6.170 Recitation 8
Single Page Applications

Motivation for Single-page apps

Recall the Fritter app that was built in P2. In that application, we wrote our server code so that, whenever the client (browser) made a request to the server, the server would do some processing before sending back an HTML page. The browser would then render this HTML page, along with attached stylesheets and Javascript, to the screen. We then talked about structuring our application in the MVC style to achieve separation of concerns between data, logic, and presentation.

Last week, we talked about server-side APIs. Some of the benefits of an API include:

1. Further decoupling of server and client. We no longer make the assumption that the client is a browser capable of rendering HTML, but rather just some abstract consumer of data.
2. Decoupling between data and presentation. When we return HTML from a server, the data that we're interested in is embedded within its presentation, the HTML document structure. We want to treat the server simply as a source of data, and have the client deal with the issues concerning presentation.
3. One concrete benefit we get from the above is compatibility across many platforms. Web apps, mobile web, mobile, third-party apps, etc. are all able to build their applications on top of a common data interface.
4. Another concrete benefit is the *amount* of data transferred. We don't need to transfer lots of HTML which surrounds our data every round-trip. Many actions that a user does on a page (e.g. a "Like" on Facebook, an "upvote" on Reddit) require *very little data transfer* and *very little change to the UI*.

In addition, one final benefit we achieve with web apps in particular is the following: page refreshes are very costly operations -- they are "synchronous" in the sense that the user cannot do anything while the page refresh is happening. In contrast, AJAX calls allow us to handle user actions *asynchronously*, so that the rest of the page is still responsive, and only the necessary data and DOM elements are modified in response to user actions.

Today we will go into detail on how exactly to set up a single-page web app that utilizes a web API as its data source.

Review: AJAX Calls

AJAX calls allow us to make calls from the browser to the server without refreshing the page. The call to the server is done *asynchronously*. As with all asynchronous Javascript, the function

making the AJAX call returns immediately, and we provide a callback to run when the server's response comes back.

Using jQuery, the syntax for an AJAX call is:

```
$.ajax({
      url: “/freets”,
      // can also be “GET”, modern browsers support “PUT” and “DELETE”
      type: “POST”,
      data: { content: “hello!” }
}).done(function(response) {
      // do things with the response
      // if your server specifies JSON in the response,
      // the response parameter will automatically be a Javascript object
}).fail(function(err) {
      // the important thing here is err.responseText
});
```

jQuery also gives us `$.get(...)` and `$.post(...)` aliases for the above.

Client-side Templating

When the AJAX call returns, we usually want to do something to update the state of the UI to reflect either the updated server state or to display the data we just requested.

One thing we can do is to update the DOM using jQuery, as you have all done in P1. For example, in order to “favorite” a freet, we might do something like:

```
$.ajax({
      url: “/freets/3/favorites”,
      type: “POST”
}).done(function(response) {
      $(‘#freet-id’).addClass(‘favorite);
});
```

This scheme works well for *small* changes to the DOM. But sometimes you may find yourself having to insert entire sections. For example, to create a freet, one might do:

```
$.ajax({
      url: “/freets”,
      type: “POST”,
```

```
      data: { content: content }
}).done(function(response) {
      var newfreet = ‘<div class=”freet”><span class=”author”>’ +
response.author + ‘</span><p class=”freet-content”>’ + response.content +
‘</p></div>’;
      $(‘#freet-list’).append(newfreet);
});
```

This is ugly, error-prone, and most importantly, couples presentation (views) with application logic and data!

One of the good things we did back when we kept views on the server is that, at least on the server, we decoupled data from presentation. We had some object or array with data, and called `res.render(‘view’, data)` to insert the data into the view. It seems like we took a step backwards.

We want to be able to emulate `res.render(...)` on the client. **In particular, what we want to avoid is constructing long strings of HTML in our Javascript!** (Think back to the P1 days of when we talked about separating HTML from Javascript in the context of event handlers).

Another way to do this with jQuery, more cleanly:

```
var renderFreet = function(author, content) {
      var authorElt = $(‘<span />’).addClass(‘author’).text(author);
      var contentElt = $(‘<p />’).addClass(‘freet-content’).text(content);
      return $(‘<div />’)
            .addClass(‘freet’).append(authorElt).append(contentElt);
};
```

In the AJAX call:

```
...
.done(function(response) {
      $(‘#freet-list’).append(renderFreet(response.author, response.content));
});
```

This is better -- we’ve extracted a function which creates our view elements, and have less view code in our application logic. But as elements get larger and layers of nesting get more complex, this scheme quickly becomes infeasible.

A third solution -- if you insist on using just jQuery. Create a bunch of hidden elements on the page, and when you need one of them, use jQuery’s `.clone()` method and populate its contents. For example, in our skeleton HTML, we might have the following:

```
<div class=”freet hidden” id=”freet-template”>
      <span class=”author”></span>
      <p class=”freet-content”></p>
</div>

var renderFreet = function(author, content) {
      var freetElt = $(‘#freet-template’).clone().removeClass(‘hidden’);
      freetElt.find(‘.author’).text(author);
      freetElt.find(‘.freet-content’).text(content);
      return freetElt;
};
```

In the AJAX call:

```
…
.done(function(response) {
      $(‘#freet-list’).append(renderFreet(response.author, response.content);
});
```

In a sense, what you’re doing here is creating an instance of the template and populating its fields. Why not just let a templating engine do that for you? … Handlebars.js -- a templating framework very similar to EJS, with slightly more convenient syntax, better documentation, and easier to call from jQuery.

To use Handlebars on your machine, on your terminal:

1. `$ npm install -g handlebars`
   (may need sudo on macs)

2. Write your templates in a templates/ directory. See the templates/ directory in the sample application. Each template must be named with a .handlebars extension.

3. Compile the templates into a Javascript file. You will need to do this every time a template is updated:
   `$ handlebars templates/ > public/javascripts/templates.js`

4. Download the Handlebars runtime and include it in your scripts in the HTML file.

5. When you need to load the template in your Javascript, call
   `Handlebars.templates[‘templateName’](data)`.

   `Handlebars.templates[‘templateName’]` gives you a *function*, which you can pass in an object (data) to obtain an HTML string.

Generally, you will insert the HTML into the DOM; e.g.
`$('#main-container').html(Handlebars.templates['freetTemp'](freets));`

Handlebars Syntax

Like EJS, Handlebars looks very much like HTML. You have an HTML structure, and fill in "holes" with the data that is passed in. All Handlebars operations and placeholders are surrounded in double braces, `{{ like_this }}`.

One difference between EJS and Handlebars is that you cannot write arbitrary Javascript in Handlebars. Handlebars only supports a small set of conditional and looping statements, namely `{{#if}}`, `{{#unless}}`, and `{{#each}}`. This helps enforce the separation between logic and presentation.

Finally, to include a template in another, insert the template like this: `{{ > templateName}}`

For example, the Handlebars template for displaying a Freet might be:

`freet.handlebars:`

```
<div class="freet">
      {{#if favorite}}
            <span class="favorite">Favorite!</span> 
      {{/if}}
      <span>Posted by {{author}} on {{createdTime}}</span>
      <p>{{contents}}</p>
</div>
```

We will get the HTML when we call `Handlebars.templates['freet'](freetData)`, where `freetData` is an object of the form:

```
{
      favorite: true,
      author: "Ben Bitdiddle",
      createdTime: "October 20, 2014 4:30PM",
      contents: "hello world!"
}
```

Note that we can simply reference the keys of the object that is passed in.

Next, if we want to show all Freets, we can use the template we just created for each freet as a *partial*.

In Javascript, do:
`Handlebars.registerPartial('freet', Handlebars.templates['freet']);`

Then, we can iterate through an array of freets:

```
{{#each freets}}
      {{ > freet }}
{{/each}}
```

A brief note: there are much bigger and more complex frameworks that enforce some sort of MVC layout on the client. You might have heard of frameworks like Backbone or Angular. Those frameworks certainly give your client-side code more structure than what we'll see here at the cost of some complexity and more "black magic". Feel free to use those frameworks; most importantly, keep your code clean and enforce good modularity.

Hands-on Example

We will now look at the example application that was posted a few days ago, but with some code missing.

https://github.com/kongming92/6170-p3demo/

Clone the repository and switch to the recitation branch:
$ git clone git@github.com:kongming92/6170-p3demo.git
$ git checkout recitation

Then do npm install, and run ./bin/www to start the server.

Server-side "entry point"

We type in the URL to the root route '/' in the browser. This goes into the single route in routes/index.js, which renders the index.ejs view. Notice that index.ejs is simply an HTML skeleton with lots of Javascript files attached and no content! Our Javascript will get data as needed (via AJAX) and populate the HTML.

From here on out, there will be no more page refreshes, and correspondingly no more calls to `res.render(...)` on the server. All server calls will return JSON, and client Javascript will handle it.

Client-side "entry point"

Next, let's look at public/javascripts/index.js. Note the call to $(document).ready(...). This callback function is run immediately when the (blank) page loads. Note that all it does is check if someone is logged in, and load the home page. **Remember that the code in `$(document).ready(...)` is run *exactly once*, since there are no page refreshes!** Therefore the only code you should really put in there is code that starts the page up.

In particular, you should not put event handlers of the form `$('#someButton').click(...)` in `$(document).ready(...)`. The element `$('#someButton')` might not exist when the page loads and only exist when the user does some sequence of actions (e.g. log in, click on "add a freet"). Thus, if that particular event handler were declared in `$(document).ready(...)` but hadn't been loaded to the DOM yet, the corresponding button will not click properly when it eventually does get loaded. One way to do event handlers is seen on index.js:14-25; we instead attach the event to `$(document)` and filter by the element ID.

Finally, examine the code for the `loadPage` function. We simply take a template name and some data, and set the contents of `$('#main-container')` to be the template loaded with data. This might seem like cheating -- after all, you're still "reloading" the entire page! However, you are doing this purely on the client and retain the benefits of keeping operations asynchronous and avoiding the delay of a page refresh.

Login code

Examine the login code, found in public/javascripts/users.js:1-13. We bind to the submit event on the login form. Immediately, we call `evt.preventDefault()` so that the form does not trigger a page refresh.

Next, we construct a POST request to the server, with the data taken from the form. The server will respond with a success status (200 OK) if the login is successful, in which case we go to the homepage. If the login credentials are incorrect, the server's response code (403) will be interpreted as an error, so we run the fail callback instead.

Your turn

1. Add code in templates/secret.handlebars and templates/secrets.handlebars to display individual secrets as well as a list of them.

   In secret.handlebars: add the line `<p>{{content}}</p>`

   In secrets.handlebars: add the lines:

   ```
   {{#each secrets}}
       {{> secret}}
   {{/each}}
   ```

Point out that each secret object has a content field -- you can do GET /secrets in Postman.

Also point out that the reason you could do `{{> secret}}` is because you declared 'secret' as a partial in index.js.

2. Next, we will add the functionality to add and delete secrets. The files that you will have to edit are:
    - secret.handlebars: add an `<a>` tag with class "`delete-secret`"
    - secrets.handlebars: add a div with an error div, a text input with id "`new-secret-input`", and a button with id "`submit-new-secret`"
    - secrets.js: add click handlers for `#submit-new-secret` and `.delete-secret`, which should be AJAX calls follow by DOM manipulation

Make sure to recompile Handlebars templates when you're done at each step to see your changes!