Serverless Single Page Web Apps, Part Four

CSCI 5828: Foundations of Software Engineering
Lecture 24 — 11/10/2016

Goals

- Cover Chapter 4 of Serverless Single Page Web Apps by Ben Rady
  - Present the issues related to managing user accounts in web apps
  - Introduce the notion of a federated identity service
    - Look at a specific example: AWS Cognito
  - Demonstrate how Cognito can be integrated into learnjs
    - Making use of Google+ as an identity provider

Identity in Web Applications (I)

- Many web applications require some way to identify the user that is accessing them
    - This allows them to
        - customize their display for each individual user, and
        - it allows them to maintain data for each individual user
- You often see identity manifested in web apps using the phrase "profile"

Identity in Web Applications (II)

- Typically, identity is managed using browser cookies
    - When you login to an application
        - your identity gets stored in a "cookie"
        - that bit of metadata gets associated with your web app's origin
            - i.e. if your web app sits at: http://example.com/
                - then your origin is "example.com"
        - the browser then automatically sends that cookie along with any HTTP request that is sent to its origin
- The problem is that this approach requires all of your app's web services to be from the same origin in order to get access to that cookie

Identity in Web Applications (III)

- If, however, you use an external web service to manage your user identities, the need to deal with "origin" goes away
  - If you receive security credentials from a third-party web service, and
  - keep them within your front-end web application
    - (i.e. running within a user's browser)
  - then you can use a wide array of services directly from the browser
- Furthermore, all the difficulties associated with managing identities get shifted to the developers who provide an identity web service
  - You get to benefit from their hard work and simply make use of their service!
- One such service is Amazon's Cognito.

Amazon's Cognito

- Cognito manages *identity* via *identity federation*
  - We can use *identities* from a variety of *identity providers*…
    - Facebook, Google, etc.
  - and link them to a single *identity record* created by Cognito
- An *identity record* is stored inside an *identity pool*
  - Users within a pool can be granted access to AWS via *policies* based on a number of criteria
- Once our user has added their identity to our application's pool, our app can then make authenticated requests on their behalf

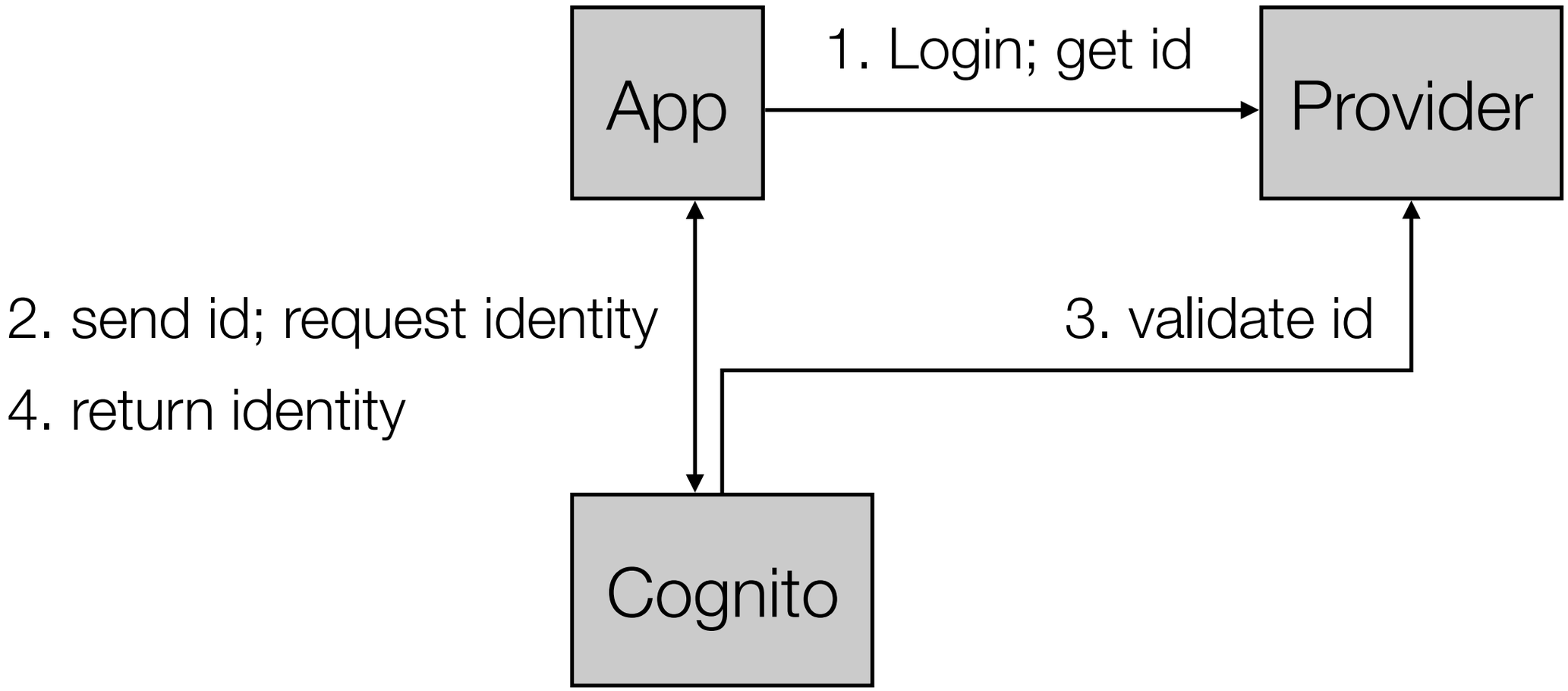
Implementation Concerns

- Normally, you would have to store user identity information in a database
    - You would have to worry about keeping that information secure
        - i.e. storing passwords that have been "properly salted and hashed"!
- Instead, once we have an identity token from Cognito, we simply store it alongside any data that we create in third-party web services
    - Then, when we retrieve our user's data from that service, we have the information we need to then make calls to other services
        - We'll see examples of that in future chapters
- With Cognito, we can avoid having to manage user passwords and instead focus on the features of our application

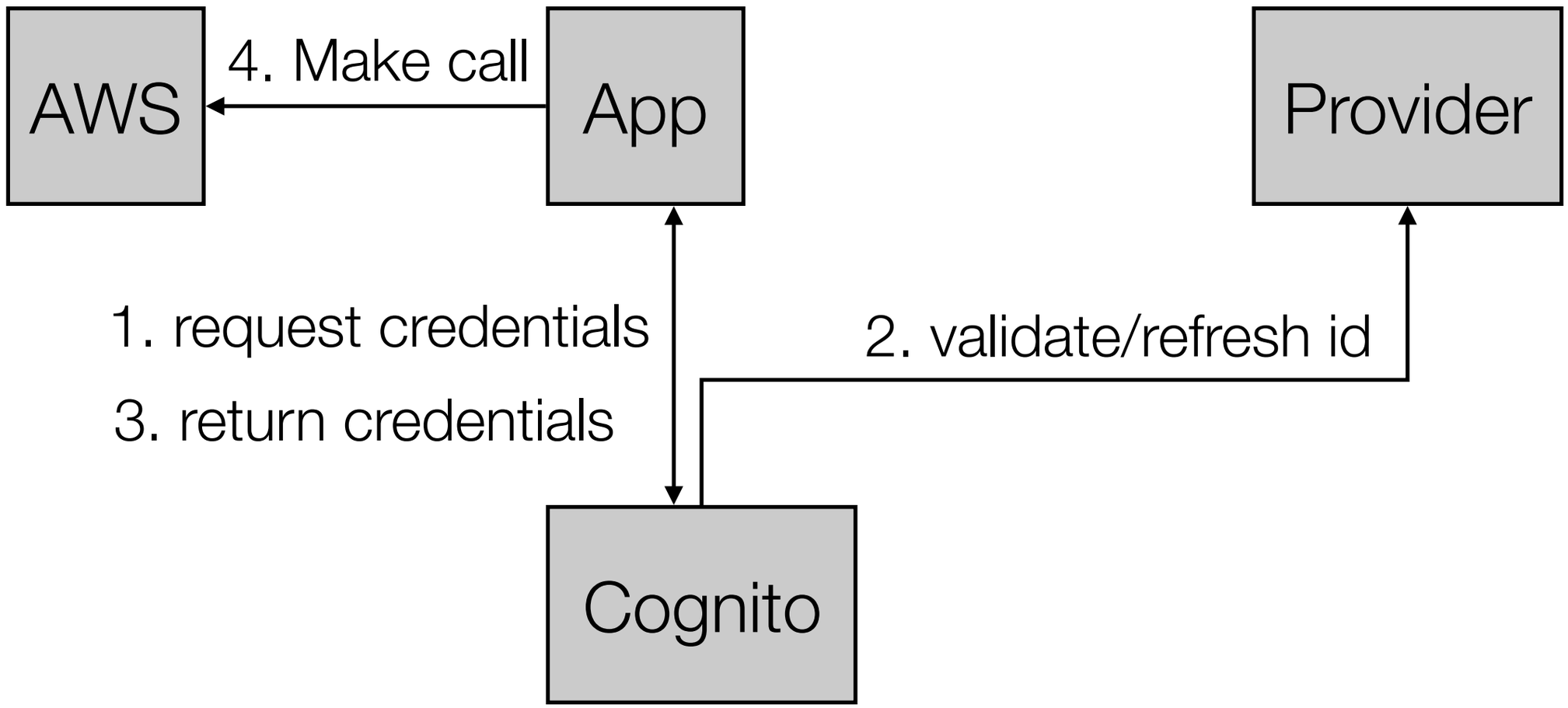
Using Cognito

- Our basic process will be the following
  - We first get a unique identifier from an identity provider
    - We will be using Google for this
  - Once we have that id, we can associate it with a Cognito identity
    - We can then use that identity to get the credentials we need to access AWS
- Working with other identity providers will be similar but each one has a different process for getting the unique id
  - This goes back to one of the limitations that we discussed in Lecture 20
    - Vendor Lock In

Workflows (I): Getting our Cognito Identity



Workflows (II): Getting our Cognito Identity



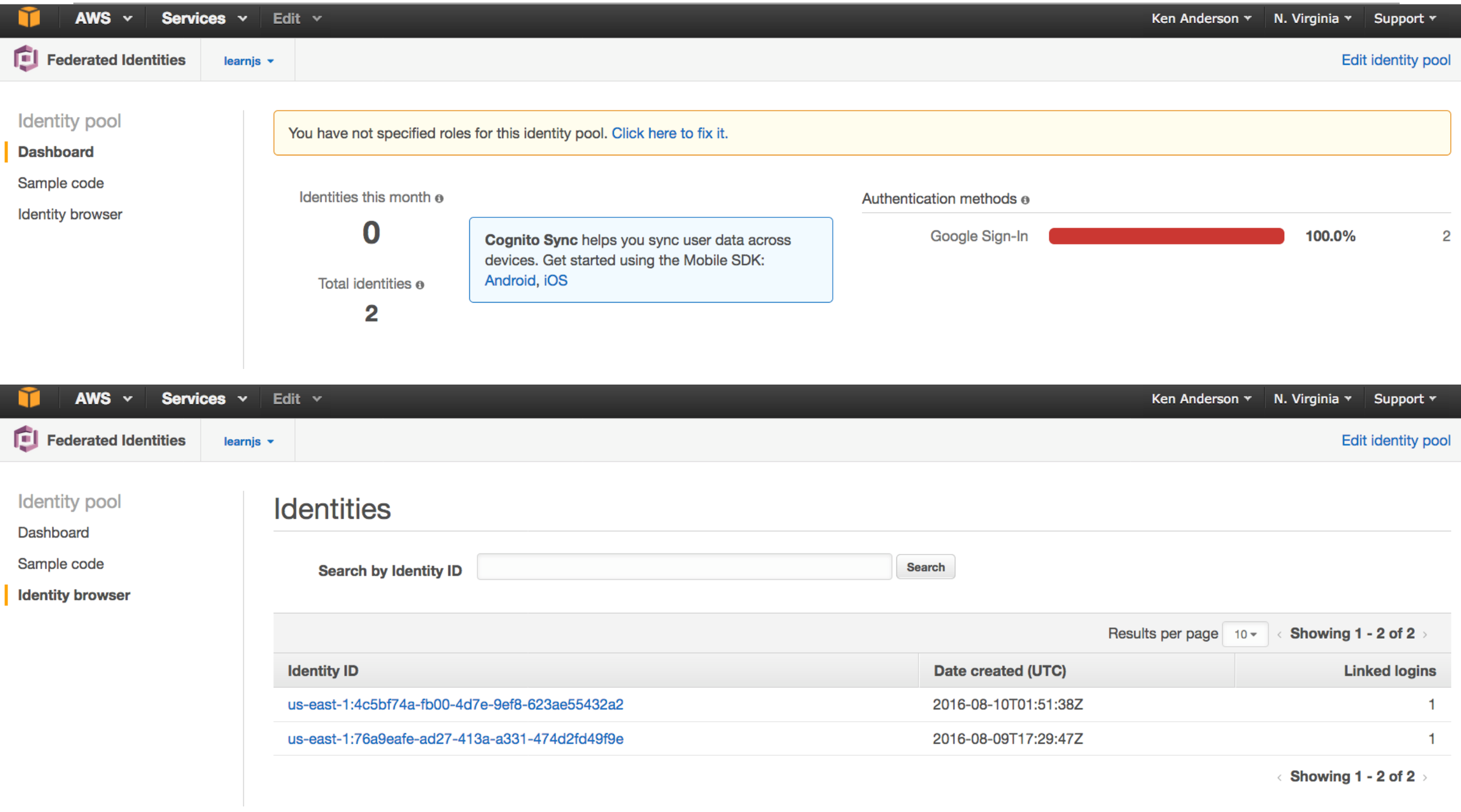
Using Google+ Sign In

- To use Google+ as an identity provider
  - You have to create a "project" on the Google Developers Console
- Once you have a project created, you can then enable the Google+ API
  - Then, you can select Credentials and click over to the OAuth consent screen. This screen will be used to represent your app.
    - Basically, the screen that will tell your user that "such and such an app is asking for you to sign in"
- See page 74 of the textbook for help with this step
  - You will need to make sure that the URLs entered into the app include at least two different domains: localhost and your AWS deployment domain
    - Look at the screenshot on page 74 for help!
- At the end of this process, we need a Google+ client id

Identity Pool

- The next step is to create a Cognito Identity Pool
    - It plays the role of a "users" database table in traditional web apps
- There is no limit to the number of users within an identity pool
    - and you can share identity pools across multiple web apps
        - which lets your users share data between them
- We take the Google client id from the previous slide and we plug it into the file at conf/cognito/identity_pools/learnjs/config.json
    - We then run the command
        - ./sspa create_pool conf/cognito/identity_pools/learnjs
- This creates a number of files that will allow our users to login and access other AWS services; see the book for details

AWS Cognito Screenshots



Getting a Google Identity

- There are a number of steps to configure our web app to make use of Google as an identity provider
    - We need
        - to load a new Javascript library
        - to list our client id in our page's metadata
        - to create a JavaScript function that handles a callback from Google
        - to create a div for Google's "connect" button
- Then, we'll be at a place where we can pass the identity we get from Google to AWS to add our user to our identity pool

Doing the Work (I): Add JavaScript Library

- We add this line to the head tag of index.html
  - <script src="https://apis.google.com/js/platform.js" async defer></script>

Doing the Work (II): Add Metadata

- Next we add a meta tag to index.html to identify our Google Client Id
  - <meta name="google-signin-client_id" content="<INSERT ID>" />
- This is the id that you got when following the steps on slide 11

Doing the Work (III): Define Callback Function

- Now we add a function that will handle the callback from Google

```
function googleSignIn() {
  console.log(arguments);
}
```

- Note: this function lives in the GLOBAL namespace
  - It does NOT go inside of our learnjs namespace
- For now, all this function does is print out its arguments. This lets us test that our connection to Google is up and running

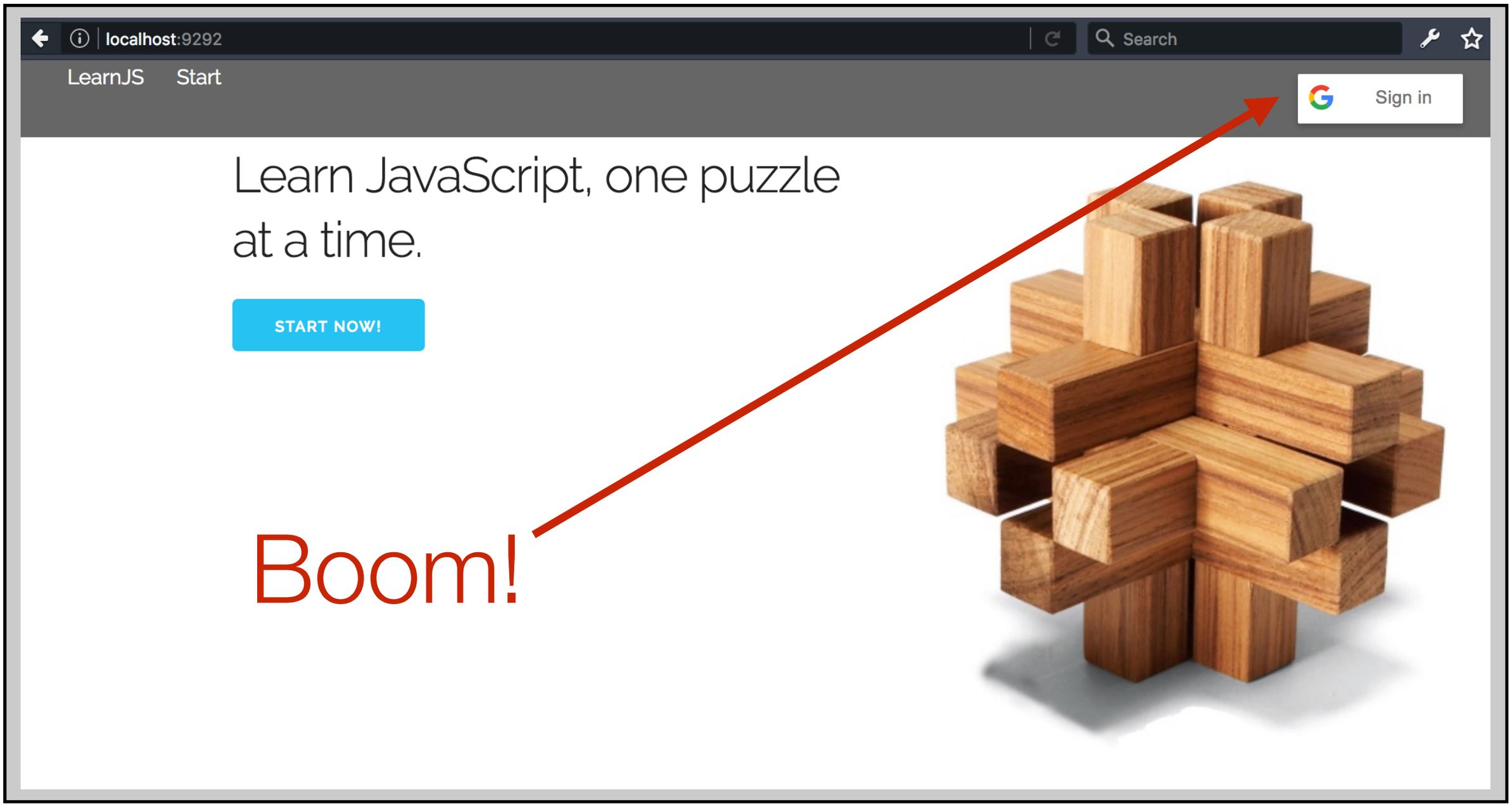
Doing the Work (IV): Add Google Connect Button

- Now we need to add a place for Google's connect button to appear

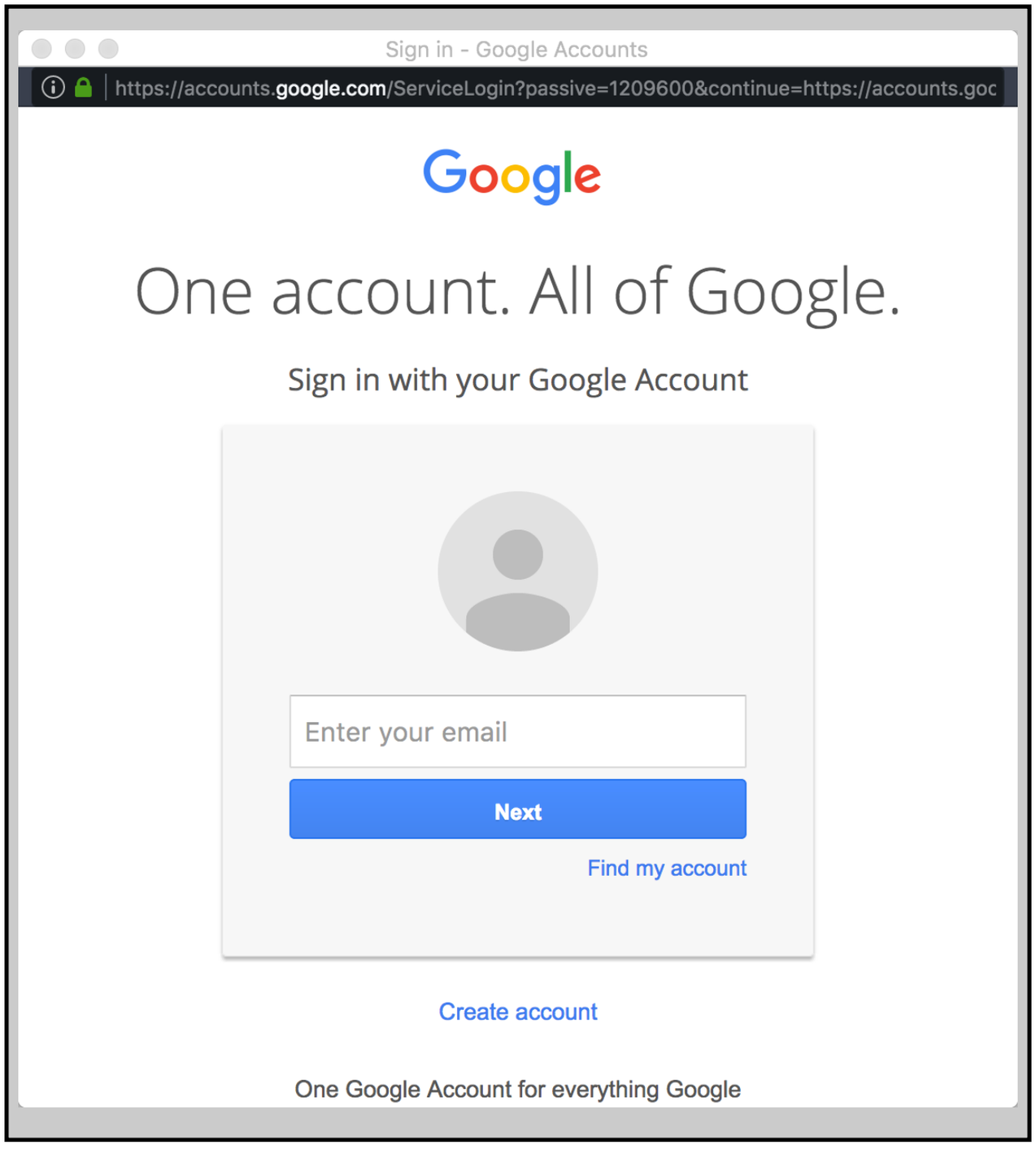
```
<div class='nav-container no-select fixed-top u-full-width'>
  <ul class='inline-list hover-links nav-list six columns'>
    <li><a class='text-lg' href="#">LearnJS</a></li>
    <li><a href="#problem-1">Start</a></li>
  </ul>
  <div class='four columns'>
    <span class='navbar-padding u-pull-right'>
      <span class="g-signin2" data-onsuccess="googleSignIn"></span>
    </span>
  </div>
```

- Here, we add a span with a class "g-signin2" that allows Google's Javascript library to find it and add its button
  - We then add some css to define what "navbar-padding" means
    - Note: I changed the book's CSS to read: padding: 10px 10px

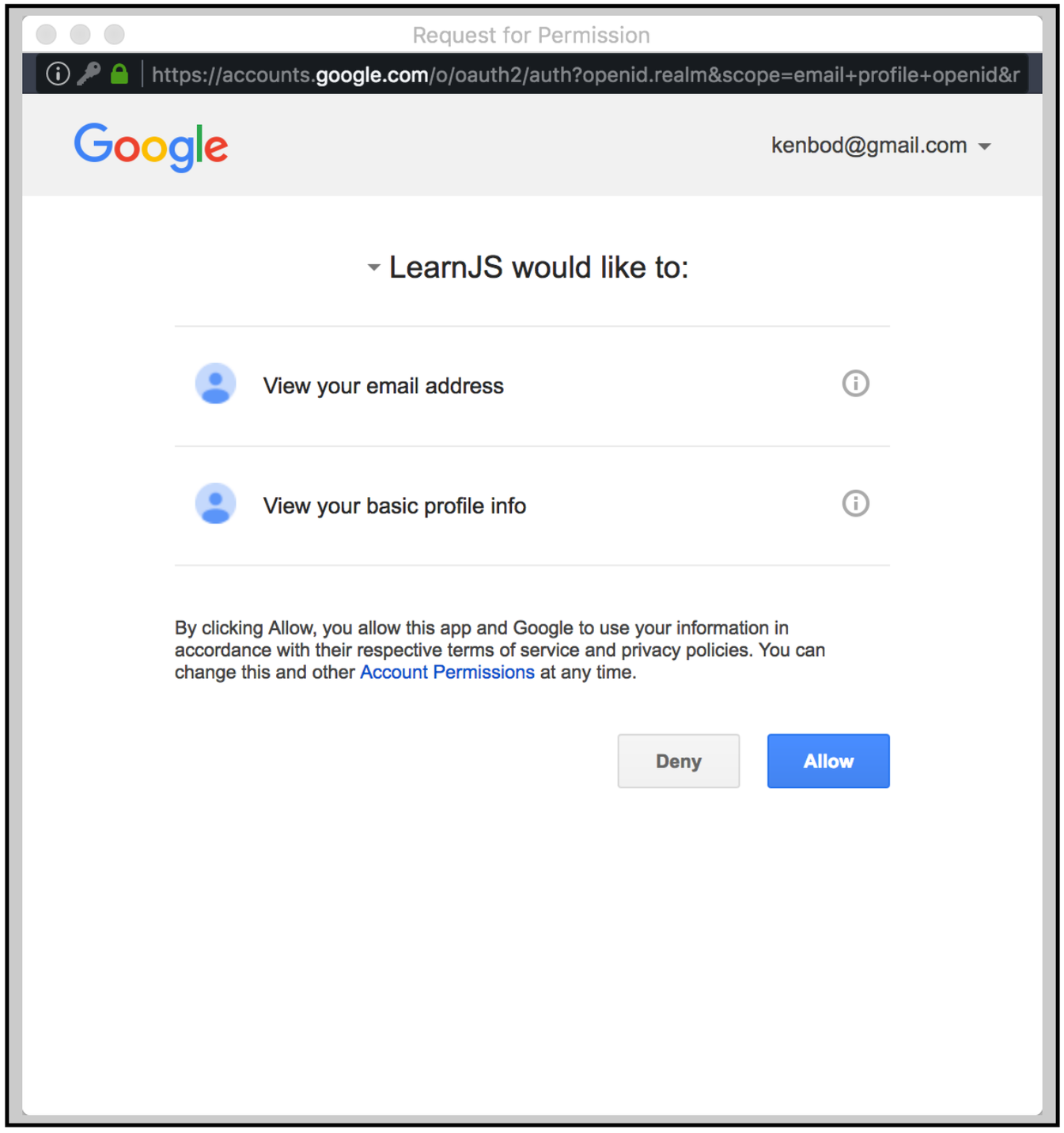
Reload the Browser and…



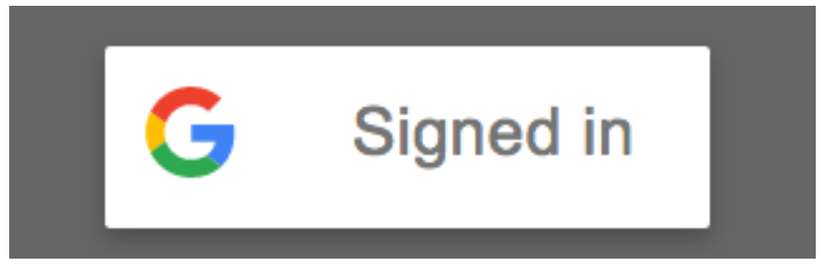
Push the magic button…



Enter your e-mail and password…



Click Allow…



- Two things happen
  - Our "sign in" button changes
  - And, we see output on the developer console
    - indicating that our callback function was, in fact, called!

- Note: we're not actually signed in since we didn't keep the token
  - Now, we need to add code that does something with the information that Google provides

First Step: Update our AWS configuration (I)

- Our prepared environment automatically adds the AWS JavaScript library to our application.
  - Now that we have a credential from Google, we need to update it's internal configuration information
    - it will then be in a state where it can make the appropriate calls to Cognito to get an identity that we can then use to access AWS services
- To do that, first, we add our pool id to our learnjs namespace

```
var learnjs = {
  poolId: '<INSERT POOL ID HERE>'
};
```

- Then, we need to modify our `googleSignIn()` function

First Step: Update our AWS configuration (II)

```
function googleSignIn(googleUser) {
  var id_token = googleUser.getAuthResponse().id_token;
  AWS.config.update({
    region: 'us-east-1',
    credentials: new AWS.CognitoIdentityCredentials({
      IdentityPoolId: learnjs.poolId,
      Logins: {
        'accounts.google.com': id_token }
      })
  });
}
```

- Here we get an identity token from Google Plus and then we update our local configuration information with a new set of Cognito credentials

Second Step: Handle Token Refresh (I)

- The token provided by Google has a one-hour lifetime
  - after that, it expires, and Cognito can't make use of it
- When we detect that it has expired, we need code that will call Google and get a new token.
  - We then have to update our configuration to use the new token
- One challenge with all of this is that these calls can take an indeterminate amount of time
  - If we discover that the credentials have expired when making a web service call (which we'll do in subsequent chapters), then we need a way to specify that our app should go update the token (however long that takes) and then complete the action that was in progress
    - How are we EVER going to chain all of these asynchronous requests together?

Promises to the rescue!

- The book makes use of promise objects returned by the Google API and jQuery deferred objects (which act like promises) to solve this problem!
- First, we create a function inside of googleSignIn called refresh()
  - It handles getting a new token from Google and then updating AWS

```
function refresh() {
  return gapi.auth2.getAuthInstance().signIn({
    prompt: 'login'
  }).then(function(userUpdate) {
    var creds = AWS.config.credentials;
    var newToken = userUpdate.getAuthResponse().id_token;
    creds.params.Logins['accounts.google.com'] = newToken;
    return learnjs.awsRefresh();
  });
}
```

Refreshing AWS

- To update our AWS credentials, we then use this function

```
learnjs.awsRefresh = function() {
  var deferred = new $.Deferred();
  AWS.config.credentials.refresh(function(err) {
    if (err) {
      deferred.reject(err);
    } else {
      deferred.resolve(AWS.config.credentials.identityId);
    }
  });
  return deferred.promise();
}
```

- Here, we create a promise and start a long-running function that will resolve when we get our updated credentials back from Cognito.

Making use of the credentials

- The last step is to configure our app to make use of the credentials
  - We'll create an identity object that is, itself, a promise
    - `learnjs.identity = new $.Deferred();`
  - Then, we update the googleSignIn function to resolve this promise and supply a value that contains everything we need to make use AWS

```
learnjs.awsRefresh().then(function(id) {
  learnjs.identity.resolve({
    id: id,
    email: googleUser.getBasicProfile().getEmail(),
    refresh: refresh
  });
});
```

Understanding the Chain

- It is important that you understand the chain of promise objects created to handle this refresh process
  - The Google signin method is called by Google when the connect button is picked and the user clicks approve
  - googleSignIn() calls awsRefresh() and registers a then() callback on it.
    - awsRefresh creates a promise object, invokes a long running operation on it (updating our AWS credentials), and then returns the promise
    - when that resolves, the then() handler fires and that resolves our identity object that will be used soon to list the e-mail address associated with the connected account
- Workflow: user clicks button => awsRefresh() => then() => learnjs.identity
- token goes invalid => refresh() => awsRefresh() => then() => learnjs.identity

Seeing the Chain in Action (I)

- We've set up the promise chain to ensure that our identity object has the connected user's e-mail address stored in a property
    - Let's create a view that displays that property when it is available
- First, we need to create a view function to display the e-mail address

```
learnjs.profileView = function() {
  var view = learnjs.template('profile-view');
  learnjs.identity.done(function(identity) {
    view.find('.email').text(identity.email);
  });
  return view;
}
```

Seeing the Chain in Action (II)

- Next, we create the profile-view template

```
<div class='profile-view'>
  <h3>Your Profile</h3>
  <div class='email'></div>
</div>
```

- and, add a route that maps a url to that view
  - `'#profile': learnjs.profileView,`
- and update our appOnReady() function to register a done handler on the identity object
  - `learnjs.identity.done(learnjs.addProfileLink);`

Seeing the Chain in Action (III)

- Finally, implement the addProfileLink method

```
learnjs.addProfileLink = function(profile) {
  var link = learnjs.template('profile-link');
  link.find('a').text(profile.email);
  $('.signin-bar').prepend(link);
}
```

- and add a new template called profile-link that will be used to add a link next to the sign-in button which takes you to the profile view.
- The result: the e-mail address of the associated identity is displayed, both in the navbar and the profile view
  - Currently, there is no way to sign out

Summary

- In this chapter, we have touched on a number of topics
  - federated identity services and Cognito
    - multiple identity providers
  - Workflows for handling identity acquisition
    - Use of promises to ensure that profile info is not allowed to populate a view until we're sure that identity acquisition is over (and did succeed)

- Next Time: Storing Data in Dynamo DB