Fullstack Concepts

Hunter Jorgensen

Course Overview

- So far this semester we've covered purely frontend technologies:
  - HTML
  - Web Design
  - CSS
  - JavaScript
  - React
  - Redux
- Everything we've written so far has run (almost*) entirely in the browser. You could go to an island without internet, and everything would run roughly the same (assuming that you downloaded Bootstrap, etc. beforehand)

Course Overview

- However, apps don’t live just in the browser!
- Everytime you open up a website, it’s making dozens or even hundreds of requests to different servers to find information you want, record metrics, download images, etc.
- We can see this if we open the ‘network’ tab of our browser when we visit a popular site like Google, Amazon.com
- These next modules are focused on building applications and how to think about coding for an app

What is Frontend?

Frontend work is everything we’ve done so far in this course:

- All of the work that users directly interact with
- Styling, templates, user logic
- HTML, CSS, JavaScript, React, Redux
- Often referred to as client-side

And the Backend?

The backend, also known as server side, is anything that happens away from our browser

Examples?

- Java, Python, Node.js, PHP, etc.
- Usually handles more complex business logic, security, accessing/storing data from the database

Understanding Frontend vs Backend

We can think of a server as a building that does business. When you buy or sell to that business, you expect the same behavior everytime, no matter the people that work there, the management, leadership, tools, etc.

This is the same with our Frontend (here, our React App) and Backend (our soon to exist Node app), which we can think of as 2 business interacting with each other. We can write this code in different programming languages, organize them different, etc. but the input and output will be the same.

Fullstack Architecture MVC (Model, View, Controller)

In React/Redux, we learned about the Action, Reducer, Store, View flow of data. As a reminder, it goes like this:

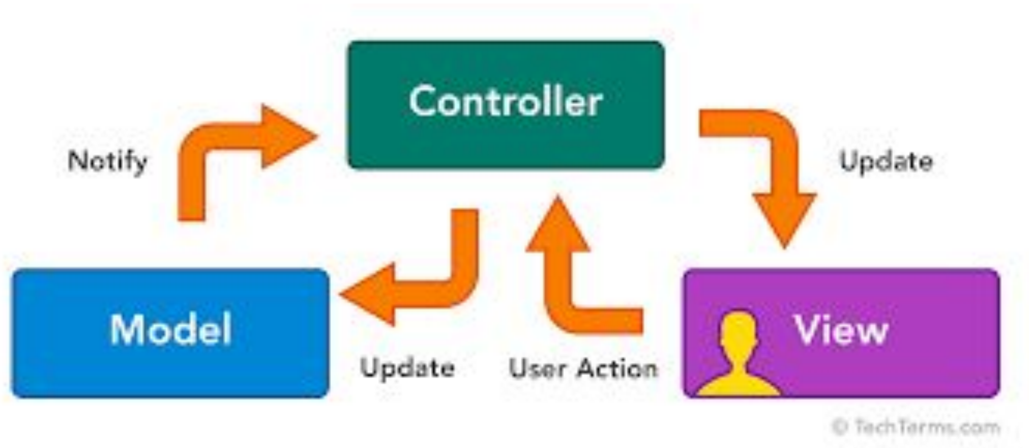
- A user interacts with an app and that *dispatches* and Action
- The Store/Reducer listen for the Action, then change the *state* based on the Action dispatched
- The new state is passed back to the View, and new information is displayed due to that

The purpose here is to separate out the logic, data, and view in a *unidirectional* manner.

What is MVC?

MVC is a somewhat similar architectural pattern that is interested in separating out logic, views and data.

There are many different variations on MVC (i.e. many different ways that the arrow flows in the image below), but the pattern we'll be interested in most like this:



MVC

MVC stands for **M**odel, **V**iew, **C**ontroller.

View: is the part of the app that users interact with. Usually this is an HTML document, but it can be any kind of template, UI, etc. Usually in MVC, this makes requests to the Controller.

Controller: is where all the business logic lives in an app. It handles requests from the view and makes requests to the Model.

Model: this is the data in our app. It is most often stored in databases, but it can also be stored in cache, in-memory, etc.

MVC Example

In previous sessions, we’ve talked about creating simple apps. For instance, say we have a food counting app, where users could count the number of bananas they eat per day.

What would the view be? What would the controller do? How would you represent the model?

Do you think that MVC and the Redux model are similar? How are the different?

HTTP

HTTP, HyperText Transfer Protocol, is a protocol for transferring media documents.

HyperText: this is a document that has references (i.e., hyperlinks) to another document

Transfer: moving something between two points

Protocol: a system of rules

So, HTTP is simply the set of rules that manage how to request and receive documents or media across the web!

HTTP

HTTP was designed in the early 90’s and is meant to be a simple and human-readable way to standardize requests across the internet.

- HTTP is constantly evolving! We currently use on HTTP/2 (i.e., version 2) and version 3 is in development.
- HTTP is simple: you’ll notice that the ideas here are pretty straightforward
- HTTP is based on the client-server model, so that one system makes a request to another system and expects a response
- HTTP is stateless which means that 2 requests will typically have no impact on each other; but it is not sessionless, as cookies can allow us to maintain a user’s current ‘state’.

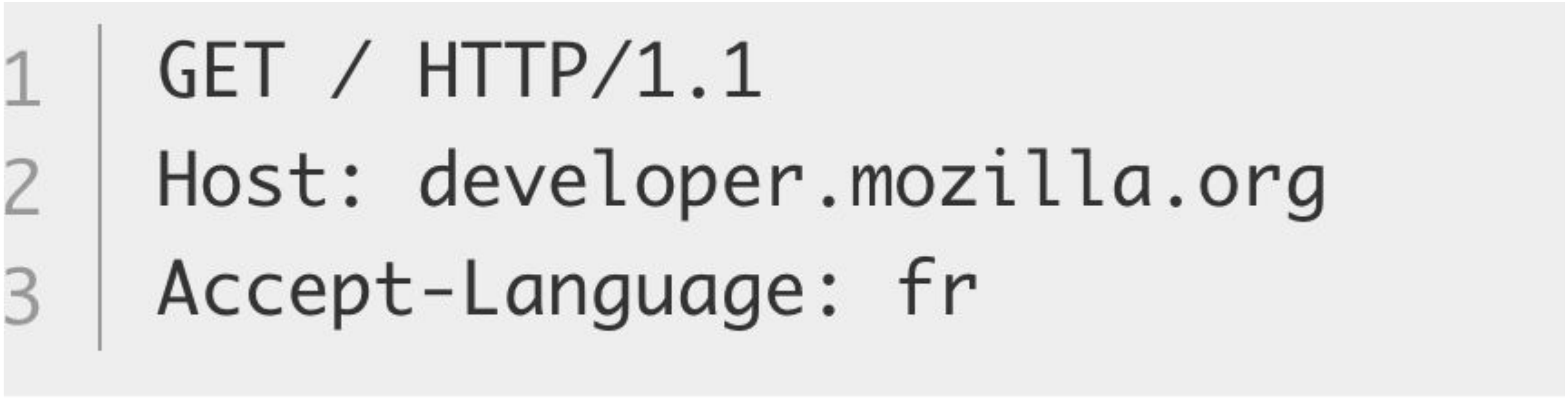
HTTP Example

We use HTTP *all the time*.

1. A user goes to a website. The browser (the user-agent) requests (on behalf of the user) the HTML document that represents that webpage. Notice that the request is coming FROM the user side.
2. The server, based on the request, *serves* the document. A server can be many computers or even a complicated architecture (w/ DB's, caches, etc.)
3. Within that document might be the need for other documents (scripts, CSS, images, etc.)
4. That document might have links to other pages, which will fetch a new page.

Let's look at an example!

HTTP Request Example



```
GET / HTTP/1.1
Host: developer.mozilla.org
Accept-Language: fr
```

HTTP Response Example

```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
Server: Apache
Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT
ETag: "51142bc1-7449-479b075b2891b"
Accept-Ranges: bytes
Content-Length: 29769
Content-Type: text/html

<!DOCTYPE html... (here comes the 29769 bytes of the request
```

HTTP Headers

HTTP requests and responses often have something known as headers, which can additional information that might be useful to the server. Some important headers are:

- Set-Cookies/Cookies, a small bit of user information (does things like keep user logged in, etc.)
- User-agent: details what technology (browser, OS, etc.) what made on the client side
- Host: domain name of the server
- Accept-Language: what language is readable on the client side
- Content-Type: the format of the included media (HTML, JSON, XML, etc.)

HTTP Request Methods/Verbs

HTTP methods, or verbs, indicate the desired action that a client wants done on a server. There are MANY methods, but the most common are:

- GET: a request for a specific resource, this should only retrieve data
- POST: the creation of a new resource on a server
- PUT: updates an existing resource on a server
- DELETE: request to remove a resource from a server

HTTP Request Method Example

It can be helpful to think of these requests with an example.

- Suppose you take a picture and want to share it with friends. So you add it to one of your favorite apps. You are requesting the app create a new resource with this picture, so you are POSTing it on the website.
- If you want to look at the picture later, you go back to your app and find a link to the photo. You are making a GET request.
- You realized that you made a typo on the text of your photo, so you fix that typo and update it. This is a PUT request.
- After a time, you don't like the photo and don't want anyone to use it anymore. You press the delete button, and the photo is removed from the app. This is a DELETE request.

RESTful APIs

You may have heard the term REST in other classes. HTTP and REST use similar verbs, but REST is the way that an HTTP request SHOULD be used.

To clarify, you can create a DELETE HTTP method that doesn’t delete anything, or a GET request that actually creates a new user login. By following REST, you are essentially committing that when you make a HTTP request, you will ONLY do what is set out in the method.

The idea here is that there are no unknown side effects when you make these requests. RESTful APIs are good in all parts of code!

Other HTTP Method Terms

Safe: a safe HTTP is one that doesn’t change the state of the server. GET is the only *safe* method (that we’re learning about), since it should not change the state of the server, whereas POST, PUT, or DELETE might change the database somehow.

Idempotency: is a method that will leave the server in the same state, which means that there are no unexpected side-effects. GET, PUT and DELETE are idempotent; POST is *not*.

HTTP Response Codes

As we develop our backends, HTTP has a set of universal codes to communicate between the server and client about what has happened. Server codes are in ranges:

- 100 - 199, represent information responses (these are not common)
- 200 - 299, represent successful response (200 is commonly used to mean everything went correctly!)
- 300 - 399, represents a redirect (either the URL has changed, etc.)
- 400 - 499, is a client error. This means the request contains something invalid. 404 is a common one, meaning ‘resource not found’.
- 500 - 599 is a server error. This means something bad or unexpected happened on the client, with 500 indicating an ‘internal service error’.

AJAX

AJAX is not a technology but a practice that relies on multiple pieces of technology. The naming of AJAX is very confusion and incorrect, but the ideas are super useful.

AJAX stands for **A**synchronous **J**avaScript **a**nd **X**ML and allows websites to make incremental changes without reloading the entire page. AJAX usually uses JSON more than XML due to ease of working with JavaScript, even though XML is in the name. AJAX relies on HTML, JavaScript, CSS, DOM, XML and JSON to make these requests work

XMLHttpRequest is the actual request from a client to a server that doesn’t require a page refresh. Again, this mainly uses JSON despite the name.

Anatomy of a URL

URLs are easy to ignore, but tell developers a lot about data hierarchy and can provide different ways to pass information to the backend. Additionally, they allow us to organize our site in a somewhat readable and meaningful way.

Anatomy of a URL

This is an example of a URL

http://www.example.com:80/resource/specificResource?key1=value1&key2=value2#SomewhereInTheDocument

- **http**: this is the protocol the browser must use when making this request; we'll learn more about this in the next module
- **www.example.com**: this is the domain name and represents the web server being requested
- **:80**: is the port. You see this locally when developing (port 3000, 5000, etc.) It is omitted when using ports standard to the protocol

Anatomy of a URL

http://www.example.com:80/resource/specificResource?key1=value1&key2=value2#SomewhereInTheDocument

- **resource** and **specificResource**: in our first assignment, these were addresses to physical resources. Now they are abstracted, where resource is part of the path and specificResource is known as a path parameter (we'll see this more detail in the next slide). So for instance, a URL might look like www.website.com/user/1234/review where 1234 is the id of a user (and in this case, maybe we'll be looking for all of their reviews).
- **?key1=value1&key2=value2**: these are query parameters and should always be optional. We'll see these a bit more in the next module, but are used very sparingly on the frontend. They take the form of key/value pairs.
- **#SomewhereInTheDocument**: this is an anchor to a specific part of the document. This will bring users to the first HTML tag with the matching ID and is not sent to the server

Example: Amazon.com, Redfin.com

Let’s look into some websites and see what their URLs look like

Closing Note

Keep in mind that none of the ideas here are mutually exclusive. For instance, an AJAX request could be made with HTTP and built within an MVC model. URL paths for instance are good ways to express what resource you want to get or modify when making HTTP request.

These all work together!