Serverless Single Page Web Apps, Part Three

CSCI 5828: Foundations of Software Engineering
Lecture 23 — 11/08/2016

Goals

- Cover Chapter 3 of Serverless Single Page Web Apps by Ben Rady
    - Creating a well-rounded single page web app
        - Views
        - Data Model
        - Data Binding
        - Navigation
        - Some bells and whistles: animation
- These items are elements found in all single page web applications

LearnJS

- Reminder: our example application is called LearnJS
    - It's primary view is going to be a "problem view" that presents a JavaScript puzzle to a user; the puzzle will have a blank "placeholder"
        - The user needs to enter text that when placed in the placeholder makes the JavaScript puzzle return a "truthy" value.
    - Example
        - function problem() { return 42 === 6 * __; }
    - If we enter "7" as our answer, this program will return true
- We need to build up a data model that holds these problems and a view to display them; we will start by getting our views a bit more organized

Extracting Views

- We currently have view functions that generate HTML markup using jQuery and then we attach that markup dynamically to places in the DOM
- For the problem view, we're going to modify this set-up to also make use of a generic HTML template
    - First, we need a place for our templates to live
- "It's div's all the way down"
    - We'll add a div to our index.html page that is tagged with the class "templates" and then we'll add a template for our problem view there
        - That template will have "problem-view" div and an H3 heading for the problem's title

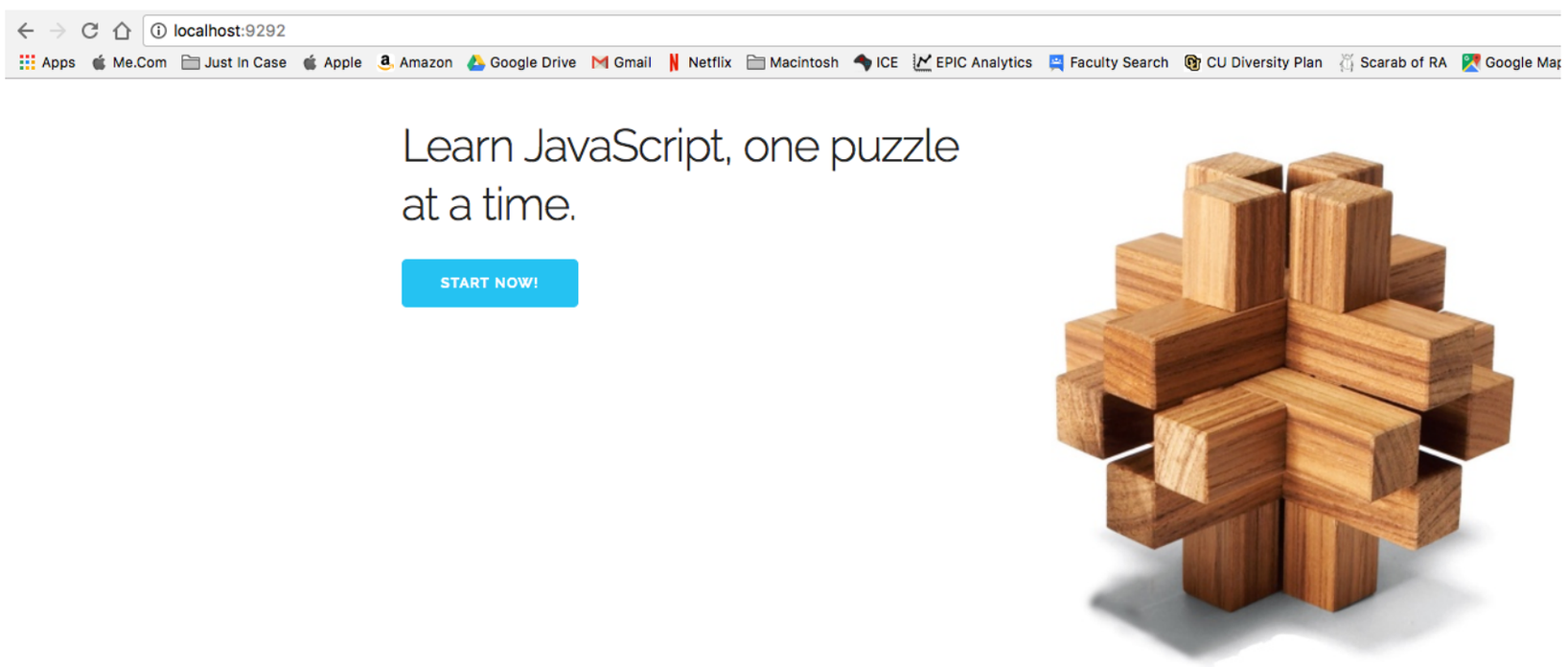
Updating index.html

```
            <div class='one-half column'>
              <h3>Learn JavaScript, one puzzle at
              <a href='#problem-1' class='button
            </div>
            <div class='one-half column'>
              <img src='/images/HeroImage.jpg'/>
            </div>
          </div>
        </div>
      </div>
    </div>
    <div class='templates'>
      <div class='problem-view'>
        <h3 class='title'></h3>
      </div>
    </div>
    <script type='text/javascript'>
      $(window).ready(learnjs.appOnReady);
    </script>
  </body>
</html>
```

Here we add the `templates` div to our document; we then add the template for our `problem-view` within it

Can the Templates be seen? (I)

- For the current template, we are safe.
  - It defines some HTML structures but no visible text
- Watch what happens if we put placeholder text in the template
  - <h3 class='title'>Put Title Here</h3>



Can the Templates be seen? (II)

- So, yes, if a template contains user-visible text
  - that text can be seen when the page is loaded
- To fix this, we will use CSS to tell the browser not to display anything inside of the templates div
  - In index.html, we add this line to the `style` tag
    - .templates { display: none; }
  - This CSS directive selects on all items with a class of "templates" and sets their display attribute to "none". This applies to the element and all of its child elements
- With this added, our placeholder text will no longer appear
  - Be sure to remove that placeholder text however.
- `git add .; git commit -m "Added templates div"` <= commit early, commit often

Template?

- We call the child elements of `templates` div a "template" because:
  - when we need to create a new problem view
    - we're going to find this template in the DOM
    - CLONE it (using jQuery)
      - and then create our target view
- The code that will do this is the view function we created in Lecture 21 called `problemView()`

Changing `problemView()` (I)

- problemView() currently looks like this

```
learnjs.problemView = function(number) {
  var title = 'Problem #' + number + ' Coming soon!';
  return $('<div class="problem-view">').text(title);
}
```

- Here, we are creating a div using jQuery and inserting our title into it
- Now, we need to find our template, copy it, and update the existing title element; then we can return it and our existing code will display it for us

```
learnjs.problemView = function(number) {
  var view = $('.templates .problem-view').clone();
  view.find('.title').text('Problem #' + number);
  return view;
}
```

Changing `problemView()` (II)

- The result?

- `git add .; git commit -m "problemView uses templates"`

- Now, we need to add a data model to our app, so we can add a bunch of new problems to our app.

Data Model

- Our data model is going to be a simple data structure
    - an array of JavaScript objects
- Each object will have attributes that store the code and description of each problem that will eventually be displayed in our problem view
    - This data structure will be called `problems` and is defined within our `learnjs` namespace and stored in app.js
- The book starts us off with two problems
    - I added three more (see next slide)

```
learnjs.problems = [
  {
    description: "What is truth?",
    code: "function problem() { return __; }"
  },
  {
    description: "Simple Math",
    code: "function problem() { return 42 === 6 * __; }"
  },
  {
    description: "Start of an Array",
    code: "function problem() { list = [42, 23, 10]; return __ === 42; }"
  },
  {
    description: "Accessing Attributues",
    code: 'function problem() { p = {name: "Ken"}; return __ == "Ken"; }'
  },
  {
    description: "Increment Variable",
    code: 'function problem() { a = 41; return (++a == __);}'
  }
];
```

Note: I screwed up on this text. The last three problems need to define their variables. They should say "var list", "var p", and "var a".

Data Binding (I)

- The code shown for inserting a title on slide 9 works for simple templates
  - but it does not scale well
- If our templates and data model become more complex
  - we want a way to ***bind*** the elements in our data model to elements in our template
- HTML 5 provides a way to do simple one-way data binding
  - via HTML data attributes
    - https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Using_data_attributes

Data Binding (II)

- To take advantage of this standard, we need to add a few things to our template; in particular, attributes that start with "data-" and (in our case) end with "name"

```
<div class='templates'>
  <div class='problem-view'>
    <h3 class='title'></h3>
    <p data-name='description'></p>
    <pre><code data-name='code'></code></pre>
  </div>
</div>
```

- We now have a p tag that has a data-name attribute that contains the value "description" and a code tag that has a data-name attribute of "code"
  - We can use these attributes to populate our template

Data Binding (III)

- To take advantage of this standard, we need to add a few things to our template; in particular, attributes that start with "data-" and (in our case) end with "name"

```
<div class='templates'>
  <div class='problem-view'>
    <h3 class='title'></h3>
    <p data-name='description'></p>
    <pre><code data-name='code'></code></pre>
  </div>
</div>
```

- We now have a `p` tag that has a data-name attribute that contains the value "description" and a `code` tag that has a data-name attribute of "code"
  - We can use these attributes to populate our template

Data Binding (IV)

- The book uses this function to perform data binding

```
28 learnjs.applyObject = function(obj, elem) {
29   for (var key in obj) {
30     elem.find('[data-name="' + key + '"]').text(obj[key]);
31   }
32 };
```

- The first parameter is an instance of the problem objects from our data model
  - The loop will loop twice for each object since it has only two properties
    - Each time, we will search for an element in "elem" that has a data-name attribute matching the key.
    - We'll then set that element's text object to the value contained in our problem object

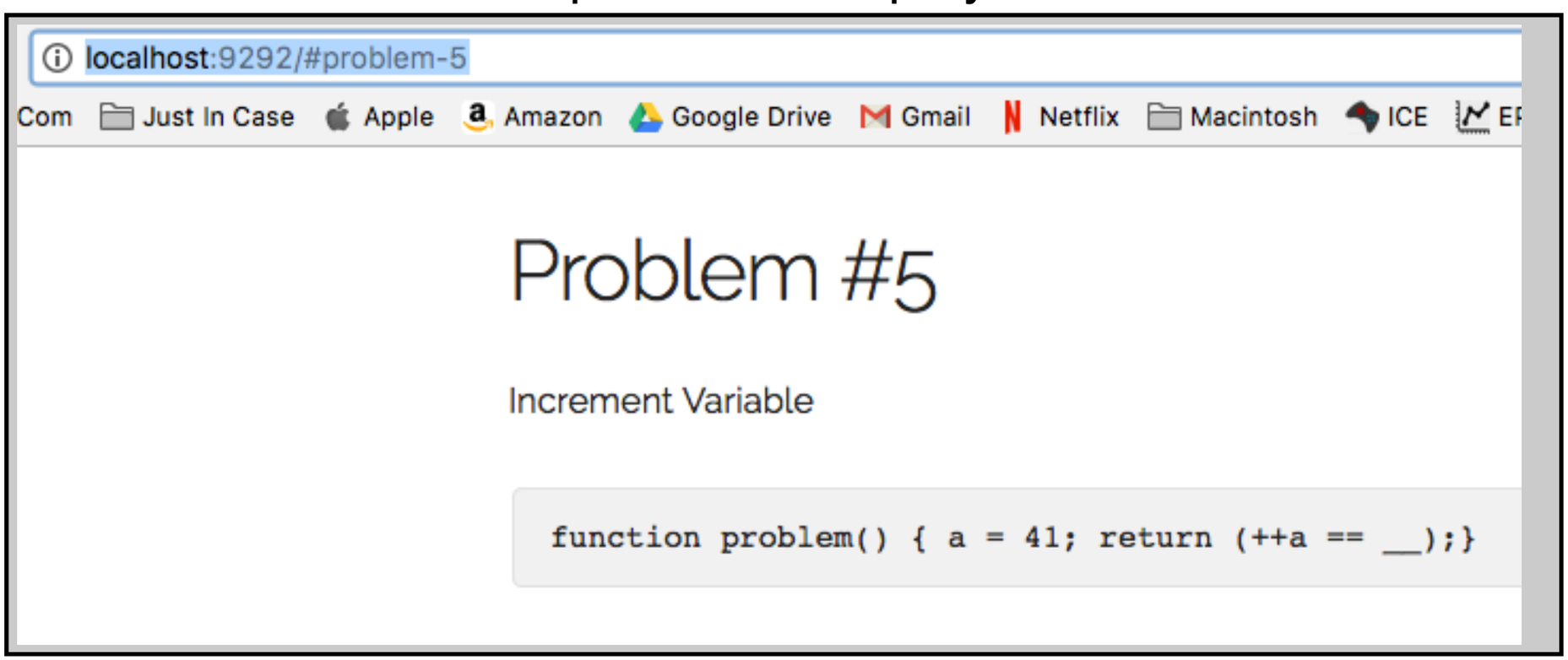
Data Binding (V)

- We just need to update our `problemView()` function to make use of the new `applyObject()` function
- We do that by adding this line after the line that sets the title
  - `learnjs.applyObject(learnjs.problems[number - 1], view);`
  - (for this to work, we also add a line to the function to parse the string passed in for the problem number and convert it to an integer

```
34 learnjs.problemView = function(data) {
35   var number = parseInt(data, 10);
36   var view = $('.templates .problem-view').clone();
37   view.find('.title').text('Problem #' + number);
38   learnjs.applyObject(learnjs.problems[number -1], view);
39   return view;
40 }
```

Data Binding: The Results?

- We can now manually enter the URLs
  - http://localhost:9292/#problem-1, …
  - http://localhost:9292/#problem-5
- And see each of our problems displayed




```
git add .; git commit -m "Data Binding Added"
```

Ready for Input (I)

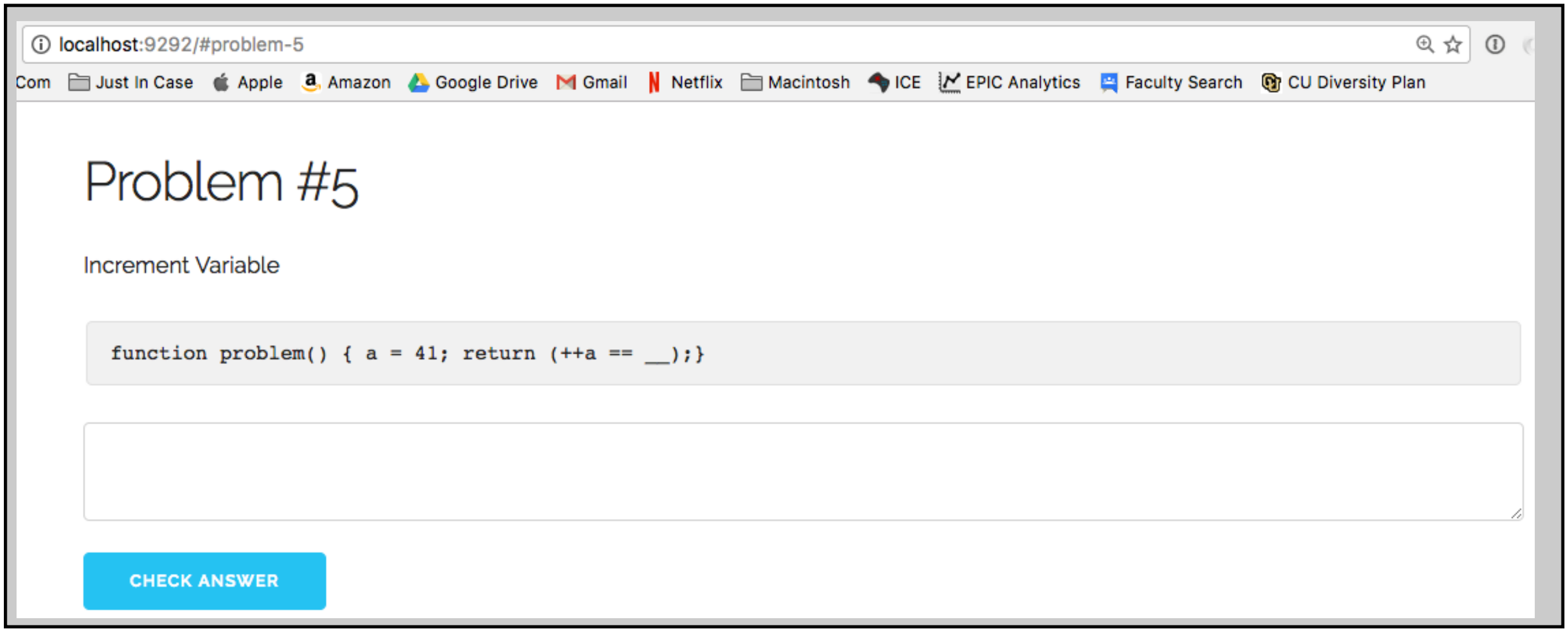
- Now, we need to configure our problem view to allow a user to enter a solution to the problem
    - We'll use a form for that; we'll modify our template like this:

```
<div class='problem-view'>
  <h3 class='title'></h3>
  <p data-name='description'></p>
  <pre><code data-name='code'></code></pre>
  <form>
    <textarea class='u-full-width answer'></textarea>
    <div>
      <button class='button-primary check-btn'>Check Answer</button>
      <p class='result'></p>
    </div>
  </form>
</div>
```

    - Some of these CSS classes are for our app; the rest come from Skeleton

Ready for Input (II)

The result?



But, right now, the button doesn't do anything

Handling the Form (I)

- We will be modifying our problemView() function to add behavior to our problem view
    - We will attach a function to the "submit" button that returns "false" to tell the browser not to do anything when the form is submitted
- We will handle everything within our JavaScript code

Handling the Form (II)

- We will start by writing the code that checks the user's solution
  - They enter text into the "answer" text area.
  - We substitute that text into the problem string and run it through eval()
    - There is a potential security risk with eval() that we will ignore for now

```
function checkAnswer() {
  var answer = view.find('.answer').val();
  var test = problem.code.replace('__', answer) + '; problem();';
  return eval(test);
}
```

  - As you can see, we substitute the user's answer for the "__" in the problem and then we invoke the problem function. This string gets executed by JavaScript's `eval()` function; will return `true` or `false`

Handling the Form (III)

- Now, we need code, that handles the form submission
  - It's really simple

```
function handleSubmit() {
  if (checkAnswer()) {
    result.text('Correct!');
  } else {
    result.text('Incorrect!');
  }
}
```

- If our previous checkAnswer() code returns true, then set the result portion of the template to "Correct" otherwise "Incorrect".

```
git add .; git commit -m "User Input Handled"
```

Need for Animation (I)

- Our code works
  - but…
    - if you enter two wrong answers in a row, the browser doesn't seem to update; that's because it's "changing" the text "Incorrect" to "Incorrect" and the user doesn't see a visual update
      - as a result, it feels like the app is broken
- To fix, this, we're going to change our code to animate the result message
  - using jQuery, of course

Need for Animation (II)

- We'll use this code to do the animation

```
learnjs.flashElement = function(elem, content) {
  elem.fadeOut('fast', function() {
    elem.html(content);
    elem.fadeIn();
  });
}
```

- This code calls jQuery's fadeOut command and passes in a callback
  - When the element has disappeared, the callback is invoked
    - It then updates the element with the new content and then makes it fade back in

```
git add .; git commit -m "Animation Added"
```

Navigating through the Problems

- We now need a way to allow our user to move from problem to problem
    - (without having to type the problem urls themselves!)
- We'll do this by updating our template to add a link to the next problem when the user gets a problem correct
    - If the problem is incorrect, we'll keep them on the same page.
    - If they solve the last problem, we'll provide a link back to the landing page
- Since the text for a correct solution will change with context
    - we will create a template for it and then populate the template as needed for the current context

Handling a Correct Solution (I)

- First, we'll add our template to index.html

```
<div class='correct-flash'>
 <span>Correct!</span> <a>Next Problem</a>
</div>
```

- We'll create a function that makes it easier to retrieve templates

```
learnjs.template = function(name) {
 return $('.templates .' + name).clone();
}
```

- Now, we're ready to create a function that uses this template to create the proper link based on which question the user has answered

Handling a Correct Solution (II)

- Here's the function that uses the template to create our "correct" response

```
45 learnjs.buildCorrectFlash = function(number) {
46   var correctFlash = learnjs.template('correct-flash');
47   var link = correctFlash.find('a');
48   if (number < learnjs.problems.length) {
49     link.attr('href', '#problem-' + (number + 1));
50   } else {
51     link.attr('href', '');
52     link.text("You're Finished!");
53   }
54   return correctFlash;
55 }
```

- This function gets called by our handleSubmit() function. We can now navigate through all the problems in our quiz!

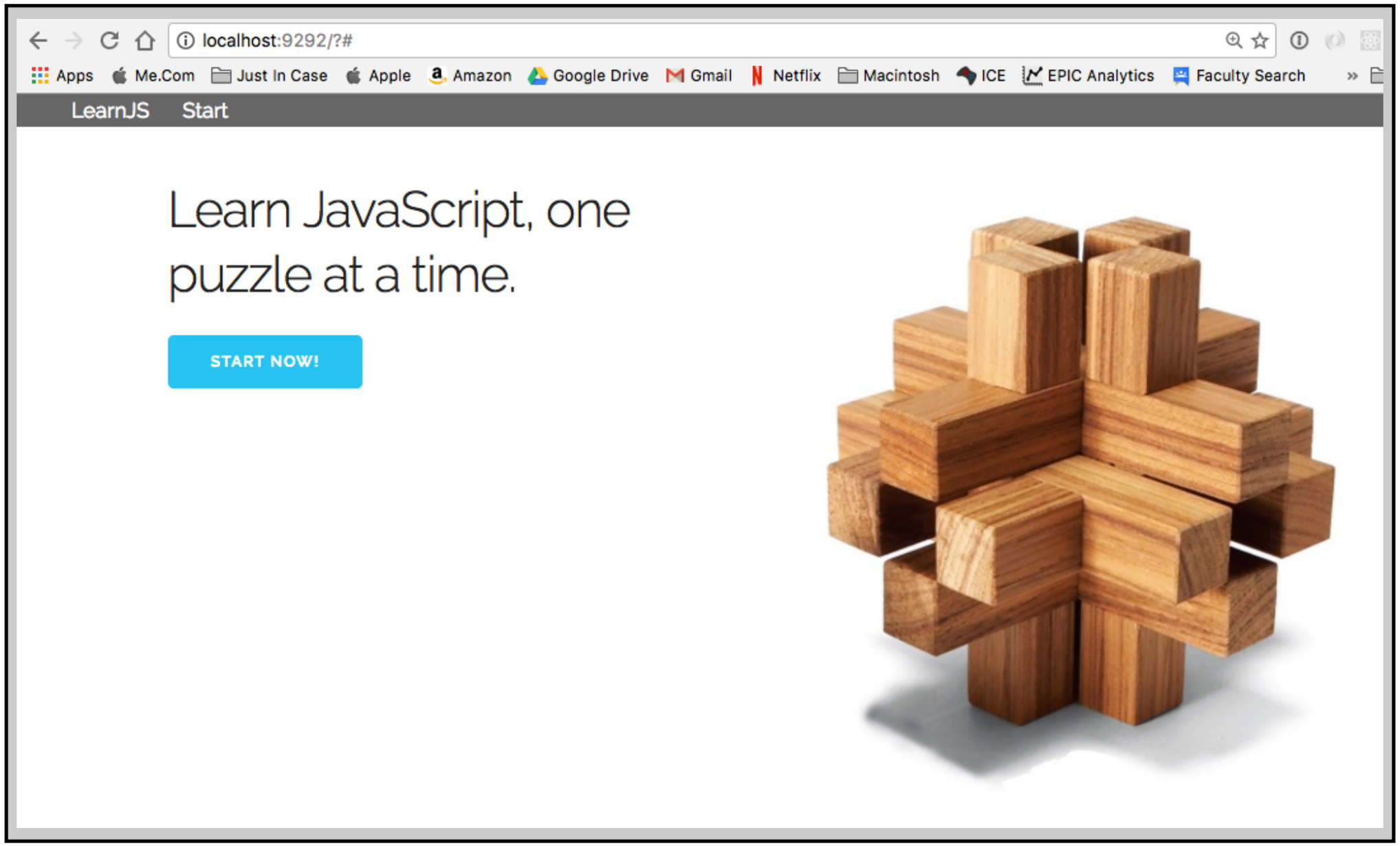
```
git add .; git commit -m "Quiz Navigation Complete"
```

Adding an Application Shell

- I'm going to skip this section of the book
  - They add a "shell" to the application which is defined as
    - those elements (like navbars) that appear in all views
- To do this requires changes that mostly involve HTML and CSS
  - The result is shown on the next slide

```
git add .; git commit -m "Application Shell Added"
```

Our App: Now with NavBar



Last Change: Custom Events

- Currently, it is easy for our view functions to reach out and access our application
    - the learnjs namespace makes that very straightforward
- But, there is no easy way for our application to trigger behavior inside our views
    - Our views get created by view functions and then attached into the DOM
        - our application doesn't have a handle that allows it to access the view (or its functions) once they have been inserted into the DOM
- The solution is to make use of HTML5's standard event mechanism
    - We can have views register for events and then trigger those events at the application level when needed

Implementing a Skip Button

- To demonstrate why this type of functionality is useful, we'll have our problem view implement a "skip" button
  - On every problem, except for the last, we'll add a "Skip" button to our navigation bar
    - If the user doesn't want to work on a problem, they can skip it
- The problem here is that we need to make sure that every time we go to a new view we check to see if a skip button is present
  - Rather than have the application do that, we'll let our views handle that
  - Before a view goes away, we'll trigger a "removingView" event
    - If a view needs to clean up after itself, it will register for this event and do the cleanup in the corresponding event handler

Implementing Events (I)

- First, we create a function that lets us trigger an event

```
45 learnjs.triggerEvent = function(name, args) {
46   $('.view-container>*').trigger(name, args);
47 }
```

  - The syntax ".view-container>*" means trigger the event on all children of the "view-container" element.

- Now, we add a call to this event in `showView()` to trigger the removingView event

  - We do this right before the call to `empty()` in `showView()`.

```
if (viewFn) {
  learnjs.triggerEvent('removingView', []);
  $('.view-container').empty().append(viewFn(parts[1]));
}
```

Implementing Events (II)

- Now, we need a new template, that will help us create a Skip button
  - `<li class='skip-btn'><a>Skip This Problem</a></li>`
- Finally, we add code to problemView() to create the skip button and register the appropriate event handlers (using jQuery)

```
if (number < learnjs.problems.length) {
  var buttonItem = learnjs.template('skip-btn');
  buttonItem.find('a').attr('href', '#problem-' + (number + 1));
  $('.nav-list').append(buttonItem);
  view.bind('removingView', function() {
    buttonItem.remove();
  });
}
```

- Note: `view.bind()` is used to register an event handler for our custom event

Summary

- In this chapter, we have touched on a number of topics
  - extracting markup into HTML templates
  - adding a data model to the application
  - adding data binding between model and view
  - adding navigation
  - adding an application shell
  - adding custom events and showing how they can be used
- We now have a solid prototype that demonstrates most of the features of a typical serverless single page web app
- Next Up: Handling user identities with Amazon's Cognito service