Recap: “Web API”

In lecture we covered several very important conceptual details about web APIs, but there can still be some disconnect between those details and what they mean in practice. Fortunately, you've already been implementing APIs since you began working on Fritter: strictly speaking, the API for your web app is just the methods, routes, and actions associated with the resources you are providing. So with Fritter for example, you may have implemented your API as follows:

Request Action: GET /freet/:freet_id
Request Body: None

Response: a status 200 with an HTML page containing the tweet, a 404, or a redirect back to “/” with an error message

Request Action: POST /freet/:freet_id
Request Body: {  freet: "This is my freet..." }

Response: a status 200 after creating a new freet, with the new freet on the rendered HTML page sent back to the client; assumes a user is logged in with an active session, otherwise reject and redirect with an error message.

Request Action: POST /freet/:freet_id
Request Body: {  delete : true }

Response: a status 200 after deleting the freet with id :freet_id; if the requester isn't logged in, or does not have access to this freet, then reject and redirect with an error message.

The point of this document is to help you refine this interaction using the concepts provided in lecture.

Why APIs?

Many of the ideas used in building APIs come from the same ideas as those seen in 6.005. In 6.005, you built applications with GUIs. Those applications had a number of “model” classes, each of which exposed some “public functions” for other components of your application to call. The model classes were responsible for defining and exposing these functions to allow callers to get the data in some pre-defined manner.
How does this translate to web applications? The modern way of thinking about web applications is that there is some Javascript application running in the browser. This client-side application requires data; it will get this data by making calls to the server. These server calls are HTTP requests, made using some HTTP verb to some route, as we talked about in lecture.

The critical difference between this design and that of Project 2 is that requests to your server are simply just requests for data and not requests for entire HTML pages. This gives us three great benefits:

1. **Greater decoupling between the server and client.** When you visit a site, all the code for views, UI elements, events and behavior, etc., are loaded once. The application runs on the client, and the server is simply treated as a data source and data store that the client application makes calls to.

2. **Greater decoupling between model and view.** We’ve talked about decoupling a lot in this class, and this is just one more example! With the design of Project 2, each HTTP response had data coupled with presentation. The data was intertwined with the HTML that was returned from the server. With a client-side application and server-side API, the responses of the server are purely data, and the client holds the templates and code for presenting the data and allowing the user to interact with it.

   This decoupling between data and presentation has become more and more important. Think about your favorite web applications -- chances are there is a mobile app for the same application. Having a well-designed API means that all of these applications can access and manipulate the same data in a predictable, platform-agnostic way.

3. **Single-page application.** In the older model, every time the user clicked on a link, the server response would trigger a page refresh. From a UI perspective, page refreshes are very expensive because the user cannot do anything until the refresh completes. Even if the user made a small change to a single object, the entire page must be reloaded. In some sense, this “synchronous” behavior results in a less fluid experience, compared to the “asynchronous” behavior that AJAX calls allow for.

A lot of those principles you have learned for designing "good" interfaces from 6.005 should transfer over fairly easily. In particular: defining a clear set of requirements for each of your routes (pre-conditions); defining a clear set of behaviors for each of your routes (post-conditions); and, generally speaking, abstracting away any particular implementation details from the user, such as the specific name of a script which is used to service a request.
Web Services

In pursuit of a refinement for your web APIs, we would like you to think about what kinds of data are at the heart of your web application and exactly how an “abstract” user of your API should obtain this data depending on their particular circumstance. You've already done this with Fritter in a certain sense, but our goal is to refine your notion of how a client and server should interact, and what each should expect of one another. We want to treat the server as an abstract “service” which delivers and manipulates data through programmatic interfaces with which client applications can interact.

Building a Web Application using Web Services

The remainder of Project 3 consists of 2 parts:

1. Build an API, which clients can access to get, add, and modify data
2. Build a client application which provides a “front-end” to your API. In this phase, you become a user of the API that you built in the previous part.

Concretely, we'd like you to build your web app in a way which affords you this kind of flexibility, and we'd like you to do this using AJAX and JSON. So per the API given earlier, for example, you might have an AJAX request to GET /freet/ :freet_id, but instead of having a call to res.render(‘freet’, data) where freet.js is an EJS file that is passed the freet, compiled into HTML on the fly, and sent over to the client, we want you to simply return a JSON object such as

```javascript
{
    freet_id : 10,
    freet : 'the content of the freet...',
    user: ‘dnj’,
    ...
}
```

which is received by the AJAX call and in turn dynamically rendered on the screen.

How do you structure the web application to do this? Generally, there is one “entry point” to the application (for example, www.myapplication.com, which maps to the route ‘/’). This entry point sends back an HTML skeleton -- a very simple, blank HTML document with a few containers and which includes all the necessary Javascript files. From there, the Javascript takes over, loading data and rendering it in the browser as needed.

Let’s take a look at how you might implement Fritter in this fashion.
User visits www.my-fritter.com. The browser makes the request GET /, which the server responds with a blank HTML skeleton. This is the only time the server will respond with HTML (using res.render(...))

The response might look something like this:

```html
<html>
  <head>
    ...
    <!-- javascripts here -->
    ...
  </head>
  <body>
    <div id="main-container"></div>
  </body>
</html>
```

Immediately, the Javascript runs the code found in $(document).ready(...), which makes an AJAX call:

```javascript
$.get('/users/current', function(response) {
  if (response.loggedIn) {
    currentUser = response.user;
    renderPage('freets');
  } else {
    renderPage('login');
  }
});
```

The AJAX call goes to GET /users/current, which responds with an object `{loggedIn: boolean, user: User}`. Of course, you should document your API so that anyone who calls this route knows what to expect.

What does it mean to “render a page”? In our case, this just means to call some function which loads HTML to the DOM. In the code example above, this would likely involve replacing the contents of the div#main-container with the appropriate HTML elements. There are many client-side templating frameworks that accomplish this, the most well-known being Handlebars, but you can also create DOM elements on the fly via jQuery.

So, supposing that we were not logged in, we would clear div#main-container and add a form element with the input fields necessary for login. We would then write code to intercept the form submit event, do an AJAX POST request to the server with the login inputs, and ask the server to verify the credentials.
If we were logged in, we would “render the Freets page”. What does that mean? Well, in order to render Freets, we need to get them first. So we’d probably do something like this:

```javascript
$.get('/freets', function(response) {
    response.freets.forEach(function(freets) {
        // append the freet to the DOM
    });
});
```

This would cause the page to be populated with HTML elements for each Freet.

It's difficult to describe in words the flow of any reasonably complex web application, so we'll be posting some example code for you to study in a bit as well.

The important thing to note here is that all rendering is done on the client. Whenever we need to render something with data that we don’t already have, we make an AJAX call to the API to get it. While this causes us to have more complex client-side code, all of our interaction with the page are asynchronous, and we never have to pay the price of reloading the page as soon as the application loads for the first time. Even our client-side “rendering” -- namely, clearing the container and inserting new things into it -- is orders of magnitude faster and more responsive than a page refresh.

Closing Thoughts

With those details behind us, you might be wondering how any content is rendered if we’re just returning JSON; don’t we need HTML somewhere? In fact the answer is no. Theoretically, you needn’t provide any kind of front-end for your data and the operations you perform on it now that we’ve achieved this decoupling between the data and services on that data the app provides and the presentation of them. Consider: could Google Maps still be useful without any graphical front-end? In a sense, of course it can, if you view Google Maps as a data service which stores massive amounts of location information and can provide things like routing, traffic, and location information through its API.

To recap, the goal here is to get you to think of client/server interaction in terms of the data or the "resources" provided by a web application, instead of viewing servers as entities that opaquely turn some gears and magically spit out a lot of HTML, CSS, and JavaScript with your desired content somehow embedded. Instead, we'd like you to think of client/server interaction in terms of these kinds of requests for and deliveries of resources, and we deliver these resources in JSON format.
Practical Example: GitHub

To tie this together, we can end with a practical example of a modern API from GitHub.

First, take a quick read through the GitHub API Overview (https://developer.github.com/v3/). Don't try to understand every detail, but instead get an idea of what kinds of things this API is trying to accomplish, and how it goes about accomplishing them. Note all the detail about the format of requests, the format of responses, and other items such as authorization and rate limiting.

Now take a look at the specific API “endpoints” for GitHub, in particular the one pertaining to repositories (https://developer.github.com/v3/repos/). Note how all the sections are actions that can be taken on a resource, and for each such action, GitHub is telling us exactly what format it expects requests to have, and exactly what format the user should expect responses to have. Also listed are several examples for each endpoint. Take a read through this documentation to get the idea of what GitHub’s API involves; most if not all of the themes are common to other APIs both large and small.

In short, GitHub’s API serves its content as a “web service” with which users can interact through clean, well-defined interfaces. When creating this API, GitHub’s engineers thought about their data model, how users would like to interact with that data model, and how to design the interface for that data model; this public API is the result of that design process, and it’s this design process which we would like you to understand and emulate.

Public vs Private APIs?

While GitHub is a great example of a well-documented API, you may be thinking that it’s a public one that is used by many thousands of people, whereas we are of course building much smaller-scale projects in 6.170. However, the really big idea from lecture that caps everything off is that you don’t just need to be a massive, public data service like GitHub or Google Maps to benefit from this concept of viewing and interacting with a web application as a web service. Rather, many times any web application can and perhaps should be programmed this way and thereby enforce a stronger separation between client and server, between function and implementation.

Therefore, instead of having your Fritter “API” be designed as earlier in this document, you might instead design the application interface as a “web service” that consumes its own data through a well-defined API interface as described already -- even if you’re the only one using it! Concretely, this means not just returning HTML from your routes, but instead “data” as JSON or XML (or some other data interchange format, but those are by far the most common) in a clear, well-specified format.
This design methodology also makes you think about your data more concretely: for example, what is really necessary information when a ticket is requested? Should we return just the message, or perhaps some subset of the author's information? How much user info should we send along; just a bit, and have a separate request for the rest, if necessary? What is the structure of error messages? These are the kinds of questions that are asked when designing a web service API, you should ask them too when designing your own applications. These kinds of questions are also directly relevant to many of the data design ideas we've discussed in class.
Addendum: Testing

As a demonstration of all these concepts, it is in fact possible to test a web application without delivering any rendered content! Using a framework like QUnit [4] (Note: if using QUnit with AJAX, you need to use the special ‘async’ method for tests) or your own test suite, you can simply make HTTP requests to your API endpoints using different inputs -- some valid, some not -- and ensure that the (JSON) responses you receive back are what you expect.

For example, a test file for Fritter designed as a web service might involve an ajax call such as the following:

```
$.ajax({
    url: "localhost:3000/api/freet/" + test_freet_id,
    method: GET,
    success : function( data ) {
        if (!assert(data.field == expected value)) {
            console.log("Failure!");
        }
        console.log( "Test passed!");
    },
    failure : function (err) {
        console.log("Test failed with error : " + err);
    }
});
```

From this call in our test file, we are synthesizing a GET request to the specified endpoint with a particular freet_id, and when the response comes back (hopefully a “200” success with our freet data), we can have some assertions that the response meets expectation. Note that these tests are exactly analogous to JUnit tests in 6.005, except that we’re using AJAX and JSON with QUnit here.
References


