/bloc-jams-angular/app/templates/player_bar.html
```

```
- <section class="player-bar">
+ <section class="player-bar" ng-controller="PlayerBarCtrl as playerBar">
...
```

We use "controller as" syntax to stay consistent with our other controllers.

Update the Template

We'll also need to update the template to clearly declare the play and pause buttons of the player bar in the view:

```
~/bloc/bloc-jams-angular/app/templates/player_bar.html
```

```
...
<a class="play-pause">
-   <span class="ion-play"></span>
+   <span class="ion-play"
+       ng-show="!song.playing"
+       ng-click="playerBar.songPlayer.play(song)">
+   </span>
+   <span class="ion-pause"
+       ng-show="song.playing"
+       ng-click="playerBar.songPlayer.pause(song)">
+   </span>
</a>
...
```

We no longer use jQuery to determine which element is visible. Instead, we include both buttons in the view and use the `ngShow` directive to tell Angular when one button or the other should display. We also include `ngClick` to trigger the play and pause functions.

When we test this, however, we receive an error in the console. From this error we learn that `song` is undefined within `PlayerBarCtrl`. The scope of `PlayerBarCtrl` is different from `AlbumCtrl` and does not have access to the `song` object.

Make `currentSong` Public

The player bar is different from the Album view in that it doesn't need to know the state of individual songs because it will only be able to affect one song at a time: the currently playing song. To access the information of the currently playing song in the player bar, we'll need to do two things.

The first is to change the private attribute `currentSong` into a public attribute named `SongPlayer.currentSong` so that we can use it within the player bar:

~/bloc/bloc-jams-angular/app/scripts/services/SongPlayer.js

```
...
/**
 * @desc Active song object from list of songs
 * @type {Object}
 */
- var currentSong = null;
+ SongPlayer.currentSong = null;
...
```

Now that the attribute is public, move it below `setSong` and above `SongPlayer.play` to maintain organization of private and public attributes/functions.

Update all instances of `currentSong` to `SongPlayer.currentSong`. (Find and replace is the quickest way to accomplish this task.)

In the player bar template, update the `ngShow` expression to reflect this change:

~/bloc/bloc-jams-angular/app/templates/player_bar.html


```

...
<a class="play-pause">
  <span class="ion-play"
-      ng-show="!song.playing"
+      ng-show="!playerBar.songPlayer.currentSong.playing"
      ng-click="playerBar.songPlayer.play(song)">
  </span>
  <span class="ion-pause"
-      ng-show="song.playing"
+      ng-show="playerBar.songPlayer.currentSong.playing"
      ng-click="playerBar.songPlayer.pause(song)">
  </span>
</a>
...

```

If we play and pause a song from the song row, we see that the play and pause buttons in the player bar display correctly. When we try to pause a song from the player bar, however, we get an error in the console. The error tells us the same thing as the previous error message – `song` is undefined within `PlayerBarCtrl`. We cannot pass `song` as an argument into the `play` and `pause` methods from the player bar.

Update the `play` and `pause` Methods

While we still can't access the `song` object, that's okay. We've already figured out that we don't need to have access to `song` in the player bar. We only need to know the currently playing song, which we can access via the service. The second step to make the player bar work is to update `play` and `pause` to account for the fact that the player bar can't pass `song` as an argument.

Add the following lines to the `play` and `pause` methods:

```
~/bloc/bloc-jams-angular/app/scripts/services/SongPlayer.js
```

```

/**
 * @function play
 * @desc Play current or new song
 * @param {Object} song
 */
SongPlayer.play = function(song) {
+   song = song || SongPlayer.currentSong;
   if (SongPlayer.currentSong !== song) {
       setSong(song);
       playSong(song);
   } else if (SongPlayer.currentSong === song) {
       if (currentBuzzObject.isPaused()) {
           playSong(song);
       }
   }
};

/**
 * @function pause
 * @desc Pause current song
 * @param {Object} song
 */
SongPlayer.pause = function(song) {
+   song = song || SongPlayer.currentSong;
   currentBuzzObject.pause();
   song.playing = false;
};

```

We use `||` to tell the function: assign (1) the value of `song` or (2) the value of `SongPlayer.currentSong` to the `song` variable. The first condition occurs when we call the methods from the Album view's song rows, and the second condition occurs when we call the methods from the player bar.

Lastly, update the player bar template to remove `song` as an argument:

```
~/bloc/bloc-jams-angular/app/templates/player_bar.html
```

```

...
<a class="play-pause">
  <span class="ion-play"
    ng-show="!playerBar.songPlayer.currentSong.playing"
-    ng-click="playerBar.songPlayer.play(song)">
+    ng-click="playerBar.songPlayer.play()">
  </span>
  <span class="ion-pause"
    ng-show="playerBar.songPlayer.currentSong.playing"
-    ng-click="playerBar.songPlayer.pause(song)">
+    ng-click="playerBar.songPlayer.pause()">
  </span>
</a>
...

```

Test the ability to play and pause from the player bar – it should work now!

Next and Previous Buttons

Being able to play and pause from the player bar is one victory; now we'll work on implementing the ability to move between songs with the next and previous buttons.

Let's first review the `nextSong` **and** `previousSong` **functions** from the Foundation.

There's a lot going on in these functions. We no longer need to get the song number cell and we won't need to update the DOM in the JavaScript. We can trim the excess from these old functions and implement them anew in the `SongPlayer` service.

Access `songs` from the Current Album

To move between songs, we need to know the index of the `song` object within the `songs` array. To access the `songs` array, we need to store the album information, which we **also did in the Foundation**.

Inject the `Fixtures` service into the `SongPlayer` service. Then use the `getAlbum` method to store the album information:

```
~/bloc/bloc-jams-angular/app/scripts/services/SongPlayer.js
```

```

...
- function SongPlayer() {
+ function SongPlayer(Fixtures) {
    var SongPlayer = {};

+   var currentAlbum = Fixtures.getAlbum();
...

```

Remember to include the Fixtures factory at the bottom of the file as well:

~/bloc/bloc-jams-angular/app/scripts/services/SongPlayer.js

```

...
angular
  .module('blocJams')
  .factory('SongPlayer', ['Fixtures', SongPlayer]);
...

```

Remember to write documentation for this private attribute.

Now that we can access the album, we can write a function to get the index of a song. Write a `getSongIndex` function, which is similar to the `trackIndex` **function** we wrote in the foundation:

~/bloc/bloc-jams-angular/app/scripts/services/SongPlayer.js

```

...
+ var getSongIndex = function(song) {
+   return currentAlbum.songs.indexOf(song);
+ };

/**
 * @desc Active song object from list of songs
 * @type {Object}
 */
SongPlayer.currentSong = null;
...

```

Add documentation for this private function.

Write a `previous` Method

Armed with the ability to get a song's index, we'll write a method to go to the previous song:

```
~/bloc/bloc-jams-angular/app/scripts/services/SongPlayer.js
```

```
...
SongPlayer.pause = function(song) {
    ...
};

+ SongPlayer.previous = function() {
+     var currentSongIndex = getSongIndex(SongPlayer.currentSong);
+     currentSongIndex--;
+ };
...
```

Don't forget the documentation! This will be the last reminder to add documentation for newly written attributes and functions/methods. Make it a habit.

We use the `getSongIndex` function to get the index of the currently playing song and then decrease that index by one.

Next, we'll add logic for what should happen if the previous song index is less than zero – that is, what should happen when the user is on the first song and clicks the previous button? There are many possibilities. We'll opt to:

- stop the currently playing song, and
- set the value of the currently playing song to the first song.

Update the `previous` method with the following conditional statement:

```
~/bloc/bloc-jams-angular/app/scripts/services/SongPlayer.js
```

```

...
SongPlayer.previous = function() {
    var currentSongIndex = getSongIndex(SongPlayer.currentSong);
    currentSongIndex--;

+   if (currentSongIndex < 0) {
+       currentBuzzObject.stop();
+       SongPlayer.currentSong.playing = null;
+   }
};
...

```

If the `currentSongIndex` is not less than zero, then it must be greater than zero. Add an `else` conditional that moves to the previous song and automatically plays it:

~/bloc/bloc-jams-angular/app/scripts/services/SongPlayer.js

```

...
SongPlayer.previous = function() {
    var currentSongIndex = getSongIndex(SongPlayer.currentSong);
    currentSongIndex--;

    if (currentSongIndex < 0) {
        currentBuzzObject.stop();
        SongPlayer.currentSong.playing = null;
-   }
+   } else {
+       var song = currentAlbum.songs[currentSongIndex];
+       setSong(song);
+       playSong(song);
+   }
};
...

```

To trigger the `previous` method, add an `ngClick` directive to the previous button anchor tag in `player_bar.html`:

~/bloc/bloc-jams-angular/app/templates/player_bar.html

```
...
- <a class="previous">
+ <a class="previous" ng-click="playerBar.songPlayer.previous()">
...
```

Test the previous button in the view and revel in another victory.

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details.

How would you rate this checkpoint and assignment?



8. Services: Part 3

 **Assignment**

 Discussion

 Submission



9 Directives: Part 1

Any Angular element in the HTML is a **directive**. A **directive** binds Angular functionality to HTML on a page. We've already used several of Angular's built-in directives to refactor Bloc Jams:

Directive	Description
<code>ngApp</code>	Designates the root element of the application
<code>ngController</code>	Attaches a controller to the view
<code>ngRepeat</code>	Instantiates a template once per item from a collection
<code>ngClick</code>	Allows us to specify custom behavior when an element is clicked
<code>ngShow</code>	Shows or hides the given HTML element based on the expression provided to the attribute

We've also used some of UI-Router's built-in directives:

Directive	Description
<code>ui-view</code>	Tells <code>\$state</code> where to place templates
<code>ui-sref</code>	Binds an <code><a></code> tag to a state

The `ng` prefix on directives is a naming convention reserved for the Angular core library. UI-Router, which is not part of the Angular core library, prefixes its directives with `ui`. When naming your own directives, you should choose a custom prefix. This will avoid potential bugs and naming conflicts that may occur with future releases of Angular.

When we declare `ng-controller="SomeCtrl"`, we're using the `ngController` directive to run certain Angular functionality. In this case, `ngController` searches for a controller named "SomeCtrl" and executes its code. Angular executes the code within the scope of the element attached to `ng-controller`.

In this example, as with every directive we have seen so far, we invoke directives as attributes in the HTML. When Angular compiles an application's HTML, however, it can match directives based on more than just attributes. It can also match directives on class names, element names, and even comments.

Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

Directive Types

Let's say we have a custom directive named `myDirective`. Here are the four ways in which we could invoke that directive in the HTML:

Attribute Directive

```
<div my-directive="expression"></div>
```

This is the directive type with which we are already familiar. The directive evaluates the `expression`.

Class Directive

```
<div class="my-directive: expression;"></div>
```

With the class directive, the `my-directive` class is applied to the element if the `expression` is true.

Element Directive

```
<my-directive></my-directive>
```

With the element directive, the directive name becomes a custom tag name.

Comment Directive

```
<!-- directive: my-directive expression -->
```

Although comment directives are an option, **Angular documentation recommends developers use element and attribute directives above other options**. Element and attribute directives often make it easier for us, the developer, to match elements to directives.

A Note on Name Normalization

When we talk about a directive we refer to it by its `camelCase` name, but when we use it in the HTML the name is `dash-delimited`.

The case-sensitive `camelCase` name is considered the *normalized* name. Angular uses the normalized name (e.g. `ngApp`) to match elements to directives. HTML, however, is not case sensitive, so directives are referred to by their lower-case forms (e.g. `ng-app`).

When Angular compiles HTML, it converts the `dash-delimited` name to `camelCase`.

Create a Custom Directive

Like controllers and services, directives are registered on modules. We'll create a directive to handle the seek bars for Bloc Jams.

Within the `scripts` directory, create a `directives` directory. Within the `directives` directory, create a file named `seekBar.js` and register a `seekBar` directive:

```
~/bloc/bloc-jams-angular/app/scripts/directives/seekBar.js
```

```
+ (function() {
+     function seekBar() {
+     }
+
+     angular
+         .module('blocJams')
+         .directive('seekBar', seekBar);
+ })();
```

For directives, the callback function (in this case, `seekBar`) is a *factory function*. It returns an object that describes the directive's behavior to Angular's HTML compiler. This object communicates the behavior through options. Add the following to the directive:

~/bloc/bloc-jams-angular/app/scripts/directives/seekBar.js

```
...
function seekBar() {
+     return {
+         templateUrl: '/templates/directives/seek_bar.html',
+         replace: true,
+         restrict: 'E'
+     };
+ }
...
```

We return three options: `templateUrl`, `replace`, and `restrict`.

Directive Option	Description
<code>templateUrl</code>	Specifies a URL from which the directive will load a template.
<code>replace</code>	Specifies what the template should replace. If <code>true</code> , the template replaces the directive's element. If <code>false</code> , the template replaces the <i>contents</i> of the directive's element.
<code>restrict</code>	Restricts the directive to a specific declaration style: element <code>E</code> , attribute <code>A</code> , class <code>C</code> , and comment <code>M</code> . If omitted, the defaults (<code>E</code> and <code>A</code>) are used. Multiple restrictions are stringed together, for example <code>AE</code> or <code>AEC</code> .

Directives can have **many more options** than the ones described above.

We've named the directive `seekBar`, which means Angular will look for `seek-bar` in the HTML and call this directive when it finds that markup.

`restrict: 'E'` instructs Angular to treat this directive as an element. For example, Angular will run the code if it finds `<seek-bar>` in the HTML, but not if it finds `<div seek-bar>`.

`replace: true` instructs Angular to completely replace the `<seek-bar>` element with the directive's HTML template rather than insert the HTML *between* the `<seek-bar></seek-bar>` tags.

The `templateUrl` option specifies the path to the HTML template that the directive will use. We've added a template URL, but we haven't yet made the template. Let's do that now.

Create a Directive Template

In the `templates` directory, create a `directives` directory to distinguish primary application templates from directive templates. Within the `directives` directory, create a file named `seek_bar.html`.

Find the seek bar element in `player_bar.html` and paste it into the newly created `seek_bar.html` file:

```
~/bloc/bloc-jams-angular/app/templates/directives/seek_bar.html
```

```
+ <div class="seek-bar">
+   <div class="fill"></div>
+   <div class="thumb"></div>
+ </div>
```

In `player_bar.html`, find **both** instances of the seek bar element and replace them with our custom `seekBar` directive:

```
~/bloc/bloc-jams-angular/app/templates/player_bar.html
```

```
...
- <div class="seek-bar">
-   <div class="fill"></div>
-   <div class="thumb"></div>
- </div>
+ <seek-bar></seek-bar>
...
```

When Angular traverses the HTML, it replaces the `<seek-bar>` element with its corresponding directive definition, which includes both the HTML template and any functionality associated with it.

To display the directive template in the view, add the `seekBar.js` source to `index.html`:

```
~/bloc/bloc-jams-angular/app/index.html
```

```
...
<script src="/scripts/controllers/PlayerBarCtrl.js"></script>
+ <script src="/scripts/directives/seekBar.js"></script>
...
```

The seek bars should now display in the view.

Link Directive Logic to the DOM

We'll add two more options to the directive:

```
~/bloc/bloc-jams-angular/app/scripts/directives/seekBar.js
```

```

...
function seekBar() {
    return {
        templateUrl: '/templates/directives/seek_bar.html',
        replace: true,
-       restrict: 'E'
+       restrict: 'E',
+       scope: { },
+       link: function(scope, element, attributes) {
            // directive logic to return
+       }
    };
}
...

```

We've added `scope` and `link` options:

Directive Option	Description
<code>scope</code>	Specifies that a new scope be created for the directive.
<code>link</code>	Responsible for registering DOM listeners and updating the DOM. This is where we put most of the directive logic.

Declaring an empty `scope` property ensures that a new scope will exist solely for the directive (referred to as **isolate-scope**). An isolate-scope allows us to bind functions from the directive's view to its scope.

The `link` function is automatically generated and scoped to the element defining the directive. Think of it as a function that executes when the directive is instantiated in the view. This is where all logic related to DOM manipulation will go.

Directive `link` functions take the same arguments (with a strict order) during declaration. Altering the order of these arguments will cause errors.

1. The `link` method's first argument is its `scope` object. Attributes and methods on the `scope` object are accessible within the directive's view.
2. The second argument is the **jQuery-wrapped element** that the directive matches.

3. The third argument is a hash of attributes with which the directive was declared. If we declare `<seek-bar>` with no attributes in the HTML, then this hash will be empty.

jQuery and Angular

Angular and jQuery are both powerful tools written with different programming design paradigms. The Angular community generally prefers to keep jQuery logic out of Angular apps because we can replicate jQuery functionality with Angular and plain JavaScript. Including both means we will have two heavy source libraries in our application when it may not be necessary.

If we are going to use jQuery logic anywhere in an Angular app, the *only* place where we would use it is in a directive's `link` function. Angular has a subset of some basic jQuery DOM functions called **jqLite**. It allows Angular users to get some of the most common DOM manipulation and event handling methods from jQuery without having to include the entire library.

View the [angular.element](#) **documentation** to view the jQuery methods that are available with jqLite.

For the Bloc Jams project, we will use more jQuery methods than are available through jqLite, so we'll need to add the jQuery library to `index.html`:

```
~/bloc/bloc-jams-angular/app/index.html
```

```
...  
+ <script src="https://code.jquery.com/jquery-3.2.1.min.js"></script>  
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.4/angular.min.js"></script>  
...
```

Add Logic to the `seekBar` Directive

Before we write logic for the directive, let's review the `setupSeekBars` **function** that we wrote in the Foundation. This function does a few things:

1. Identifies the seek bar elements.
2. Has a click event handler that, depending on which seek bar the user clicks, either jumps to a new point in the currently playing song or sets a new volume level.
3. Has a mousedown event handler that tracks if a user has clicked on the seek bar

thumb and is dragging it to a new position. If so, like the click event, either the song position or the volume changes.

We'll also review the `updateSeekPercentage` **function** because we need to determine the position of the thumb as well as the width of the seek bar playback that allows a user to see how much of a song has played.

Let's rewrite this logic in the `link` function:

~/bloc/bloc-jams-angular/app/scripts/directives/seekBar.js

```
...
function seekBar() {
  return {
    templateUrl: '/templates/directives/seek_bar.html',
    replace: true,
    restrict: 'E',
    scope: { },
    link: function(scope, element, attributes) {
+       scope.value = 0;
+       scope.max = 100;
+
+       var percentString = function () {
+         var value = scope.value;
+         var max = scope.max;
+         var percent = value / max * 100;
+         return percent + "%";
+       };
+
+       scope.fillStyle = function() {
+         return {width: percentString()};
+       };
    }
  };
}
...
```

`seekBar`'s HTML template can access the attributes and methods of the directive's `scope` object – in this case: `scope.value`, `scope.max`, and `scope.fillStyle`.

Attribute / Method	Description
--------------------	-------------

<code>scope.value</code>	Holds the value of the seek bar, such as the currently playing song time or the current volume. Default value is 0.
<code>scope.max</code>	Holds the maximum value of the song and volume seek bars. Default value is 100.
<code>percentString()</code>	A function that calculates a percent based on the <i>value</i> and <i>maximum value</i> of a seek bar.
<code>scope.fillStyle()</code>	Returns the width of the seek bar fill element based on the calculated percent.

Eventually we'll add `scope.value` and `scope.max` to the view, but for now we'll just add `scope.fillStyle()` using the `ngStyle directive`, which allows us to set CSS styles on an HTML element conditionally:

~/bloc/bloc-jams-angular/app/templates/directives/seek_bar.html

```
<div class="seek-bar">
-   <div class="fill"></div>
+   <div class="fill" ng-style="fillStyle()"></div>
    <div class="thumb"></div>
</div>
```

Note that `scope` does not precede the method name in the view (e.g. `scope.fillStyle()`). The directive knows which attributes and methods are on its `scope` and can be used in the view. We could not, for instance, call `percentString()` in the view because it is not on the directive's `scope` object.

Update the Seek Bar from a Click Event

The logic we've added to the directive so far is similar to the logic of the `updateSeekPercentage` **function** in the Foundation. Now that we can calculate the seek bar's `value`, we'll add the first of two functions: one to call when a user clicks on the seek bar:

~/bloc/bloc-jams-angular/app/scripts/directives/seekBar.js

```

...
+ var calculatePercent = function(seekBar, event) {
+   var offsetX = event.pageX - seekBar.offset().left;
+   var seekBarWidth = seekBar.width();
+   var offsetXPercent = offsetX / seekBarWidth;
+   offsetXPercent = Math.max(0, offsetXPercent);
+   offsetXPercent = Math.min(1, offsetXPercent);
+   return offsetXPercent;
+ };

return {
  templateUrl: '/templates/directives/seek_bar.html',
  replace: true,
  restrict: 'E',
  scope: { },
  link: function(scope, element, attributes) {
    scope.value = 0;
    scope.max = 100;

+   var seekBar = $(element);

    var percentString = function () {
      var value = scope.value;
      var max = scope.max;
      var percent = value / max * 100;
      return percent + "%";
    };

    scope.fillStyle = function() {
      return {width: percentString()};
    };

+   scope.onClickSeekBar = function(event) {
+     var percent = calculatePercent(seekBar, event);
+     scope.value = percent * scope.max;
+   };
  }
  ...

```

Attribute / Method

Description

Calculates the horizontal percent along the seek bar

<code>calculatePercent()</code>	Calculates the horizontal percent along the seek bar where the event (passed in from the view as <code>\$event</code>) occurred.
<code>seekBar</code>	Holds the element that matches the directive (<code><seek-bar></code>) as a jQuery object so we can call jQuery methods on it.
<code>scope.onClickSeekBar()</code>	Updates the seek bar value based on the seek bar's width and the location of the user's click on the seek bar.

Despite using jQuery, Angular still dictates the style of our code – we declare in the HTML which element should execute `scope.onClickSeekBar()`:

```
~/bloc/bloc-jams-angular/app/templates/directives/seek_bar.html
```

```
- <div class="seek-bar">
+ <div class="seek-bar" ng-click="onClickSeekBar($event)">
    <div class="fill" ng-style="fillStyle()"></div>
    <div class="thumb"></div>
</div>
```

When a user clicks on a point on the `seek-bar` class element, the function will execute.

Update the Seek Bar from a Mousedown Event

The second function we need to implement is for when a user drags the seek bar thumb. In the Foundation, this logic was the **second half** of the `setupSeekBars` function. We'll reimplement that logic in `seekBar.js` using much of the original code:

```
~/bloc/bloc-jams-angular/app/scripts/directives/seekBar.js
```

```

...
scope.onClickSeekBar = function(event) {
    var percent = calculatePercent(seekBar, event);
    scope.value = percent * scope.max;
};

+ scope.trackThumb = function() {
+     $document.bind('mousemove.thumb', function(event) {
+         var percent = calculatePercent(seekBar, event);
+         scope.$apply(function() {
+             scope.value = percent * scope.max;
+         });
+     });
+
+     $document.bind('mouseup.thumb', function() {
+         $document.unbind('mousemove.thumb');
+         $document.unbind('mouseup.thumb');
+     });
+ };
...

```

Attribute / Method	Description
<div>scope.trackThumb()</div>	<p>Similar to <code>scope.onClickSeekBar</code>, but uses <code>\$apply</code> to constantly apply the change in value of <code>scope.value</code> as the user drags the seek bar thumb.</p>

We use `$document` as we did in the Foundation, but with Angular, `$document` must be injected as a dependency for us to use it. Add it as a dependency to the `seekBar` directive:

```
~/bloc/bloc-jams-angular/app/scripts/directives/seekBar.js
```

```

(function() {
-   function seekBar() {
+   function seekBar($document) {
        ...
    }

    angular
    .module('blocJams')
-   .directive('seekBar', seekBar);
+   .directive('seekBar', ['$document', seekBar]);
})();

```

The `scope.trackThumb` function should execute when a user interacts with the `thumb` class element in the view. More specifically, when the event is a mousedown event. Add an `ngMousedown` directive, which allows us to specify custom behavior on a mousedown event, to trigger the `scope.trackThumb` function:

~/bloc/bloc-jams-angular/app/templates/directives/seek_bar.html

```




<div class="seek-bar" ng-click="onClickSeekBar($event)">
    <div class="fill" ng-style="fillStyle()"></div>
-   <div class="thumb"></div>
+   <div class="thumb" ng-mousedown="trackThumb()"></div>
</div>

```

Test the seek bars in the browser. We should be able to slide or click to a new position. Note, however, that the thumb will not change position. You will implement that ability in the following assignment.

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details.

9. Directives: Part 1		
 Assignment	 Discussion	 Submission



10 Directives: Part 2

We can drag and click on the seek bars, but they're not yet functional. When we interact with the seek bars, we need to change the song position and the volume. We'll start by working on the ability to alter the song position.

Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

Pass Attributes to a Directive

In the previous checkpoint, we discussed the three arguments that we can pass to a directive's `link` function: `scope`, `element`, and `attributes`. We'll explore the third argument, `attributes`, to modify the `seekBar` directive so we can change the playback position of the currently playing song.

Recall from the previous checkpoint that we've already declared `scope.value` and `scope.max` attributes. We'll add these attributes to the directive in the view:

```
~/bloc/bloc-jams-angular/app/templates/player_bar.html
```

```
...  
- <seek-bar></seek-bar>  
+ <seek-bar value="" max=""></seek-bar>  
...
```

When we declared `scope.value` and `scope.max` we gave them default values of `0` and

100, respectively. In the view, however, we've set the value of `value` to `length` and the value of `max` to `length`.

The second expression, `length`, is one with which we are already familiar. The expression will evaluate to the length of the currently playing song.

The first expression, `currentTime`, however, does not yet exist. Update the `SongPlayer` service with this new attribute:

~/bloc/bloc-jams-angular/app/scripts/services/SongPlayer.js

```
...
SongPlayer.currentSong = null;
+ /**
+  * @desc Current playback time (in seconds) of currently playing song
+  * @type {Number}
+  */
+ SongPlayer.currentTime = null;
...
```

When the page first loads, the timecode will show 'NaN'; this is expected, and we will modify it in the next checkpoint.

If the length of a song is the `max` value of the seek bar, then the current playback time of the song is the `value` of the seek bar, which determines the position of the seek bar thumb.

"Observe" Attribute Changes

To monitor the value changes of these attributes in a manner specific to this directive, we have to "observe" them. We can notify the directive of all changes to attribute values by using the `$observe` method on the Angular attributes object:

~/bloc/bloc-jams-angular/app/scripts/directives/seekBar.js


```

...
var seekBar = $(element);

+ attributes.$observe('value', function(newValue) {
+     scope.value = newValue;
+ });
+
+ attributes.$observe('max', function(newValue) {
+     scope.max = newValue;
+ });
...

```

This code observes the values of the attributes we declare in the HTML by specifying the attribute name in the first argument. When the observed attribute is set or changed, we execute a callback (the second argument) that sets a new scope value (`newValue`) for the `scope.value` and `scope.max` attributes. We use the directive's scope to determine the location of the seek bar thumb, and correspondingly, the playback position of the song.

Set the Playback Position of a Song

One priority in programming is reusability. It saves time (in the form of future programming and planning) when a function or other unit is devised to be broadly usable.

Our `seekBar` directive is a general tool. Our goal is to design it so that a user can use it in any instance that requires the ability to control the state of something using a seek bar interface. So far, for Bloc Jams, this only includes song position and volume. To maintain the universality of the seek bar as a tool, we don't want to include instance-specific behavior in the directive. Instead, we want to pass in the behavior dynamically.

In other words, we want to specify an external function that the directive will call when the seek bar position changes. In doing so, the directive can remain a general visualization that supports a broad set of use cases.

Add an attribute named `on-change` that takes an expression to execute when we change the `value` of the seek bar. In this case, we'll pass it a callback that **sets the time of the song** to correspond to the `value` of the seek bar:

```
~/bloc/bloc-jams-angular/app/templates/player_bar.html
```

```

...
- <seek-bar value="" max=""></seek-bar>
+ <seek-bar value="" max="" on-change="playerBar.songPlayer.setCurrentTime(value)"></seek-bar>
...

```

Note that `value` passed into the `onChange` call is not inherently the same as `scope.value`. It is a variable that shares the name. We can (and will, shortly) pass in `scope.value` as the `value` that is the argument for this function, but the two are not the same by definition.

The `SongPlayer` service doesn't yet have a `setCurrentTime` method, so we'll add it to the service now. Add the method below the `SongPlayer.next` method:

~/bloc/bloc-jams-angular/app/scripts/services/SongPlayer.js

```

...
+ /**
+  * @function setCurrentTime
+  * @desc Set current time (in seconds) of currently playing song
+  * @param {Number} time
+  */
+ SongPlayer.setCurrentTime = function(time) {
+   if (currentBuzzObject) {
+     currentBuzzObject.setTime(time);
+   }
+ };
...

```

The `setCurrentTime` method checks if there is a current Buzz object, and, if so, uses the Buzz library's `setTime` **method** to set the playback position in seconds.

Evaluate the `on-change` Expression

We want Angular to treat the `on-change` attribute differently than the `value` or `max` attributes. We don't want `on-change` to act like a number, string, or static object. Instead, we want the directive to evaluate the `on-change` expression and execute it.

To make sure the directive evaluates the attribute, we'll attach it to the directive's scope, rather than the attributes object. Scoping the attribute allows us the flexibility to specify how we want to handle the value passed to the `on-change` attribute:

```

...
return {
  templateUrl: '/templates/directives/seek_bar.html',
  replace: true,
  restrict: 'E',
-  scope: { },
+  scope: {
+    onChange: '&'
+  },
...

```

The `&` in the isolate scope object is a **type of directive scope binding**. The three types of directive scope bindings – `@`, `=`, and `&` – allow us to treat the value of the given attribute differently. The `&` binding type provides a way to execute an expression in the context of the parent scope.

Pass Updated value to onChange

The function we evaluate through `onChange` may seem like the final piece of the puzzle, but if we use the web inspector to view the value of `value`, we find that there isn't one!



```

▼ <div class="seek-control">
  ▼ <div class="seek-bar ng-isolate-scope" ng-click="onClickSeekBar($event)"
    value max="161.71" on-change="playerBar.songPlayer.setCurrentTime(value)">
    <div class="fill" ng-style="fillStyle()" style="width: 10.064%;"></div>
    <div class="thumb" ng-style="thumbStyle()" ng-mousedown="trackThumb()"
      style="left: 10.064%;"></div>
    </div>
    <div class="current-time ng-binding"></div>
  </div>

```

Recall the `onClickSeekBar` and `trackThumb` methods we created in `seekBar.js`. We update the value of `scope.value` and yet we don't see that update reflected on the attribute in the view.

We need to pass the updated `value` to the `onChange` attribute. To do so, we'll write a function to call in the `onClickSeekBar` and `trackThumb` methods that will send the updated `scope.value` to the function evaluated by `onChange`:

```

...
scope.onClickSeekBar = function(event) {
    var percent = calculatePercent(seekBar, event);
    scope.value = percent * scope.max;
+   notifyOnChange(scope.value);
};

scope.trackThumb = function() {
    $document.bind('mousemove.thumb', function(event) {
        var percent = calculatePercent(seekBar, event);
        scope.$apply(function() {
            scope.value = percent * scope.max;
+           notifyOnChange(scope.value);
        });
    });

    $document.bind('mouseup.thumb', function() {
        $document.unbind('mousemove.thumb');
        $document.unbind('mouseup.thumb');
    });
};
...

```

We name the function `notifyOnChange` because its purpose is to notify `onChange` that `scope.value` has changed. Add the function to the directive's logic:

~/bloc/bloc-jams-angular/app/scripts/directives/seekBar.js

```

...
scope.trackThumb = function() {
    ...
};

+ var notifyOnChange = function(newValue) {
+   if (typeof scope.onChange === 'function') {
+       scope.onChange({value: newValue});
+   }
+ };
...

```

This function is short but dense. Let's break it down:

- We test to make sure that `scope.onChange` is a function. If a future developer fails to pass a function to the `on-change` attribute in the HTML, the next line will not be reached, and no error will be thrown.
- We pass a full function call to the `on-change` attribute in the HTML – `scope.onChange()` calls the function in the attribute.
- The function we pass in the HTML has an argument, `value`, which isn't defined in the view (remember that it's not the same as `scope.value`). Using built-in Angular functionality, we specify the value of this argument using hash syntax. Effectively, we're telling Angular to insert the local `newValue` variable as the `value` argument we pass into the `SongPlayer.setCurrentTime()` function called in the view.

View Bloc Jams in the browser and test the click and mousedown events. The song playback position should update accordingly.

Broadcast the Time Change to the Application

We've added the ability to set the song playback time via the seek bar, but once we've set a song position, the thumb and fill don't update with the progress of the song. We also have yet to address the undefined value of `` in our seek bar directive.

The `SongPlayer` service needs to handle updating time as the song progresses. We want to know, in any part of our Angular application, when the time updates. This ensures that regardless of where we reference the time (in this case, we reference it with ``), our application is aware that it is changing on a second-by-second basis.

In Angular, one way to scope a variable to all parts of an application is to use the `$rootScope` service. Every Angular application has just one `$rootScope`, from which all other scopes inherit. Any Angular component, then, can access `$rootScope` variables, events, and functions.

`$rootScope` is not to be used lightly, however. In the Foundation we learned that polluting a namespace is not good practice, and the same is true for an application's `$rootScope`. Only append to the `$rootScope` if absolutely necessary.

Because `$rootScope` is a service, we must inject it as a dependency before we can use it:

```
~/bloc/bloc-jams-angular/app/scripts/services/SongPlayer.js
```

```

    (function() {
-     function SongPlayer(Fixtures) {
+     function SongPlayer($rootScope, Fixtures) {

        ...

    };

    angular
        .module('blocJams')
-       .factory('SongPlayer', ['Fixtures', SongPlayer]);
+       .factory('SongPlayer', ['$rootScope', 'Fixtures', SongPlayer]);
    })();

```

To update the song's playback progress from anywhere, we'll add a `$rootScope.$apply` event in the `SongPlayer` service. This creates a custom event that other parts of the Angular application can "listen" to. We've dealt with common JavaScript events before, like click, mouseover, and mousedown. This will be our first custom event. Add the `$apply` to the `SongPlayer.setSong` method so that it starts "applying" the time update once we know which song to play:

~/bloc/bloc-jams-angular/app/scripts/services/SongPlayer.js

```

...
var setSong = function(song) {
    if (currentBuzzObject) {
        currentBuzzObject.stop();
        SongPlayer.currentSong.playing = null;
    }

    currentBuzzObject = new buzz.sound(song.audioUrl, {
        formats: ['mp3'],
        preload: true
    });

+    currentBuzzObject.bind('timeupdate', function() {
+        $rootScope.$apply(function() {
+            SongPlayer.currentTime = currentBuzzObject.getTime();
+        });
+    });

    SongPlayer.currentSong = song;
};
...

```

We used the Buzz library `bind()` **method** in **the Foundation as well**. `timeupdate` is one of a number of HTML5 audio events we can use with Buzz's `bind()` method.



The `bind()` method adds an event listener to the Buzz sound object – in this case, we listen for a `timeupdate` event. When the song playback time updates, we execute a callback function. This function sets the value of `SongPlayer.currentTime` to the current playback time of the current Buzz object using another one of the Buzz library methods: `getTime()`, which gets the current playback position in seconds. Using `$apply`, we apply the time update change to the `$rootScope`.

Reload the app in the browser. We should now see the seek bar update as the song plays!

Git

Commit your checkpoint work in Git. See **Git Checkpoint Workflow: After Each Checkpoint** for details.

How would you rate this checkpoint and assignment?





11 Filters

The seek bar now updates as a song plays, but the song playback and total time are still represented in seconds, which isn't the way we'd normally view the duration of a song. To solve this, we'll add an **Angular filter**.

Filters format data used in Angular applications. We can use them either directly with an expression in a view, or as an injectable service in the JavaScript logic of other Angular components. Angular has **several built-in filters** that illustrate the types of intended use cases for filters, like formatting numbers for **currency** and **dates**, or strings with a particular **capitalization**.

Git

Create a new Git feature branch for this checkpoint. See **Git Checkpoint Workflow: Before Each Checkpoint** for details.

Create a `timecode` Filter

Like controllers, services, and directives, filters are defined on an Angular module.

Within the `scripts` directory, create a `filters` directory. Within the `filters` directory, create a file named `timecode.js` and register a `timecode` filter:

```
~/bloc/bloc-jams-angular/app/scripts/filters/timecode.js
```



```
+ (function() {  
+     function timecode() {  
+         return function(seconds) {  
+             return output;  
+         };  
+     }  
+  
+     angular  
+         .module('blocJams')  
+         .filter('timecode', timecode);  
+ })();
```

Filter functions must return another function which takes at least one argument, the input of the filter – in this case, our input is `seconds`. We'll take the number of seconds (e.g. "61") and convert it to a time-readable format (e.g. "1:01").

Review the `filterTimeCode` **function** created in the Foundation. We've already written the exact logic that we need to use in our filter. Copy the logic from the `filterTimeCode` function and add it to the Angular filter:

```
~/bloc/bloc-jams-angular/app/scripts/filters/timecode.js
```

```

(function() {
    function timecode() {
        return function(seconds) {
+           var seconds = Number.parseFloat(seconds);
+           var wholeSeconds = Math.floor(seconds);
+           var minutes = Math.floor(wholeSeconds / 60);
+           var remainingSeconds = wholeSeconds % 60;
+
+           var output = minutes + ':';
+
+           if (remainingSeconds < 10) {
+               output += '0';
+           }
+
+           output += remainingSeconds;
+
            return output;
        };
    }

    angular
        .module('blocJams')
        .filter('timecode', timecode);
})();

```

Note that we've changed the argument from `timeInSeconds` to just `seconds`.

Add the `timecode.js` script source to `index.html`, as we've done with all the other JavaScript files so far.

Use a Filter in an Angular View

We can use filters directly in a view's HTML with Angular expressions. In the expression, filters are delimited by the vertical pipe character (`|`), provided after a variable in the view's scope, for example:

```
{{ exampleVar | exampleFilter }}
```

Many filters **take optional arguments**, which are delimited by a colon (`:`) that follows the

filter name:

```
{{ exampleVar | exampleFilter:arg }}
```

Multiple arguments are possible, and are passed in with repeated colons:

```
{{ exampleVar | exampleFilter:argOne:argTwo }}
```

Use the `timecode` Filter in The View

Unlike services, we do not need to inject a filter as a dependency unless we use it within the code of an Angular component, such as a service, directive, or controller. For Bloc Jams, we'll use the filter in the view only, and therefore won't need to inject it as a dependency anywhere.

To use the `timecode` filter in the view, add a vertical pipe (`|`) after each expression of time, followed by the name of the filter: `timecode`. We display time in the song rows for each song, as well as in the player bar for the current and total time of the currently playing song – **update all instances in which we display time in the view to use the `timecode` filter.**

For example, in the player bar template, update the current time and total time of the song with the `timecode` filter by adding the following code:

```
~/bloc/bloc-jams-angular/app/templates/player_bar.html
```

```
...
- <div class="current-time">{{ playerBar.songPlayer.currentTime }}</div>
+ <div class="current-time">{{ playerBar.songPlayer.currentTime | timecode }}</div>
- <div class="total-time">{{ playerBar.songPlayer.currentSong.duration }}</div>
+ <div class="total-time">{{ playerBar.songPlayer.currentSong.duration | timecode }}</div>
...
```

View Bloc Jams in the browser and play a song to test the filter. The timecode filter should work. Notice that when the view first loads, however, that "NaN:NaN" appears. "NaN" means "not a number" and does not, unfortunately, refer to the tasty Indian flat bread. When the view first loads and there is no current Buzz object, there is no time to display. We need to add a statement that checks for this condition.

Add a Condition that Checks for NaN

Add the following conditional statement to `timecode.js`:

~/bloc/bloc-jams-angular/app/scripts/filters/timecode.js

```
...
return function(seconds) {
    var seconds = Number.parseFloat(seconds);

+   if (Number.isNaN(seconds)) {
+       return '-:--';
+   }

    var wholeSeconds = Math.floor(seconds);
    var minutes = Math.floor(wholeSeconds / 60);
    var remainingSeconds = wholeSeconds % 60;

    var output = minutes + ':';

    if (remainingSeconds < 10) {
        output += '0';
    }

    output += remainingSeconds;

    return output;
}
...
```

If `seconds` is not a number, then we return "-:--" to the view. Reload Bloc Jams in the browser and view our working filter in action. Then go get some delicious naan to celebrate, if you feel so inclined.

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details.

How would you rate this checkpoint and assignment?



11. Filters

 **Assignment**

 Discussion

 Submission

1. Read the **AngularJS Developer Guide on filters**.
2. Take a look at **some of Angular's included filters** to get a sense for the contexts in which a filter is appropriate.
3. **Optional:** The Buzz library has a method named `toTimer()` that formats seconds in an easy to read timer like "00:00", which is what we've just accomplished with the `timecode` filter. Read the **documentation for the `toTimer()` method** and rework the `timecode` filter logic to use the Buzz method instead of your custom code.