Mastering the Ionic Framework: Learn to Build & Deploy Native Speed HTML5 Based Apps

Your instructor

(/author/eric) **Eric Simons (/author/eric)**
(https://twitter.com/ericsimons40) (https://github.com/ericsimons)

Download Completed Source Code ()

A Phoenix Arises

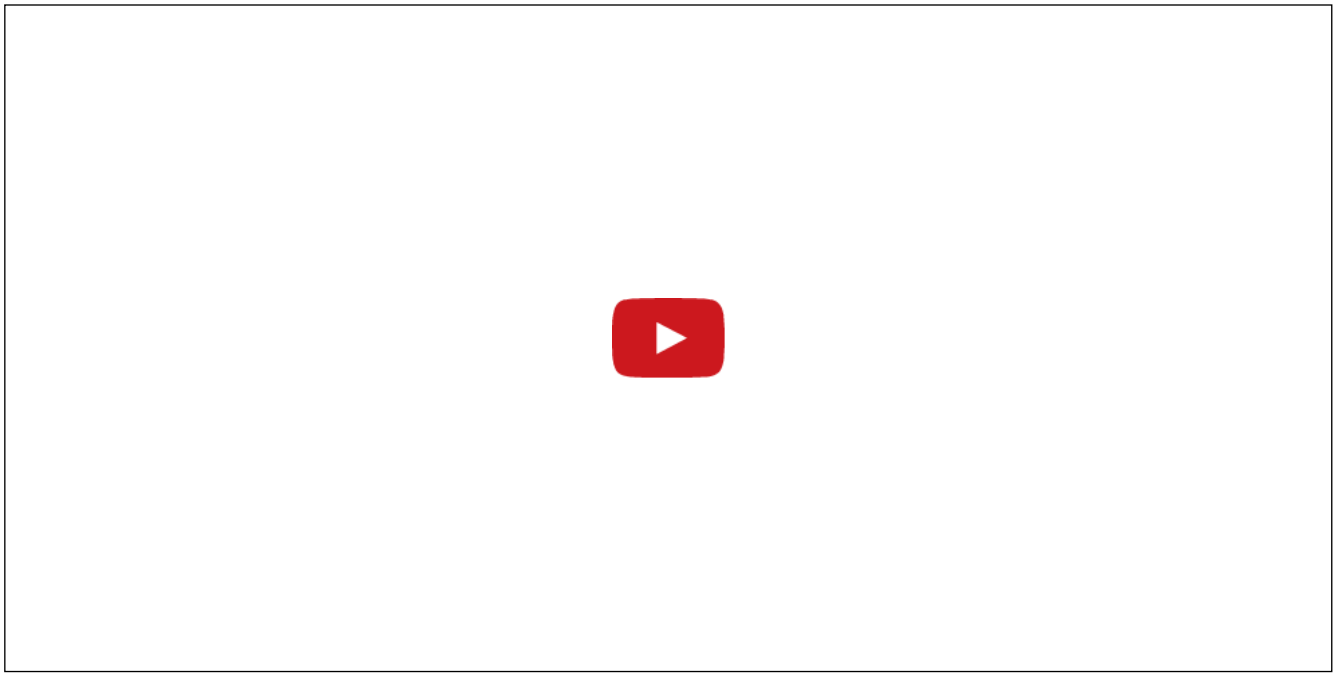
Web development technologies have evolved at an incredible clip over the past few years. We've gone from rudimentary DOM manipulation with libraries like jQuery to supercharged web applications organized & powered by elegant MV* based frameworks like AngularJS. Pair this with significant increases in browser rendering speeds, and it is now easier than ever before to build production quality applications on top of Javascript, HTML5, and CSS3.

While these advances have been incredible, they are only just starting to affect the clear platform of the future: mobile. For years, mobile rendering speeds were atrocious, and the MVC frameworks & UI libraries provided by iOS and Android were far superior to writing mobile apps using web technologies. There were also some very public failures -- Facebook famously wrote their first iOS app in 2011 using HTML5 but ended up scrapping it due to terrible performance.

For years now, hybrid apps have been mocked and jeered by native app developers for being clunky and ugly, having subpar performance, and having no advantages over native apps. While these may have been valid reasons in 2011, they are now virtually baseless, thanks to a collection of new technologies that have emerged over the past two years. With these technologies, you can design, build, and deploy robust mobile apps faster than you could with native technologies, all while incurring little to no app performance penalties. This is thanks in large part to super fast mobile browser rendering speeds and better JavaScript performance. This course is designed to teach you how to effectively use these new technologies to build insanely great mobile apps.

Without further ado, we'd like to welcome you to the future of mobile app development, freed from the shackles of native languages & frameworks. Let's learn what the new mobile stack consists of and how it works.

Introducing Ionic.



Before, building hybrid apps was a chore -- not because it was hard to build web pages, but because it was hard to build full-fledged web applications. With AngularJS, that has changed. As a result, Angular became the core innovation that made hybrid apps possible. The bright folks at Drifty were some of the first to realize this and subsequently created the Ionic Framework (http://ionicframework.com/) to bridge the gap between AngularJS web apps and hybrid mobile apps. Since launching a little over a year ago, the Ionic Framework has quickly grown in popularity amongst developers (http://www.google.com/trends/explore?hl=en-US&q=ionic+framework&cmpt=q&tz&tz&content=1) and their main Github repo (https://github.com/driftyco/ionic) has over 13K stars as of this writing.

Ionic provides similar functionality for AngularJS that iOS UIKit (https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIKit_Framework/) provides for Obj-C/Swift, and that Android UI elements (http://developer.android.com/guide/topics/ui/overview.html) provides for Java. Core mobile UI paradigms are available to developers out of the box, which means that developers can focus on building apps, instead of common user interface elements. Some examples of these include list views (http://ionicframework.com/docs/api/directive/ionList/), stateful navigation (http://ionicframework.com/docs/api/directive/ionNavView/), tab bars (http://ionicframework.com/docs/nightly/api/directive/ionTabs/), action sheets (http://ionicframework.com/docs/api/service/$ionicActionSheet/), and so much more (http://ionicframework.com/docs/nightly/api/).

- [x] For a more in-depth view of where Ionic fits into our hybrid app stack, read this blog post (http://ionicframework.com/blog/where-does-the-ionic-framework-fit-in/) from one of Ionic's core developers and read the "Overview" section (http://ionicframework.com/docs/overview/) on their About page.

Ionic is a great solution for creating both mobile web apps and native apps. The first sections of this course will go over structuring Ionic apps that can run on the web. Then we will cover packaging that same exact code into a native app. We will be using a build tool called Cordova for packaging our app. For those unfamiliar with Cordova, it is the open source core of Adobe's proprietary PhoneGap build system. Adobe describes it with this analogy: Cordova is to PhoneGap as Blink is to Chrome. Basically, PhoneGap is Cordova plus a whole bunch of other Adobe stuff.

- [x] Read this post for a more in depth understanding of the differences between Cordova and PhoneGap (http://ionicframework.com/blog/what-is-cordova-phonegap/)

The folks at Ionic have done a fantastic job of making Cordova super easy to use by directly wrapping it in their 'ionic' command line tool (don't worry, we'll cover this later). Just remember that Cordova is something that is running under the hood of your hybrid app that you will rarely need to worry about, but we will cover some common interactions with it in this course.

What we're going to build

We will be building an app called Songhop, a "Tinder for music" app that allows you to listen to 30-second song samples and favorite the ones you like. This is based on a real Ionic/Cordova powered app we built that exists on the iOS App Store (https://itunes.apple.com/us/app/songhop/id899245239?mt=8) -- feel free to download it to get a feeling for what Ionic is capable of (and rate it 5 stars :). It's also worth noting that it only took us a month to build the Songhop app that's on the App Store, so that should give you an idea of how fast you can build & iterate using Ionic / Cordova.

You can also see a live demo of the completed application we'll be building here (https://ionic-songhop.herokuapp.com) (resize your browser window to the size of a phone for the best experience).

We'll be covering a wide variety of topics in this course: scaffolding a new application, testing it in the emulator, installing native plugins for manipulating audio & files, swipe gestures for our interface, installing the app on your own device, deploying to the iOS & Android app stores, and so much more.

Prerequisites

- [x] **Basic programming / JavaScript knowledge.** If you don't know JavaScript, this guide is a good place to start (http://sivers.org/learn-js).

- [x] **Familiarity with AngularJS.** We will be using AngularJS for much of our app logic. If you aren't familiar with AngularJS, we highly recommend going through the course "A Better Way to Learn AngularJS" (https://thinkster.io/a-better-way-to-learn-angularjs/).

In addition, you should be comfortable with basic web application concepts including REST (http://www.restapitutorial.com/lessons/whatisrest.html) and CRUD (http://en.wikipedia.org/wiki/Create,_read,_update_and_delete).

It's best to have either a Linux based OS or Mac to complete this tutorial. While it's certainly possible to develop on Windows, we won't be covering the specifics of running, debugging, and troubleshooting for Windows based systems.

- [x] **Install Node.js**. Since you will need to install various packages for Node.js, we recommend that you follow these installation instructions (https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager) which use npm (https://www.npmjs.com/).

- [x] **Install the Ionic CLI (http://ionicframework.com/docs/cli/install.html)** through npm by running `npm install -g ionic`. We'll be using it to develop and test our app in a web browser, and later for packaging our app into native binaries for mobile devices.

Recommendations for completing this course

We have structured this course to accommodate folks who are interested in Ionic for building mobile websites as well as those who are interested in building hybrid apps. The first chapters of this course will exclusively cover the JS & CSS portions of the Ionic framework (the only knowledge necessary for building mobile websites), and in the last few chapters we will learn how to package & deploy our application natively for iOS and/or Android.

Throughout this course, links to additional concepts and information will be included. You can use these links as supplementary material that can help you gain insight into the stack and its various components. As always, if you have any questions, Google and Stackoverflow are your best bet, but if you're unsure about something specific to this course, **feel free to tweet me @ericsimons40 (https://twitter.com/ericsimons40)**.

We're firm believers in actually writing code while learning a new language or framework. Therefore, we strongly encourage you to type out all the code instead of copy+pasting it.

Starting our project

When you want to start a new Ionic project (http://ionicframework.com/docs/cli/start.html), typically all you need to do is run `ionic start myApp blank`, and it will scaffold a new application for you. However, for simplicity's sake, we've created a starter project specifically for this course that includes all the base HTML/CSS/JS files we'll be using.

- [x] Clone the repo for this course (https://github.com/EricSimons/ionic-course), navigate to the /code/ folder, and then run the command `npm install`.

Now that we have the Songhop project downloaded to our computer, let's get it running in our browser! To do this, we will use the Ionic CLI's command 'serve'. This command starts a local web server with live reload enabled, then opens your browser directly to your application. As you develop your app, it will automatically refresh your web browser every time you save a file. *You can learn more about how the ionic serve command works & various configuration options available here (http://ionicframework.com/docs/cli/test.html).*

- [x] In your terminal, head into the ionic-songhop-app folder and run `ionic serve`.

If all went well, you should be seeing this in your browser window (be sure to resize your browser to be about the size of a phone, or use your browser's phone emulation tools):

The guts of an Ionic application

When you scaffold a new application using the Ionic CLI, it automatically creates a handful of files to be used as the baseboard for your hybrid app development. This post (ionic-app-structure/) walks through all of these files and what purposes they serve. The www folder is where you'll be spending most of your time, as that's where you will be writing your JavaScript, HTML and CSS for the Songhop app. Lets explore the different files in /www and see how they're connected to each other.

The www/index.html file is where your application starts: it loads the JavaScript & CSS files you write (from www/js and www/css), the JavaScript & CSS files that your application depends on, and sets up HTML elements for your AngularJS app to bind to.

Open up app/index.html in your code editor and look over its structure. Notice that we're attaching our AngularJS application to the body element `<body ng-app="songhop">` and all of our views are being rendered in an ion-nav-view (http://ionicframework.com/docs/api/directive/ionNavView/) (and notice the ion-nav-back-button (http://ionicframework.com/docs/api/directive/ionNavBackButton/) too):

```
<body ng-app="songhop">
  <!--
    The nav bar that will be updated as we navigate between views.
  -->
  <ion-nav-bar class="bar-stable">
    <ion-nav-back-button>
    </ion-nav-back-button>
  </ion-nav-bar>
  <!--
    The views will be rendered in the <ion-nav-view> directive below
    Templates are in the /templates folder (but you could also
    have templates inline in this html file if you'd like).
  -->
  <ion-nav-view></ion-nav-view>
</body>
```

As we navigate through our app, Ionic uses the AngularUI Router (https://github.com/angular-ui/ui-router) to populate ion-nav-view's with necessary templates and controllers. At this point, you might be wondering, how do we tell an ion-nav-view which templates to display? If you open up /www/js/app.js, in the config block, it defines the states of our application. There are three states currently defined for our app: an abstract state (https://github.com/angular-ui/ui-router/wiki/Nested-States-%26-Nested-Views#abstract-states) called 'tab' that displays templates/tabs.html, and two tab states ('tab.discover', 'tab.favorites') for our Discover and Favorites pages, respectively.

You may have noticed that our `tab.discover` and `tab.favorites` states specify views to be displayed in (in this case, tab.discover is asking to be displayed in a view titled 'tab-discover'):

```
.state('tab.discover', {
  url: '/discover',
  views: {
    'tab-discover': {
      templateUrl: 'templates/discover.html',
      controller: 'DiscoverCtrl'
    }
  }
})
```

If you open up templates/tabs.html, you'll see that there are two ion-nav-view elements wrapped inside their respective ion-tab (http://ionicframework.com/docs/api/directive/ionTabs/). This allows us to programmatically display our templates into a specified tab, with the added benefit of each tab having its own navigation history.

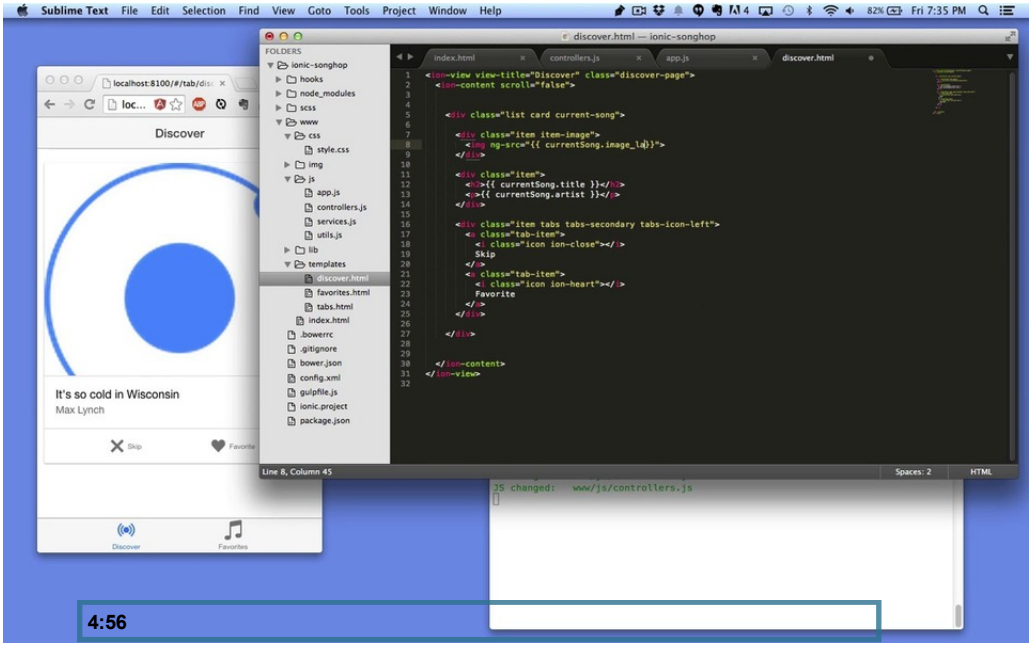
Finally, let's take a look at the templates of our Discover and Favorites pages. Our templates/discover.html file contains an Ionic Card Image (http://ionicframework.com/docs/components/#card-showcase) containing the song information along with favorite/skip buttons, and templates/favorites.html contains an ion-list (http://ionicframework.com/docs/api/directive/ionList/) of our favorite songs. You'll notice that both pages are wrapped in an ion-view (http://ionicframework.com/docs/api/directive/ionView/) for navigation and header bar purposes, and then an ion-content (http://ionicframework.com/docs/api/directive/ionContent/) where we place all of our content.

We've used a couple of Ionic's CSS Components (http://ionicframework.com/docs/components/) and AngularJS Extensions (http://ionicframework.com/docs/api/) in this base application, and we strongly recommend checking out other components and extensions that Ionic can do out of the box. Knowing what Ionic can do for you will allow you to build applications quickly, as you can lean on Ionic's pre-built components & extensions instead of writing your own hand-rolled solutions.

Now that we've explored our base application structure and how everything is wired together, lets start building out the functionality our application!

Building interface functionality

Lets start with our main interface, "Discover", where the user will be presented with songs to "favorite" or "skip".In a few chapters we will retrieve these songs from a server, but for now we'll add some mock data to the discover controller to prototype with.



Playback speed

1

◂◂ 9s

Add the `$scope.songs` array to your discover controller (feel free to copy and paste)

```
.controller('DiscoverCtrl', function($scope) {
  // our first three songs
  $scope.songs = [
     {
        "title":"Stealing Cinderella",
        "artist":"Chuck Wicks",
        "image_small":"https://i.scdn.co/image/d1f58701179fe768cff26a77a46c56f291343d68",
        "image_large":"https://i.scdn.co/image/9ce5ea93acd3048312978d1eb5f6d297ff93375d"
     },
     {
        "title":"Venom - Original Mix",
        "artist":"Ziggy",
        "image_small":"https://i.scdn.co/image/1a4ba26961c4606c316e10d5d3d20b736e3e7d27",
        "image_large":"https://i.scdn.co/image/91a396948e8fc2cf170c781c93dd08b866812f3a"
     },
     {
        "title":"Do It",
        "artist":"Rootkit",
        "image_small":"https://i.scdn.co/image/398df9a33a6019c0e95e3be05fbaf19be0e91138",
        "image_large":"https://i.scdn.co/image/4e47ee3f6214fabbbed2092a21e62ee2a830058a"
     }
  ];
```

When a user is on the discover page, at any given time there will be a song that's currently playing. Since we'll need to display the song's metadata in the template (song name, artist, etc), lets create a scope variable called `currentSong` that will hold all of the current song's information.

When we first initialize the discover controller, lets set `currentSong` to the first song in our songs array:

```
// initialize the current song
$scope.currentSong = angular.copy($scope.songs[0]);
```

You'll notice that we used angular.copy() (https://docs.angularjs.org/api/ng/function/angular.copy), which makes a deep copy of the object, instead of creating a reference to the object in our songs array. If you're unfamiliar with how `angular.copy()` works, you can learn more by watching this video (https://thinkster.io/egghead/angular-copy).

Now that the current song is being exposed to the scope, we can wire up our template to the `currentSong` object.

Update our discover.html template to show the current song:

```
    <div class="item item-image">
      <img ng-src="{{ currentSong.image_large }}">
    </div>

    <div class="item">
      <h2>{{ currentSong.title }}</h2>
      <p>{{ currentSong.artist }}</p>
    </div>
```

At this point you should be seeing the album art, artist name, and title of the first song: "Stealing Cinderella" by Chuck Wicks. If you're feeling unmotivated right now, go listen to this song on Spotify, because it totally sucks, and you'll want to hit the skip button out of sheer spite. So for everyone's sanity, let's make that skip button work (as well as the favorite button).

In our discover controller, let's create a scope method calledl `sendFeedback()` that is fired when the skip and favorite buttons are pressed. This method will accept one argument: a boolean where true = the song was favorited, and false = the song was skipped.

Create the `sendFeedback()` method, which (for now) just changes the `currentSong` to a random song in our `songs` array. (For those wondering about the unused `bool` argument, don't worry - we'll be using it very soon).

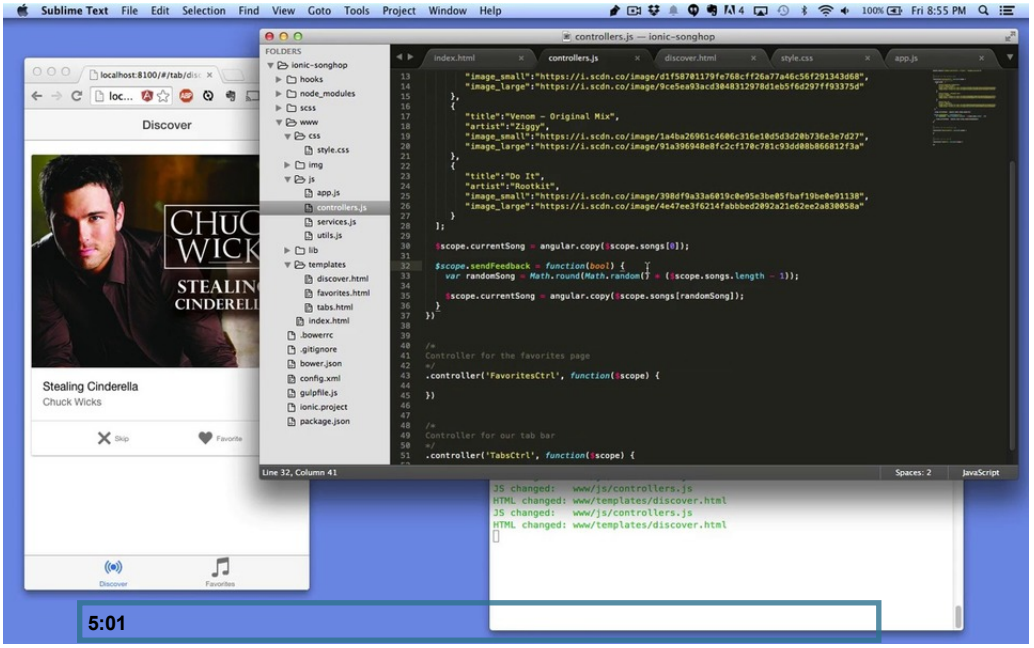
```
// fired when we favorite / skip a song.
$scope.sendFeedback = function (bool) {

  // set the current song to one of our three songs
  var randomSong = Math.round(Math.random() * ($scope.songs.length - 1));

  // update current song in scope
  $scope.currentSong = angular.copy($scope.songs[randomSong]);

}
```

Now, wire up the skip & feedback buttons to fire the `sendFeedback()` method, with `false` and `true` as the argument respectively:

```
    <div class="item tabs tabs-secondary tabs-icon-left">
      <a class="tab-item" ng-click="sendFeedback(false)">
        <i class="icon ion-close"></i>
        Skip
      </a>
      <a class="tab-item" ng-click="sendFeedback(true)">
        <i class="icon ion-heart"></i>
        Favorite
      </a>
    </div>
```

Now, when you hit skip or favorite, you will proceed to a new song!

Using animations



Playback speed

1

◂◂ 9s

Having the song information change is cool, but it would be *super* cool if the entire card animated left or right if the user skipped or favorited, respectively. We can do this by using angular animations (https://docs.angularjs.org/guide/animations); open up css/style.css and you can see the `ng-hide` styles associated with `.discover-page .current-song`. We've also defined separate animations for `.current-song.skipped.ng-hide` and `.current-song.favorited.ng-hide` - this means we will have to add the CSS class `.skipped` or `.favorited` to our current song div, depending if the user skips or favorites.

☑ First, let's update our `sendFeedback()` method to enable animations. `currentSong.rated` will tell the scope whether the current song was favorited (true) or skipped (false). We will also tell the current card to hide itself with `currentSong.hide()`.

```
$scope.sendFeedback = function (bool) {

  // set variable for the correct animation sequence
  $scope.currentSong.rated = bool;
  $scope.currentSong.hide = true;

  $timeout(function() {
    // $timeout to allow animation to complete before changing to next song
    // set the current song to one of our three songs
    var randomSong = Math.round(Math.random() * ($scope.songs.length - 1));

    // update current song in scope
    $scope.currentSong = angular.copy($scope.songs[randomSong]);

  }, 250);
}
```

☑ Since we're using $timeout, we'll need to inject it into our discover controller:

```
.controller('DiscoverCtrl', function($scope, $timeout) {
```

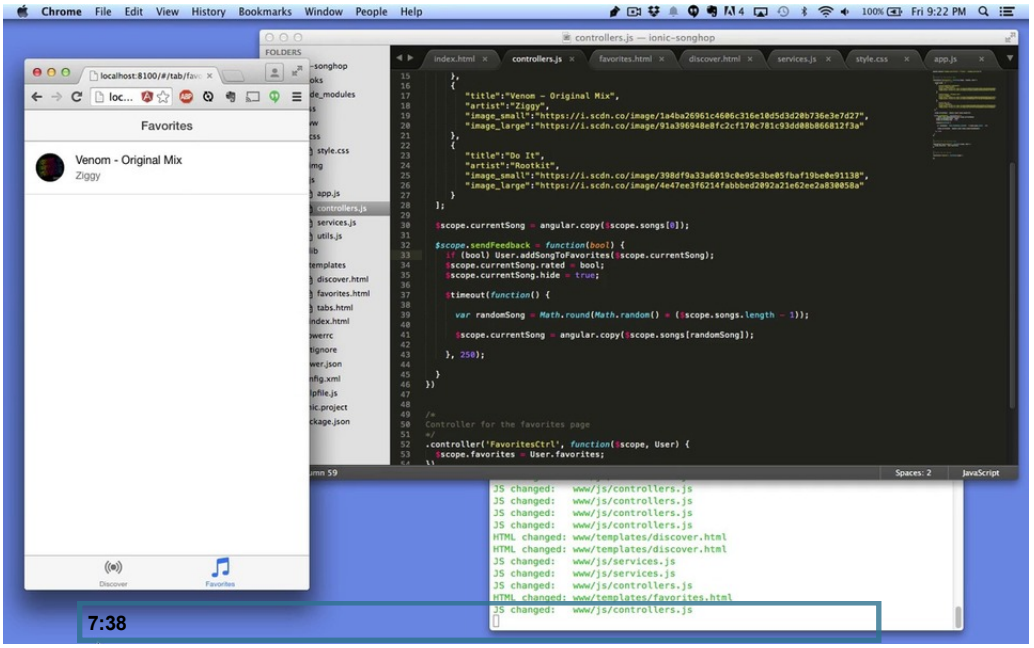
Now open up discover.html and wire these two variables up to our CSS animation classes.

```
    <div class="list card current-song"
         ng-class="{skipped: currentSong.rated == false,
                    favorited: currentSong.rated == true}"
         ng-show="currentSong && !currentSong.hide">
```

Now, our sexy skip and favorite animations should work!

Adding, Removing and Retrieving Favorited Songs

Right now, nothing happens when we favorite a song - but it should actually appear over in our favorites tab. To do that, we need to create a service that will add, remove, and retrieve the songs we've favorited.



Playback speed

1

9s

Let's create a User factory (https://docs.angularjs.org/guide/providers) in services.js, which we'll use to store an array of our favorite songs:

```
angular.module('songhop.services', [])
.factory('User', function() {

  var o = {
    favorites: []
  }

  return o;
});
```

Now, create a method for adding songs to the `favorites` array:

```
  o.addSongToFavorites = function(song) {
    // make sure there's a song to add
    if (!song) return false;

    // add to favorites array
    o.favorites.unshift(song);
  }
```

We'll need to be able to display this on our favorites page, so crack open our favorites controller and drag down our list of favorites into scope:

```
.controller('FavoritesCtrl', function($scope, User) {
  // get the list of our favorites from the user service
  $scope.favorites = User.favorites;
})
```

Now, open up favorites.html and have our `<ion-list>` repeat through our list of favorites:

```
    <ion-item ng-repeat="song in favorites" class="item-avatar">
      <img ng-src="{{ song.image_small }}">
      <h2>{{ song.title }}</h2>
      <p>{{ song.artist }}</p>
    </ion-item>
```

Finally, let's add songs to our favorites when the user hits the favorite button.

In our discover controller, inject User as a dependency:

```
.controller('DiscoverCtrl', function($scope, $timeout, User) {
```

And then add the current song to our favorites at the beginning line of our `sendFeedback()` method:

```
  // first, add to favorites if they favorited
  if (bool) User.addSongToFavorites($scope.currentSong);
```

Go ahead and favorite a couple of songs, then go to the favorites page -- all of our songs now show up!

But what if we want to remove a song from our favorites?

Let's add a method in our User service to do that:

```
o.removeSongFromFavorites = function(song, index) {
  // make sure there's a song to add
  if (!song) return false;

  // add to favorites array
  o.favorites.splice(index, 1);
}
```

In our favorites controller, expose a method that will fire the `removeSongFromFavorites()` User method:

```
$scope.removeSong = function(song, index) {
  User.removeSongFromFavorites(song, index);
}
```

You could just write `$scope.removeSong = User.removeSongFromFavorites`, but if we ever want to have a success message or perform other scope related activities when a song is removed, we need to do it this way.

Finally, let's add an (http://ionicframework.com/docs/api/directive/ionOptionButton/) for deleting the song from the (http://ionicframework.com/docs/api/directive/ionItem/) `ng-repeat` in favorites.html:

```
  <ion-list>
    <ion-item ng-repeat="song in favorites" class="item-avatar">
      <img ng-src="{{ song.image_small }}">
      <h2>{{ song.title }}</h2>
      <p>{{ song.artist }}</p>

      <ion-option-button class="button-assertive"
                  ng-click="removeSong(song, $index)">
        <i class="ion-minus-circled"></i>
      </ion-option-button>
    </ion-item>
  </ion-list>
```

Now, you can swipe a song to the right, hit the delete button, and the song will disappear - pretty neat!

Our base interface functionality is complete. In the next section, we'll start wiring our app up to data retrieved from a server.

Wiring Up to A Server

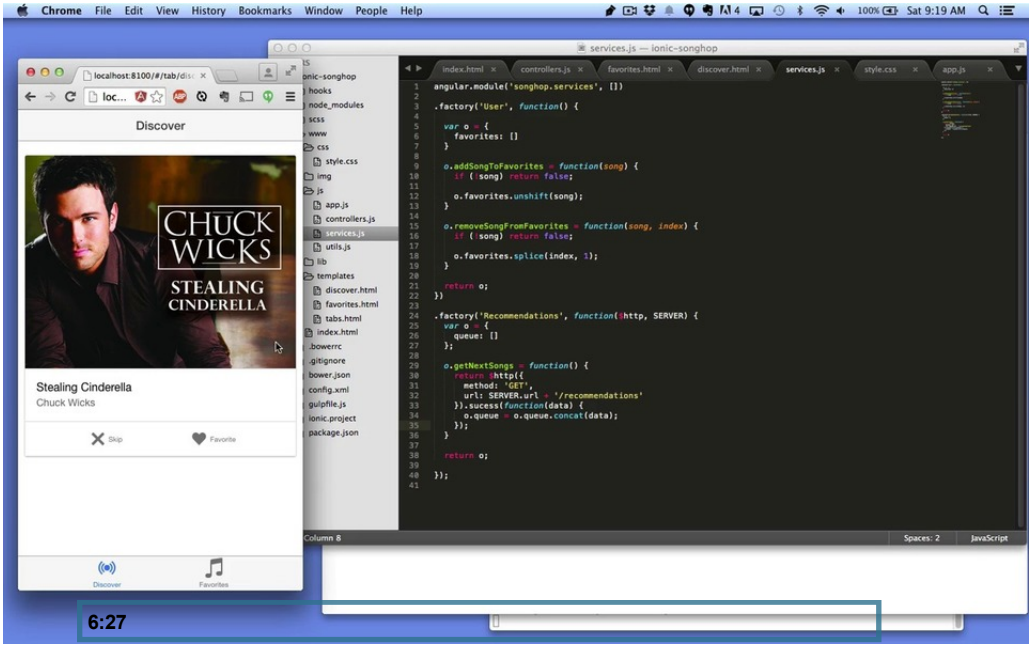
For this course, we've created a node.js server with which our app will be interacting. There is a publicly accessible version of this server available on Heroku at `https://ionic-songhop.herokuapp.com`. Very soon, we will be releasing a version of this server that you can run locally, but until then, you can just use the Heroku server.

You'll notice in app.js that we've defined a constant called `SERVER` that has a url attribute:

```
.constant('SERVER', {
  // if using local server
  //url: 'http://localhost:3000'

  // if using our public heroku server
  url: 'https://ionic-songhop.herokuapp.com'
});
```

This allows us to easily swap our backend out with just one line of code. For now, you'll be using the hosted version we're providing for you over at `https://ionic-songhop.herokuapp.com`, but in the future, it will be trivial use a local server instead.

The first thing we'll wire up is actual song recommendations instead of using the mock data. To do this, let's create a "Recommendations" service that will handle retrieving, storing, and manipulating our queue of song recommendations.



Playback speed

1

9s

- [x] In services.js, create a new factory called `Recommendations` that returns a object containing an array (named `queue`) of our current recommendations. Since this factory will be interacting with our server, be sure to inject `$http` and our `SERVER` constant:

```
.factory('Recommendations', function($http, SERVER) {
  var o = {
    queue: []
  };

  return o;
})
```

Our recommendations queue will be empty until we retrieve songs from our server. The server route we need to hit to retrieve new song recommendations is a GET request to http://SERVER-URL/recommendations (http://SERVER-URL/recommendations). It will return an array of ten random songs, which you can see in action here: https://ionic-songhop.herokuapp.com/recommendations (https://ionic-songhop.herokuapp.com/recommendations).

- [x] Create the `getNextSongs()` method that makes a $http (https://docs.angularjs.org/api/ng/service/$http) GET request to /recommendations. The `success()` function will add the new songs to our `queue` array:

```
o.getNextSongs = function() {
  return $http({
    method: 'GET',
    url: SERVER.url + '/recommendations'
  }).success(function(data){
    // merge data into the queue
    o.queue = o.queue.concat(data);
  });
}
```

When our app first loads up, we'll want to fire a function to initialize our first set of songs. In our `DiscoverCtrl`, we can just fire `Recommendations.getNextSongs()`. When it returns successfully, we'll set the current song to be the first one in our queue (aka `Recommendations.queue[0]`). The song at index 0 of `Recommendations.queue` should always be the current song.

- [x] Inject `Recommendations` into `DiscoverCtrl`, remove `$scope.songs` and `$scope.currentSong`, and replace with `getNextSongs()` w/ callback

```
.controller('DiscoverCtrl', function($scope, $timeout, User, Recommendations) {
  // get our first songs
  Recommendations.getNextSongs()
    .then(function(){
      $scope.currentSong = Recommendations.queue[0];
    });
```

Our favorite and skip buttons no longer work, as our `sendFeedback()` method isn't wired up to our `Recommendations` service. We need to create a method in our `Recommendations` service that allows us to remove the current song from the queue and proceed to the next one. At the same time we'll check to see if the queue is low on songs, and if it is, we'll fire `getNextSongs()`. This ensures our users will never run out of songs to sample.

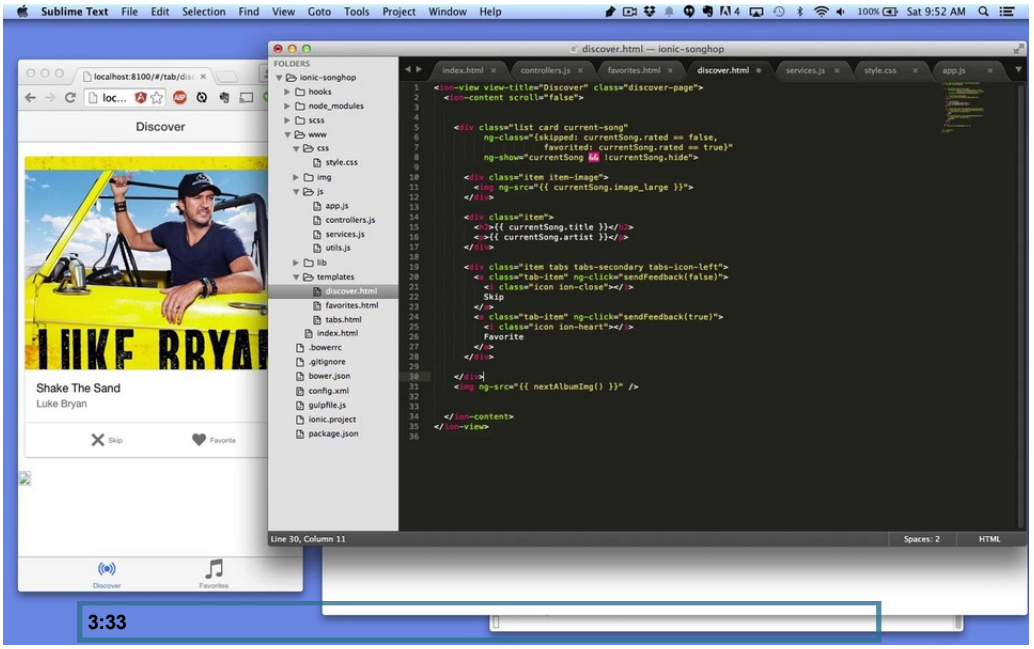
- [x] Create a `nextSong()` method that pops the first array element off of our `queue` and fires `getNextSongs()` if necessary:

```
o.nextSong = function() {
  // pop the index 0 off
  o.queue.shift();

  // low on the queue? lets fill it up
  if (o.queue.length <= 3) {
    o.getNextSongs();
  }

}
```

In the `sendFeedback` method, fire `Recommendations.nextSong()` and update `$scope.currentSong` accordingly:

```
  // prepare the next song
  Recommendations.nextSong();

  $timeout(function() {
    // $timeout to allow animation to complete
    $scope.currentSong = Recommendations.queue[0];
  }, 250);
```

Cool, now favoriting and skipping works again. Skip a few songs quickly though and you might notice that the images are taking a long time to load, especially if you're on a slow internet connection (which is often the case on mobile). The way to solve this is by caching the image for the next song in the queue - but what's the best way of doing that in a hybrid app?



Playback speed

1

◂◂ 9s

> "You have to learn the rules of the game. And then you have to play better than anyone else." - *Albert Einstein*

When we were building the original Songhop app, we originally planned to download the next image to the device and then provide a link to the local file in cache. This would've required us to use native plugins as well as added a good amount of code to our app. This would also be the only solution to use if you were developing this app in a native language, like Swift/Obj-C or Java. However, we have a killer advantage over native apps: we can lean on the web browser for common tasks, like caching. Just by creating `<img src="hello.jpg" />`, we can place an image in the web browser's cache.

Now, imagine if we had an image tag with an ng-src (https://docs.angularjs.org/api/ng/directive/ngSrc) always pointed at the image url for `Recommendations.queue[1]` (aka the next song in the queue). We can make this image nearly invisible with CSS: 1px tall and 1px wide, with an opacity of 0.01 - yet the browser will still download it and place it in cache. Perfect!

- [x] In our Discover controller, create a method called `nextAlbumImg()` that will return the next album's image:

```
// used for retrieving the next album image.
// if there isn't an album image available next, return empty string.
$scope.nextAlbumImg = function() {
  if (Recommendations.queue.length > 1) {
    return Recommendations.queue[1].image_large;
  }

  return '';
}
```

- [x] Then in the discover.html template, add a div with the class `img-lookahead` that contains a nested image tag:

```
  <div class="img-lookahead">
    <img ng-src="{{ nextAlbumImg() }}" />
  </div>
```

Even if you skip songs as fast as you can, the images are almost always preloaded into the image tag in our img-lookahead div - with no extra code to download them and keep track of their location in cache. This is a great example of how building hybrid apps can often be easier than building native apps.

- [x] Remember - don't be afraid to use the unique features web browsers provide to your advantage! With great power comes great responsibility, and your responsibility when building hybrid apps is to create code that native app developers would drool over.

Playing & managing audio

The key functionality of our app is the ability to quickly sample songs, but getting the proper licensing rights to play song clips is exceptionally difficult. For months we dug around various APIs trying to find a reasonable solution, but to no avail. That was until the summer of 2014, when Spotify made waves by releasing a new set of APIs (https://developer.spotify.com/web-api/) for interacting with their platform. Their new API provides us with the key bit of functionality for our app: 30 second song previews, as well as album art and other relevant song metadata.

For this course, our node.js server will be handling all of the interactions with the Spotify API, which means that we can just focus on playing the song data that is returned to our Ionic client. All you need to know is that the data being returned to us from the /recommendations endpoint is actually being powered by the amazing people at Spotify, so show them some love on Twitter! (https://twitter.com/SpotifyPlatform)

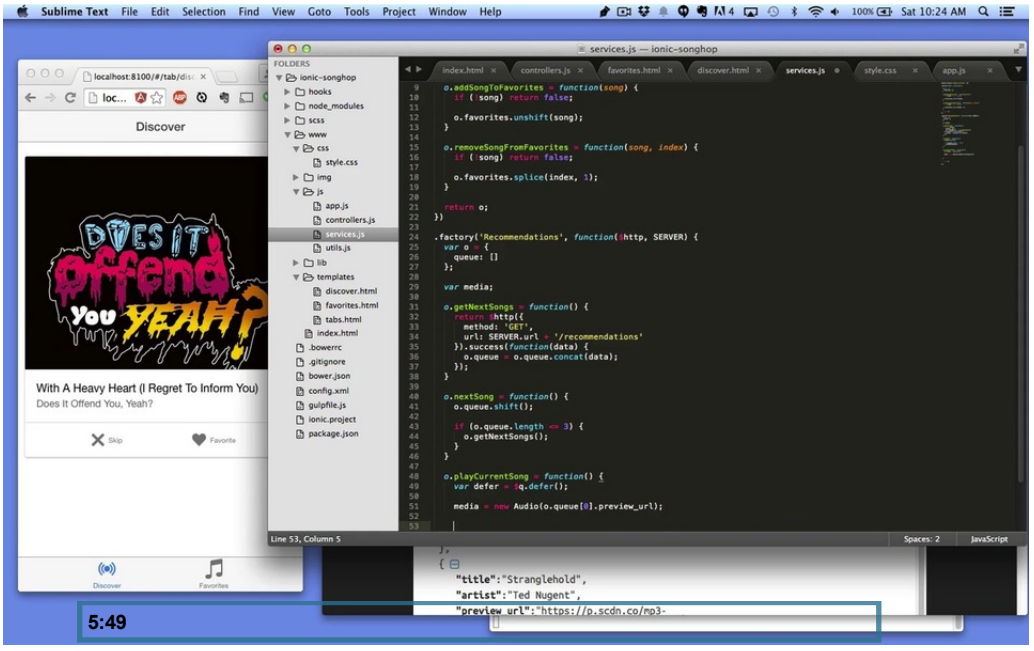
> If you're interested in learning how to use the Spotify API, make sure you enter your email into the subscription box at the top or bottom of this page or tweet me [@ericsimons40](https://twitter.com/ericsimons40 (https://twitter.com/ericsimons40)]. We're releasing another tutorial on how we created the node.js server for this course soon!

Let's figure out how we can play Spotify's 30 second song previews. Every song that is returned to us by the /recommendations endpoint contains the following attributes:

```
{
    "title":"Blood Brothers",
    "artist":"Luke Bryan",
    "preview_url":"https://p.scdn.co/mp3-preview/b92a1dc3c0420d8c6fe1bab6ae15c181bf63926d",
    "image_small":"https://i.scdn.co/image/082645c420a3ea4fb58a2b325c191df1a19cc8c7",
    "image_medium":"https://i.scdn.co/image/beac04c66c73fdb1765ba645bcad6a73e5e9ba90",
    "image_large":"https://i.scdn.co/image/ca93252350b8819ac51252783e1431af9dabd59f",
    "open_url":"https://open.spotify.com/track/4ggPIxXwOWReIBfyCVW8QI",
    "song_id":"54d3f75eb9fd644f061b3c40"
}
```

The attribute `preview_url` contains the URL for this song's mp3 preview clip. You can verify this by visiting the URL in your browser and it will start playing an mp3 stream.

Since modern browsers can play mp3 files without installing any extra stuff, for our app we can simply use the HTMLAudioElement (https://developer.mozilla.org/en-US/docs/Web/API/HTMLAudioElement) for playing 30 second song previews. Instead of cluttering our discover controller with the logic to play & pause songs, let's create methods in our `Recommendations` service that will allow us to easily invoke these actions from any controller.



Playback speed

1

9s

- [x] At the top of the `Recommendations` service, initialize a variable called `media` that we'll use for our HTMLAudioElement:

```
var media;
```

We don't allow the user to manipulate audio playback other than playing and pausing (when navigating to our favorites page).

- [x] Create methods for playing and halting the current song preview:

```
o.playCurrentSong = function() {
  var defer = $q.defer();

  // play the current song's preview
  media = new Audio(o.queue[0].preview_url);

  // when song loaded, resolve the promise to let controller know.
  media.addEventListener("loadeddata", function() {
    defer.resolve();
  });

  media.play();

  return defer.promise;
}

// used when switching to favorites tab
o.haltAudio = function() {
  if (media) media.pause();
}
```

This code uses promises so don't forget to inject the $q (https://docs.angularjs.org/api/ng/service/$q) service into the `Recommendations` service!

- [x] Next, let's modify the `nextSong()` method to halt the current audio clip:

```
o.nextSong = function() {
  // pop the index 0 off
  o.queue.shift();

  // end the song
  o.haltAudio();

  // low on the queue? let's fill it up
  if (o.queue.length <= 3) {
    o.getNextSongs();
  }
}
```

- [x] Now in our `DiscoverCtrl`, fire `Recommendations.playCurrentSong()` when the promise returned from `Recommendations.getNextSongs()` resolves successfully:

```
Recommendations.getNextSongs()
  .then(function(){
    $scope.currentSong = Recommendations.queue[0];
    Recommendations.playCurrentSong();
  });
```

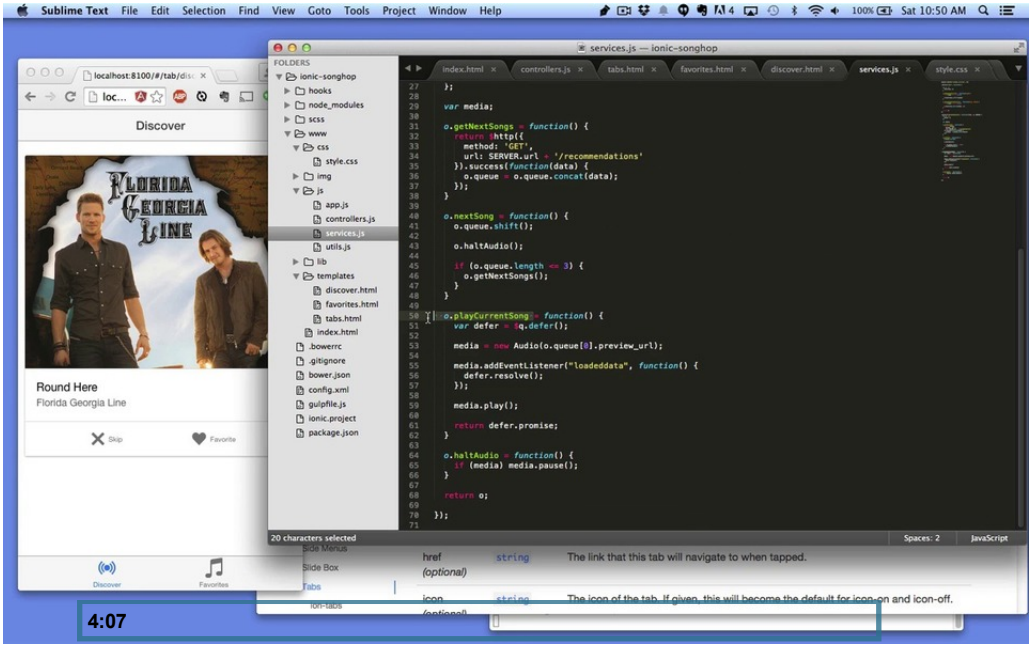
This will cause the song to begin playing when the Discover page is first loaded.

- [x] Finally, we need to add a call to `Recommendations.playCurrentSong()` after we set a new song in `sendFeedback()`:

```
  $timeout(function() {
    // $timeout to allow animation to complete
    $scope.currentSong = Recommendations.queue[0];
  }, 250);

  Recommendations.playCurrentSong();
```

Epic, each song's preview now plays as we skip/favorite the songs in the queue. There's only one problem: if you go to the favorites page, the song being previewed continues to play. We'll need to somehow fire the `haltAudio()` method in our `Recommendations` service when we're navigating to the favorites page, and resume playback of the song when navigating back to the discover page.

If you look at the documentation for ion-tab (http://ionicframework.com/docs/api/directive/ionTab/), you'll see that there are API events for when a tab is being selected (`on-select`) and when it is being deselected (`on-deselect`). For our favorites tab, we want to pause audio when it is being selected, and we want to play audio when it's being deselected (which means we're going back to the Discover page).



Playback speed

1

◂◂ 9s

Create an enteringFavorites method in TabsCtrl that halts the current song's audio:

```
.controller('TabsCtrl', function($scope, Recommendations) {
  // stop audio when going to favorites page
  $scope.enteringFavorites = function() {
    Recommendations.haltAudio();
  }

});
```

Open tabs.html and set `enteringFavorites` as the expression that on-select will evaluate:

```
<ion-tab title="Favorites" icon-off="ion-music-note" icon-on="ion-music-note" on-select="enteringFavorites()" href="#/tab/favorites">
```

When you tap on the favorites page, the current song will now pause. But when you go back to the discover page, the song won't resume. We could just fire `Recommendations.playCurrentSong()` for on-deselect, but that method assumes that there is at least one song in our queue, which is not a good assumption. For example, what if the user is on the favorites page, refreshes, and then goes to the Discover page? playCurrentSong would throw an error trying to call play() on a nonexistent audio object.

We'll need to refactor some code to make this work. Specifically, we need to create an init function that will either retrieve songs (if there aren't any in the queue) or play the current song (if there's at least one song in the queue).

Create an `init()` method in the `Recommendations` service that will either get the next songs or play the current song:

```
  o.init = function() {
    if (o.queue.length === 0) {
      // if there's nothing in the queue, fill it.
      // this also means that this is the first call of init.
      return o.getNextSongs();

    } else {
      // otherwise, play the current song
      return o.playCurrentSong();
    }
  }
```

In DiscoverCtrl, change `getNextSongs()` to `init()`:

```
Recommendations.init()
  .then(function(){
    $scope.currentSong = Recommendations.queue[0];
    Recommendations.playCurrentSong();
  });
```

- [x] Now, create a scope method in TabsCtrl that will fire `Recommendations.init()` when called by on-deselect:

```
$scope.leavingFavorites = function() {
  Recommendations.init();
}
```

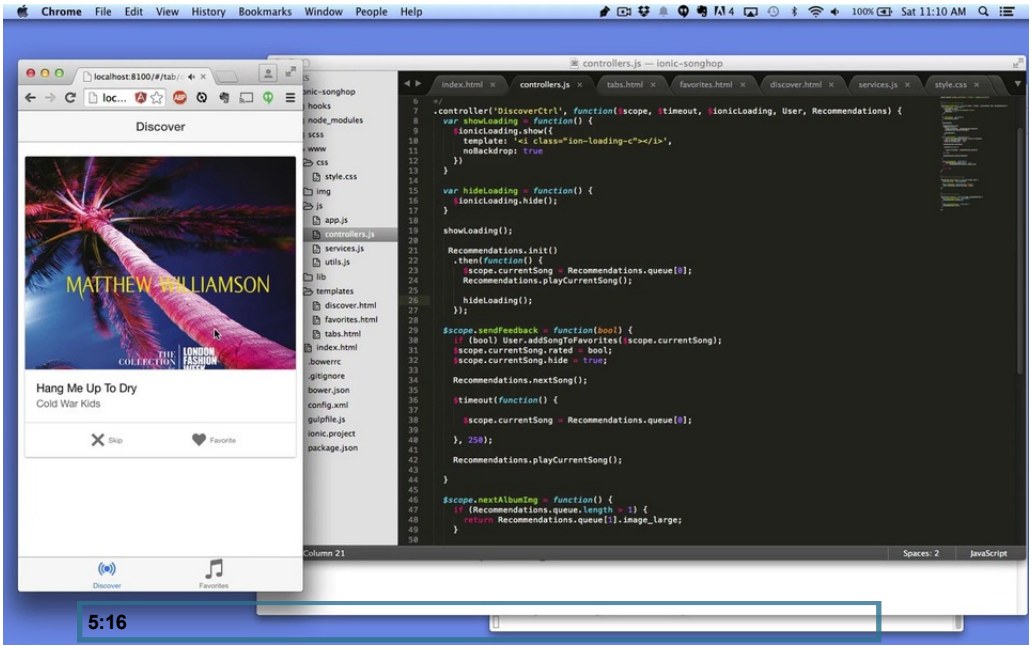
- [x] Finally, invoke the `leavingFavorites()` method `on-deselect` in tabs.html:

```
<ion-tab title="Favorites" icon-off="ion-music-note" icon-on="ion-music-note" on-select="enteringFavorites()" on-deselect="leavingFavo
rites()" href="#/tab/favorites">
```

You should now be able to switch between the discover and favorites pages with automatic playing & pausing of audio. Pretty neat!

Wrapping up the UI and core functionality

When our app is retrieving data from a server, it's important that we give the user some sort of visual indicator while the app is waiting for the server's response. This especially relevant for the music sampling experience, so let's set up some loading indicators on the Discover page.



Playback speed

1

◂◂ 9s

When the application first boots up and requests the first set of song recommendations, we'll want to display a large loading indicator that ideally will block further user input until the loading completes. Ionic just happens to have a fantastic $ionicLoading (http://ionicframework.com/docs/api/service/$ionicLoading/) extension that does exactly this, so let's use it!

- [x] At the top of `DiscoverCtrl`, create methods for showing and hiding the `$ionicLoading` overlay, and then fire the `showLoading()` function (this only happens when the controller is first initialized aka the first time the app is loaded). We'll also need to inject `$ionicLoading` into `DiscoverCtrl`:

```
.controller('DiscoverCtrl', function($scope, $ionicLoading, $timeout, User, Recommendations) {
  // helper functions for loading
  var showLoading = function() {
    $ionicLoading.show({
      template: '<i class="ion-loading-c"></i>',
      noBackdrop: true
    });
  }

  var hideLoading = function() {
    $ionicLoading.hide();
  }

  // set loading to true first time while we retrieve songs from server.
  showLoading();
```

- [x] Add `hideLoading()` to the `Recommendations.init()` success function in `DiscoverCtrl`, as we want to hide the loading indicator once the songs have been retrieved.

If you're running the server locally, you probably won't see the $ionicLoading overlay due to how fast the init promise is resolved. If you shut down your server and refresh the Songhop app, you should be able to see the $ionicLoading overlay.

Our loading indicator works really well! But wait a second - we should let the user know when the song's MP3 has finished loading when we skip/favorite through songs. Instead of having a blocking loading indicator like $ionicLoading, this time we'll have a loading icon next to the title of the song.

- [x] In discover.html, add an `.ion-loading-c` ionicon (http://ionicons.com/) next to the title that will `ng-hide` when `$scope.currentSong.loaded == true`:

```
    <div class="item">
      <h2>{{ currentSong.title }} <span ng-hide="currentSong.loaded"><i class="ion-loading-c"></i></span></h2>
      <p>{{ currentSong.artist }}</p>
    </div>
```

- [x] We'll want to modify the `init()` resolve to set `$scope.currentSong.loaded` to `true` when `Recommendations.playCurrentSong()` resolves successfully:

```
Recommendations.init()
  .then(function(){

    $scope.currentSong = Recommendations.queue[0];

    return Recommendations.playCurrentSong();

  })
  .then(function(){
    // turn loading off
    hideLoading();
    $scope.currentSong.loaded = true;
  });
```

- [x] Then modify `sendFeedback` to set the new song's loading status to `false`, but will set it to `true` after `Recommendations.playCurrentSong` resolves successfully:

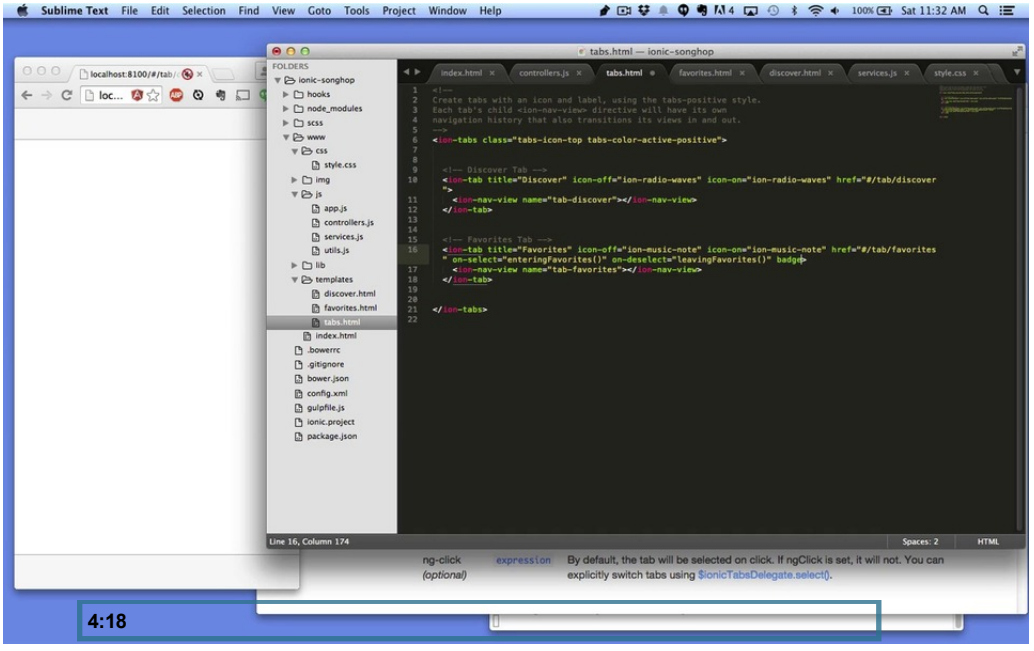
```
// fired when we favorite / skip a song.
$scope.sendFeedback = function (bool) {

  // first, add to favorites if they favorited
  if (bool) User.addSongToFavorites($scope.currentSong);

  // set variable for the correct animation sequence
  $scope.currentSong.rated = bool;
  $scope.currentSong.hide = true;

  // prepare the next song
  Recommendations.nextSong();

  // update current song in scope, timeout to allow animation to complete
  $timeout(function() {
    $scope.currentSong = Recommendations.queue[0];
    $scope.currentSong.loaded = false;
  }, 250);

  Recommendations.playCurrentSong().then(function() {
    $scope.currentSong.loaded = true;
  });

}
```

Sick, now loading indicators work!

Adding badges to the tab bar

As we add songs to our favorites, it would be really nice if there was some sort of visual indicator of how many new favorites we've added. This will also encourage our users to visit their favorites and review their songs.

You may have noticed in the documentation for ion-tab (http://ionicframework.com/docs/api/directive/ionTab/) that you can display badges on a given tab. This is exactly what we need!



Playback speed

1

◂◂ 9s

First, we need to modify our user service to keep track of how many new favorites we have.

☑ Attach a new variable `newFavorites` to the User return object:

```
var o = {
  favorites: [],
  newFavorites: 0
}
```

- [x] When a user favorites a song via `addSongToFavorites()`, we'll increase the `newFavorites` count:

```
  o.addSongToFavorites = function(song) {
    // make sure there's a song to add
    if (!song) return false;

    // add to favorites array
    o.favorites.unshift(song);
    o.newFavorites++;
  }
```

- [x] Create a new method that returns the total number of new favorites the user has:

```
  o.favoriteCount = function() {
    return o.newFavorites;
  }
```

- [x] Expose the `User.favoriteCount()` method to the `TabsCtrl` scope (don't forget to inject the User service):

```
.controller('TabsCtrl', function($scope, User, Recommendations) {
  // expose the number of new favorites to the scope
  $scope.favCount = User.favoriteCount;

  ...
}
```

- [x] When the user taps on the favorites tab, we want to zero out `User.newFavorites`. Lets update our `enteringFavorites()` method to do this:

```
  // method to reset new favorites to 0 when we click the fav tab
  $scope.enteringFavorites = function() {
    User.newFavorites = 0;
    Recommendations.haltAudio();
  }
```

- [x] The final step is to set up our badge in tabs.html. Set the badge type to badge-assertive with the badge content displaying `favCount()`:

```
<ion-tab title="Favorites" icon-off="ion-music-note" icon-on="ion-music-note" badge="favCount()" badge-style="badge-assertive" on-selec
t="enteringFavorites()" on-deselect="leavingFavorites()" href="#/tab/favorites">
```

Now when we favorite songs the number of new favorites show up in the tab bar, but disappears when we navigate to the favorites page. Cool!

The last piece of core functionality missing from our app is the ability to access our favorite songs directly on Spotify website/app. Since Spotify gives us a web URL for every song, we can just open a new browser window pointed at the song's `open_url` attribute.

- [x] In favorites.html, fire a scope method called `openSong()` on ng-click that takes the song as an argument:

```
<ion-item ng-repeat="song in favorites" class="item-avatar" ng-click="openSong(song)">
```

- [x] Then create the open song method in our favorites controller (be sure to inject `$window` into `FavoritesCtrl`):

```
  $scope.openSong = function(song) {
    $window.open(song.open_url, "_system");
  }
```

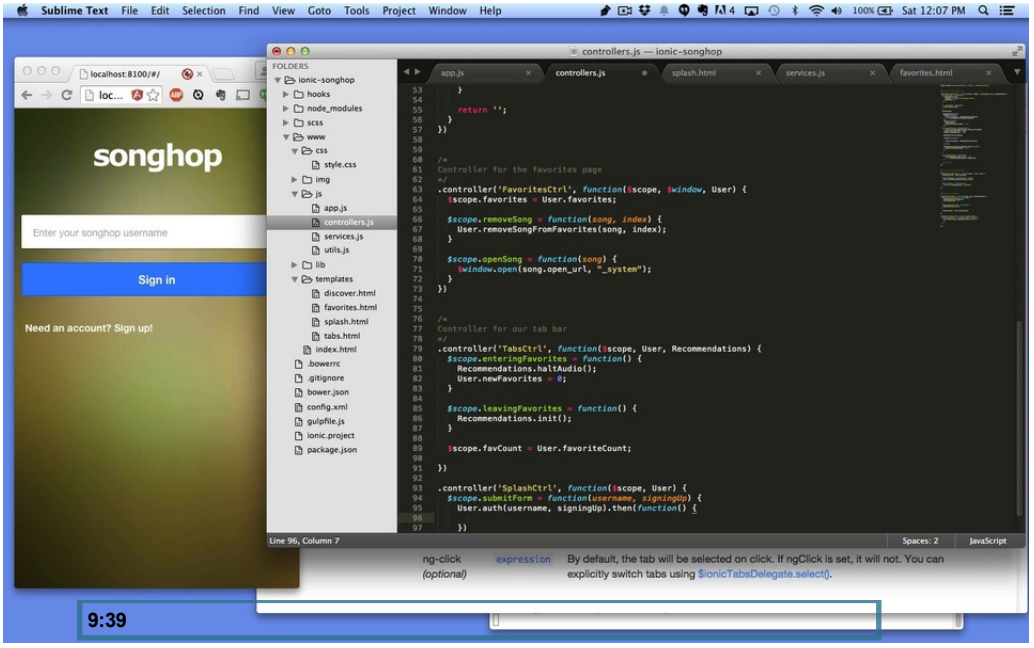
Now we're done! In the next section, we're going to cover authentication & persisting user sessions between app uses.

Creating and persisting user data

When an Ionic application needs to authenticate with a server, it's usually done with token based session management instead of cookies. It's easier to store an access token on the device than to ensure a browser cookie persists across app usages. For this course, we're going to use an insanely simple implementation of token based authentication. If you're planning on implementing token based authentication in a real app, we recommend reading this post (https://auth0.com/blog/2014/01/07/angularjs-authentication-with-cookies-vs-token/) on the topic as well as taking a look at this Github repo (https://github.com/sean-hill/ionic-user-auth-express) of an Ionic app that performs user authentication using bearer tokens, as specified by RFC 6750 (http://tools.ietf.org/html/rfc6750).

Before we jump into authentication with our server, first, we will need to create a page for users to sign in/sign up. For this app, we will only require a username to sign up and sign in (in the real world you'd normally require a password along with the username, but since this isn't the real world, we're keeping it simple).

> Because other people will be using the same Heroku server for this course, use *very* unique usernames to ensure your accounts aren't colliding with others!



Playback speed

1

◂◂ 9s

- [x] In app.js, create a new state called `splash` at the route '/'. Being the root route means that it will be the first page loaded when the app initializes. We also want to update `$urlRouterProvider.otherwise()` to point at '/' from now on, instead of the discover page.

```
// splash page
.state('splash', {
  url: '/',
  templateUrl: 'templates/splash.html',
  controller: 'SplashCtrl'
})
```

```
$urlRouterProvider.otherwise('/');
```

- [x] We told UI-Router that the `splash` state will use a controller called `SplashCtrl`, so lets create it in controllers.js:

```
.controller('SplashCtrl', function($scope) {
});
```

- [x] Create the splash.html file defined in the `splash` state inside the templates/ directory, and paste the following HTML into it:

```
<ion-view view-title="Songhop" hide-nav-bar="true" class="splash-page">
  <ion-content scroll="false">

    <div class="padding logo">
      <h1>songhop</h1>
    </div>


    <div class="login-form">
      <div class="card">
        <div class="list">
          <label class="item item-input">
            <input type="text" ng-model="form.username" placeholder="Enter your {{ form.signingUp ? 'desired' : 'songhop' }} username">
          </label>
        </div>
      </div>

      <div class="padding-left padding-right">
        <button class="button button-block action-button" ng-class="{'button-positive': !form.signingUp, 'button-balanced': form.signing
Up}" ng-click="submitForm(form.username, form.signingUp)">
          {{ form.signingUp ? 'Sign up' : 'Sign in' }}
        </button>
      </div>
    </div>


    <div class="padding signup-opt" ng-click="form.signingUp = !form.signingUp">
      {{ form.signingUp ? 'Have an account? Sign in!' : 'Need an account? Sign up!' }}
    </div>

  </ion-content>
</ion-view>
```

Take a quick glance at what's in our template, as there are a few things worth noting. First, our form is submitted through a scope method: `submitForm(form.username, form.signingUp)`. We pass along the username from the input field, along with a boolean if we're signing up (false means we're logging in). Also note that in our ion-view we request to hide the nav bar with `hide-nav-bar="true"` (since there's no places to navigate to/from on this page).

We need a place to store the session token ID and other data we receive back from the server. Since this is information is related to our user, it seems a natural choice to store this in the User service.

- [x] We'll be storing `username` and `session_id` in the `User` service, so let's define them in our return object:

```
  var o = {
    username: false,
    session_id: false,
    favorites: [],
    newFavorites: 0
  }
```

In our User service, we need a method that will send a username to the server for attempting login or signup and handling the result properly.

- [x] First, inject `$http` and `SERVER` into the User service, as we'll need them for communicating with the server. Then create the `auth()` method that return a `$http` promise. Note that the routes for login and signup are /login and /signup and require a `username` field in the post request:

```
.factory('User', function($http, SERVER) {
```

```
  // attempt login or signup
  o.auth = function(username, signingUp) {

    var authRoute;

    if (signingUp) {
      authRoute = 'signup';
    } else {
      authRoute = 'login'
    }

    return $http.post(SERVER.url + '/' + authRoute, {username: username});
  }
```

- [x] In `SplashCtrl`, create the method `submitForm()` that calls the `User.auth()` method (be sure to inject the `User` and `$state` service):

```
.controller('SplashCtrl', function($scope, $state, User) {

  // attempt to signup/login via User.auth
  $scope.submitForm = function(username, signingUp) {
    User.auth(username, signingUp).then(function(){
      // session is now set, so lets redirect to discover page
      $state.go('tab.discover');

    }, function() {
      // error handling here
      alert('Hmm... try another username.');

    });
  }

});
```

Part of the data we need to retrieve on authentication success is our list of favorite songs. The only problem is that our methods for adding, removing, and retrieving favorites aren't wired up to the server...yet! Let's quickly take care of that.

Adding, removing, and retrieving favorites are actions that are tied to a specific user account. This means that we need to pass along our `session_id` with all of these requests in order to attach the desired request with the correct user.

- [x] Have `addSongToFavorites()` return an `$http` POST request to /favorites with the ID of the song we favorited, along with our user's `session_id`:

```
  o.addSongToFavorites = function(song) {
    // make sure there's a song to add
    if (!song) return false;

    // add to favorites array
    o.favorites.unshift(song);
    o.newFavorites++;

    // persist this to the server
    return $http.post(SERVER.url + '/favorites', {session_id: o.session_id, song_id:song.song_id });
  }
```

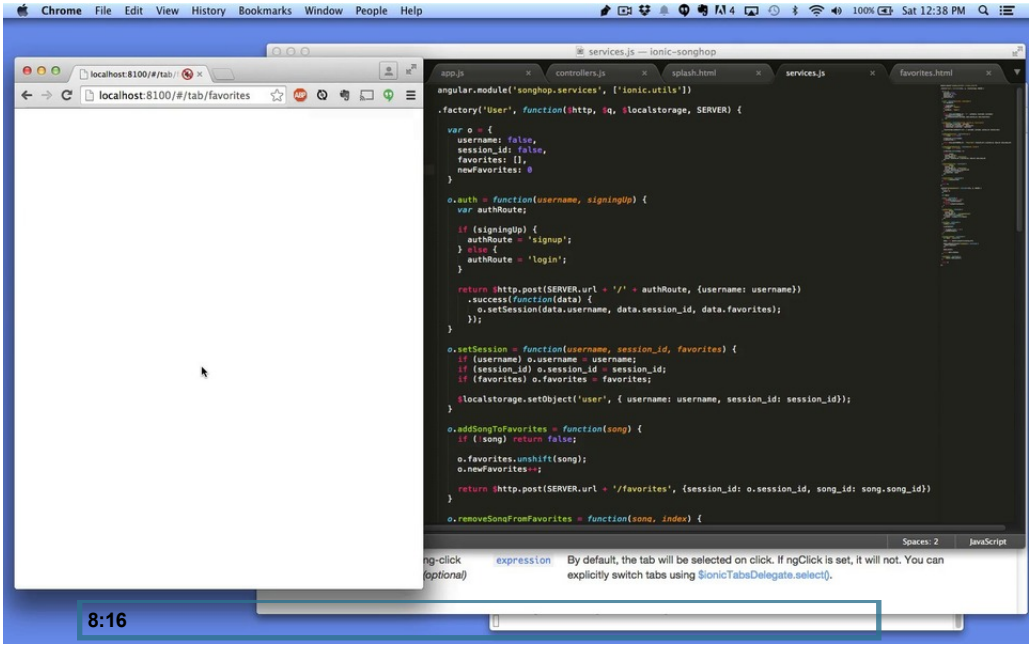
- [x] Have `removeSongFromFavorites()` return an `$http` DELETE request to /favorites with the ID of the song we favorited, along with our user's `session_id`:

```
  o.removeSongFromFavorites = function(song, index) {
    // make sure there's a song to add
    if (!song) return false;

    // add to favorites array
    o.favorites.splice(index, 1);

    // persist this to the server
    return $http({
      method: 'DELETE',
      url: SERVER.url + '/favorites',
      params: { session_id: o.session_id, song_id:song.song_id }
    });

  }
```

- [x] Finally, we'll need to request our list of favorites from the server on load. Let's create a method called `populateFavorites()` that will do this by sending an `$http` GET request to /favorites:

```
  // gets the entire list of this user's favs from server
  o.populateFavorites = function() {
    return $http({
      method: 'GET',
      url: SERVER.url + '/favorites',
      params: { session_id: o.session_id }
    }).success(function(data){
      // merge data into the queue
      o.favorites = data;
    });
  }
```

Right now, our sessions only persist if the browser doesn't reload. This isn't expected functionality, especially if we want to deploy this app natively. We need some way of storing a user's session ID and username across app uses that won't be cleared from cache automatically.



Playback speed

1

◂◂ 9s

To solve this problem, local storage (http://diveintohtml5.info/storage.html) is an ideal solution. In js/utils.js, you can see a service called `$localstorage` that wraps the native browser's local storage API for our convenience. This was actually taken from an Ionic formula (http://learn.ionicframework.com/formulas/localstorage/) created by Max Lynch (https://twitter.com/maxlynch), a cofounder of the company behind the Ionic Framework.

- [x] First, we need to require `ionic.utils` in our services module. Then, require `$localstorage` and `$q` in the User service:

```
angular.module('songhop.services', ['ionic.utils'])
```

```
.factory('User', function($http, $q, $localstorage, SERVER) {
```

- [x] Lets create a method for storing our session data into localStorage (for persisting user info between app uses) as well as the User return object (for accessing by our running application):

```
// set session data
o.setSession = function(username, session_id, favorites) {
  if (username) o.username = username;
  if (session_id) o.session_id = session_id;
  if (favorites) o.favorites = favorites;

  // set data in localstorage object
  $localstorage.setObject('user', { username: username, session_id: session_id });
}
```

- [x] Now, update the return promise in the `auth()` method to fire the `setSession()` method on success with the data returned from the server:

```
return $http.post(SERVER.url + '/' + authRoute, {username: username})
  .success(function(data){
    o.setSession(data.username, data.session_id, data.favorites);
  });
```

In various parts of our app, we'll need to check if there is currently a user session. For example, if a non-authenticated used tries to access the Discover page, we need to detect this and redirect them accordingly.

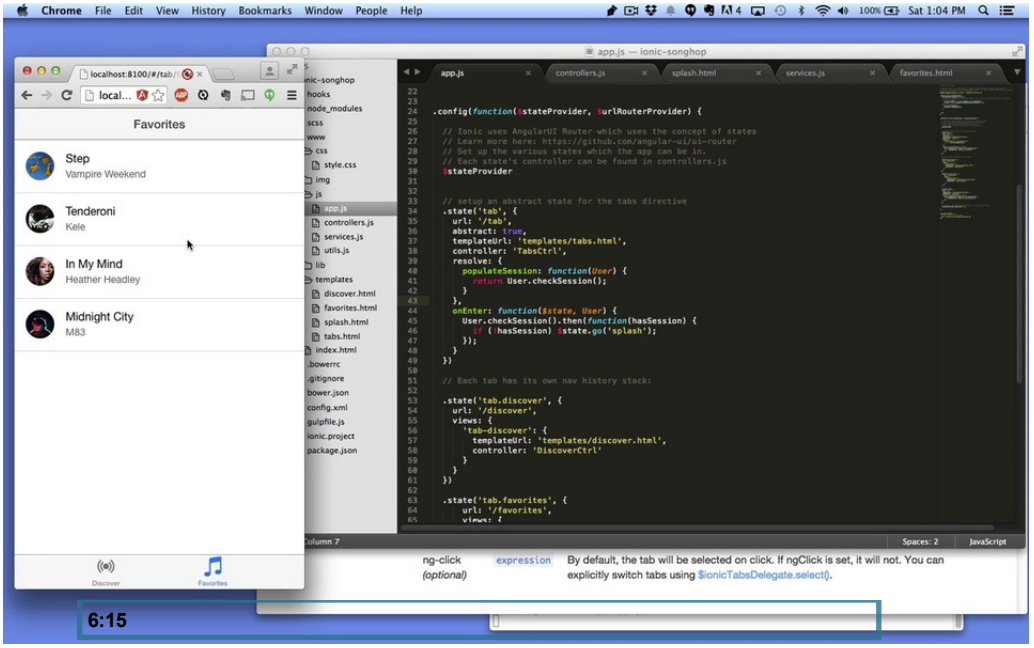
- [x] Create a `checkSession()` method to determine if a user is currently logged in:

```
// check if there's a user session present
o.checkSession = function() {
  var defer = $q.defer();

  if (o.session_id) {
    // if this session is already initialized in the service
    defer.resolve(true);

  } else {
    // detect if there's a session in localstorage from previous use.
    // if it is, pull into our service
    var user = $localstorage.getObject('user');

    if (user.username) {
      // if there's a user, lets grab their favorites from the server
      o.setSession(user.username, user.session_id);
      o.populateFavorites().then(function() {
        defer.resolve(true);
      });

    } else {
      // no user info in localstorage, reject
      defer.resolve(false);
    }

  }

  return defer.promise;
}
```

- [x] If a user wants to log out, we need a `destroySession()` method that clears the User service variables along with our local storage object:

```
// wipe out our session data
o.destroySession = function() {
  $localstorage.setObject('user', {});
  o.username = false;
  o.session_id = false;
  o.favorites = [];
  o.newFavorites = 0;
}
```

We now have all of the methods necessary to ensure non-authenticated users can't access the core parts of the app (and vice versa, if a user is logged in and visits the Splash page, they should be redirected to the Discover page).



Playback speed

1

◂◂ 9s

- [x] In app.js, update the splash state to check if there is a session onEnter (https://github.com/angular-ui/ui-router/wiki#onenter-and-onexit-callbacks):

```
// splash page
.state('splash', {
  url: '/',
  templateUrl: 'templates/splash.html',
  controller: 'SplashCtrl',
  onEnter: function($state, User){
    User.checkSession().then(function(hasSession) {
      if (hasSession) $state.go('tab.discover');
    });
  }
})
```

- [x] Perform the same check on the tab abstract state (which covers both the discover and favorites states, since it's their parent state), but we need to resolve the `checkSession()` method first. If there is a user session in local storage, we need to retrieve the user's favorites from the server, and we don't want to display our app until that data is returned successfully:

```
// setup an abstract state for the tabs directive
.state('tab', {
  url: '/tab',
  abstract: true,
  templateUrl: 'templates/tabs.html',
  controller: 'TabsCtrl',
  // don't load the state until we've populated our User, if necessary.
  resolve: {
    populateSession: function(User) {
      return User.checkSession();
    }
  },
  onEnter: function($state, User){
    User.checkSession().then(function(hasSession) {
      if (!hasSession) $state.go('splash');
    });
  }
})
```

Now users can only access our core app's functionality if they're logged in. But what if a user wants to log out? Lets provide a logout button the favorites page. While we're at it, lets display their username at the top of the favorites page too.

- [x] In `FavoritesCtrl`, expose the `User.username` string to the scope:

```
$scope.username = User.username;
```

- [x] We may want the logout button somewhere else in our app in the future, so we'll put the logic for logging out in `TabsCtrl` (be sure to inject `$window`):

```
$scope.logout = function() {
  User.destroySession();

  // instead of using $state.go, we're going to redirect.
  // reason: we need to ensure views aren't cached.
  $window.location.href = 'index.html';
}
```

- [x] Finally, in favorites.html update the `<ion-view>` title attribute to include both our username and an ion-nav-button (http://ionicframework.com/docs/api/directive/ionNavButtons/) for logging out:

```
<ion-view view-title="Favorites | {{ username }}" class="favorites-page">
  <ion-nav-buttons side="right">
    <button class="button" ng-click="logout()">
      Log out
    </button>
  </ion-nav-buttons>
  <ion-content>
```

And with that, our app is now 100% complete. We've covered a tremendous amount of knowledge about Ionic, but there is still much more to learn. As we mentioned at the beginning of the course, we highly recommending exploring the CSS Components (http://ionicframework.com/docs/components/) and AngularJS Extensions (http://ionicframework.com/docs/api/) that Ionic provides out of the box. The documentation on Ionic's website is hands down some of the best docs I've ever seen, so don't be afraid to peruse it when you have questions or are just looking for some inspiration.

The folks behind Ionic are also building some amazing tools that make your experience working with the Ionic Framework even better. For example, the Ionic Creator (http://creator.ionic.io/) is a drag and drop tool that allows you to quickly mock up applications and have it export the working code (crazy, right?). It gets even better than that, though: They're starting to roll out a new service called ionic.io (http://ionic.io/) that comes with a

handful of killer features, like the ability to update the HTML/CSS/JS of your hybrid app over the air, thus sidestepping Apple's lengthy app review process. Most recently, they announced a push notification service (https://apps.ionic.io/landing/push) that will make sending push notifications across platforms super simple.

The next few years will be an exciting time for hybrid app developers using Ionic, so keep yourself in the loop and follow @IonicFramework (https://twitter.com/Ionicframework). I'll be updating this course with every new release, so follow me @ericsimons40 (https://twitter.com/ericsimons40) to keep your magical Ionic skills up to date.

With the Ionic section of this course completed, let's start learning how to...what's that you say? Encore? Alrighty then...

Some Ionic challenges for those hungry for more

- [x] On the favorites page, include a share button on each ion-item (http://ionicframework.com/docs/components/#item-buttons) that will open an $ionicActionSheet (http://ionicframework.com/docs/api/service/$ionicActionSheet/) with the option to share the song's Spotify URL on Twitter and Facebook.

- [x] Wouldn't it be great if we could swipe the song previews left and right, like on Tinder? Well, with Ionic's Tinder-for-X (http://ionicframework.com/blog/tinder-for-x/) directive, we can do just that. Take a crack at implementing it on the Discover page!

- [x] Get creative! Think of something cool to enhance the experience of using the app, build it, and then tweet us about what you made :) (https://twitter.com/intent/tweet?text=@EricSimons40%20@GoThinkster%20I%20created...)

With the Ionic section of this course *actually* completed, let's start learning how to build this app for use on native devices.

Deploying to the emulator, devices and App Stores

Sometimes, mobile websites just won't cut it. Maybe you need to allow users to download your app from their respective app store, or maybe you need to access native device features that don't have web APIs (like the device's accelerometer, for example). Cordova is the solution.

- [x] Learn exactly what Cordova provides us out of the box on the about section (http://cordova.apache.org/#about) of their website.

We mentioned this at the beginning of the course, but we'll explain it again for those of you who skimmed over it: Cordova is to PhoneGap as Blink is to Chrome. Basically, PhoneGap is Cordova plus a whole bunch of other Adobe stuff. Seriously, read this post that the Ionic folks wrote about the matter (http://ionicframework.com/blog/what-is-cordova-phonegap/) - it's super good, I promise. There's also a great post by Tommy Williams (http://blog.devgeeks.org/post/73789983750/cordova-vs-phonegap-an-update) that should clear up any remaining questions you have about the history and differences between Cordova, Phonegap, and Phonegap build.

This is a common point of confusion when first building hybrid apps, so don't feel bad if you're still a little confused. In fact, the Ionic team wrapped common Cordova functions under their CLI tool to ensure the experience of using Cordova was seamless and intuitive. I've used Cordova extensively without the Ionic CLI and recommend using the two together. The Ionic CLI provides some handy extensions that make the experience better, like automatically livereloading apps that are in the emulator (amongst many other things).

Setting up our project for Cordova

We need to let Cordova know what platforms we intend to deploy our app to. This can be done via the Ionic CLI:

```
$ ionic platform ios android
```

While we're at it, let's install the amazing Cordova keyboard plugin (https://github.com/driftyco/ionic-plugins-keyboard) that the Ionic team built, as it uses a custom implementation of the native keyboard instead of the web browser version:

```
$ ionic plugin add com.ionic.keyboard
```

The syntax for adding and removing plugins is `ionic plugin add [plugin]` and `ionic plugin rm [plugin]`, respectively. This same syntax also applies to adding and removing platforms (`ionic platform rm ios`, etc).

Running apps in the emulator

Simply run the emulate command to open your app in the emulator, where platform is either 'ios' or 'android' (the `-l` flag enables live reloading of the app):

```
$ ionic emulate [platform] -l
```

There is currently a bug in Cordova: After adding a plugin and then attempting to build/emulate/run the project, the build will fail. The way you fix this is by removing the target platform (`ionic platform rm [platform]`) and then adding it again (`ionic platform add ios`). This should be fixed soon, but until then, this is the easiest solution.

- [x] Read this (http://ionicframework.com/docs/cli/run.html) to learn about all of the options available in the Ionic CLI for running & emulating.

At this point, you should see the Songhop app running in the emulator. There's one bug, though - if you favorite a couple of songs, go the favorites page, and tap a song to open it in Spotify, it doesn't open a new browser window. Instead, it opens it in our app's web view. To get around this, we need to install another plugin from Cordova called inappbrowser (https://github.com/apache/cordova-plugin-inappbrowser/blob/master/doc/index.md) that will allow the link to open in the device's native browser.

- [x] Install the Cordova inappbrowser via the Ionic CLI:

```
ionic plugin add org.apache.cordova.inappbrowser
```

Now, if you emulate the app again and tap a song in our favorites, it will open in the device's native web browser!

Challenge

- [x] We're currently using the HTMLAudioElement to playback audio, but Cordova has a media plugin (https://github.com/apache/cordova-plugin-media/blob/master/doc/index.md) that we could use instead. In general, using plugins instead of browser implementations can help ensure your app stays speedy and stable. It often helps with troubleshooting, as Cordova plugins are open source and can therefore be troubleshooted and modified. In fact, **on iOS our app currently crashes after previewing 5-10 songs due to a bug in mobile Safari**, so try modifying the audio playback features of our app to use the Cordova media plugin instead of HTMLAudioElement!

Running the application on your device

To run your app on a device connected to your computer, the command `$ ionic run [platform]` will automatically package and install the app. However, your development environment & device have to be set up properly to make this work. The requirements here have nothing to do with Ionic and Cordova (as they were set by Apple and Google), and as such, the requirements are different for both iOS and Android.

Android

For Android, the process is super simple: enable developer mode (http://www.syncios.com/blog/enable-developer-optionsusb-debugging-mode-on-devices-with-android-4-2-jelly-bean/) on your device, plug it into your computer, and `$ ionic run android` . Super easy, right? Compare that to...

iOS

For iOS, you'll need to setup a development provisioning profile before you can successfully run `$ ionic run ios` . Setting up provisioning profiles first requires an Apple Developer account (https://developer.apple.com/register/index.action), which costs $99/yr. If you think that price is a bit steep, I hear that Apple will also trade you their development tools for your left kidney, should you have one to spare.

Lets dive into what many iOS developers refer to as "Provisioning Profile Hell" -- buckle up, kids, because we're going for a ride (https://www.youtube.com/watch?v=l482T0yNkeo).

Highway to (Provisioning Profile) Hell

- [x] If you're unfamiliar with iOS certificates and provisioning profiles, read this post (http://escoz.com/blog/demystifying-ios-certificates-and-provisioning-files/). It will help you understand how Apple has structured this process.

- [x] Learn how to spin up a provisioning profile for your device by going through this guide (http://www.bignerdranch.com/we-teach/how-to-prepare/ios-device-provisioning.html).

In your project's config.xml file, look for this line: `<widget id="com.ionicframework.songhop616624"` . The ID attribute needs to match the app ID in your provisioning profile, otherwise the application will not be able to run on your device.

If you've set up everything properly, you can now run `$ ionic run ios` and your app will be packaged and installed on your device.

Debugging on iOS/Android

You can use remote debugging tools from Chrome and Safari to debug your running Android and iOS applications, respectively. They're the same inspector debugging tools that you use for normal web development. From the Cordova docs on debugging (https://github.com/phonegap/phonegap/wiki/Debugging-in-PhoneGap):

Chrome Remote Debugging

If you are doing Android PhoneGap debugging and have an Android 4.4 device and Chrome 30+, you can use the new WebView Debugging tools added in Android 4.4. If you are using Cordova 3.3 or higher, this is already supported, and only requires the Debuggable flag in your AndroidManifest.xml. For Cordova 3.2, you will need to enable WebView debugging using some code, or by use of a plugin.

Safari Remote Debugging

If you are doing iOS PhoneGap debugging and have the Safari Develop Menu enabled, you can access the currently active session through the built-in Safari Web Inspector. To activate, go to Develop -> (iPad || iPhone) Simulator (normally, the third menu item) and click the active session you want to connect to. Voila!

Deploying to App Stores

Android Play Store

The Ionic team has a fantastic guide on how to prepare and submit your app to the Google Play Store; you can go through it here (http://ionicframework.com/docs/guide/publishing.html).

iOS App Store

You will need to create a 'Distribution' provisioning profile to submit apps to the App Store (https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/SubmittingYourApp/SubmittingYourApp.html). Once you have that done, follow this guide (https://iphonedevlog.wordpress.com/2013/12/19/using-phonegap-3-3-cli-on-mac-os-x-mavericks-to-build-ios-projects/) on how to prepare & upload your app for App Store submission.

App development is a journey, not a destination

While this course is a very comprehensive overview of how to build amazing hybrid apps, it really is just an overview. The world of hybrid app development is changing at a rapid pace, and there is so much more to learn. To stay up to date on the latest and greatest, I highly recommend following @IonicFramework (https://twitter.com/Ionicframework) and signing up for their newsletter on their website (http://ionicframework.com/). This course will be updated frequently, so if you're interested in keeping your skills up to date with Ionic and other hybrid app technologies, feel free to follow me @ericsimons40 (https://twitter.com/ericsimons40). I also tweet about other cool things besides programming stuff, like Aaron Levie's socks (https://twitter.com/ericsimons40/status/448960308132859904).

With that said, welcome to the bleeding edge of mobile app development! You did an amazing job completing this course, and I'd love to hear your feedback on it. (https://twitter.com/intent/tweet?text=@EricSimons40%20I%20love%20you)

You finished!

Nicely done! Share your awesome new skills with the world:

(/author/eric)

Receive emails when Eric Simons (/author/eric) posts new content.

you@hello.com Subscribe

View our latest courses (/)