fetch, async/await, and APIs

Michael Chang

Spring 2023

Plan for today

Background: (server-side) dynamic content

fetch and Promises

- Reading external files
- async and await

Intro to REST APIs

- Accessing APIs using fetch
- Using classes to model data from APIs

The problem

We have a web page

- Has some content in the HTML
- Maybe has some data in the JS files

We want to add more dynamic content

- Data from a database or external source
- Data about the current user
- Data that is constantly changing

The problem

One approach

Have the web server return pre-filled HTML

```
<p>Logged in as <span>Michael</span>.</p>
```

Advantages

No work done in the browser

Disadvantages

Data and presentation coupled

Inflexible; can't build e.g. a mobile app or another tool

Can't integrate with other services

Less dynamic (can't get data in background, need to refresh for new data)

The problem

Another approach

Server returns JS code, which client executes

```
"document.querySelector(...).textContent = ...;"
```

Tradeoffs

More dynamic pages

Still embeds page structure in server response

Potential security issues if you're not careful

Still can't integrate external services

APIs

A set of "messages" between programs

- In our case, the client (browser) and server
- Defines what the client can ask for and do
- Defines how the server will respond

Solution

- Write (client-side) JS that makes API requests
  - Server responds with data in some format client understands
- Client interprets the data, makes DOM changes

Detour: asynchronous programming

JavaScript is based on asynchronous, or event-driven, programming

We see this with event listeners and callbacks

Example (pseudocode)

```
main():
  when Add button clicked, call onAdd
  when Delete button clicked, call onDelete
  when checkbox changes, call onUpdate
```

Then main returns

Detour: asynchronous programming

Contrast this with synchronous program

Used in some languages/libraries

Example (pseudocode)

```
main():
  loop forever:
    wait for next event to happen
      if Add button clicked, call onAdd
      if Delete button clicked, call onDelete
      if checkbox changes, call onUpdate
      if Exit button clicked, return
```

main won't return until program exit

Detour: Promises

Promise: standard interface for handling asynchronous code

- Represents something that will happen later (or is happening in background)
- Once finished, the promise "settles"
- It can be in one of three states
  - pending: still waiting on result
  - fulfilled: has a result
  - rejected: error occurred

Detour: Promises

Cannot access result of Promise directly

Need to attach a callback

```
p.then(onFulfill, onReject)
```

After p settles, call one of the callbacks according to its state

(If one arg, called for both fulfilled and rejected)

fetch API

`fetch(url[, options])`

Read contents from a URL (which could be relative)

Returns a `Promise` with the response

`response.status`

Read the HTTP status code of the response

`response.text()`

`response.json()`

Interpret the response body

Returns a `Promise` with the data

fetch example

```
fetch("myfile.txt").then(response => {
  console.log(response.status);
  response.text().then(text => [
    console.log(text);
  });
});
```

"Callback hell"

Problem: too many callbacks

- Each Promise requires a new callback
- Hard to track variables across Promises
- Code gets messy

Partial solution: Promise chaining

- Avoids the nesting, but still annoying
- (We won't talk about this)

Better solution: `async` and `await`

async and await

Same code using async/await

```
const makeRequest = async () => {
  let response = await fetch("myfile.txt");
  console.log(response.status);
  let text = await response.text();
  console.log(text);
};
```

await

await operator

```
await <promise>;
```

Wait for the promise to settle

If fulfilled, return its result

If rejected, throw exception

Only valid inside an async function

async function

async

Mark a function as using await

Function returns a Promise of whatever you return

Syntax

```
const fn = async (args) => { ... };
class Binky {
  async method(args) {
    ...
  }
}
```

async/await gotchas

Can't use await in non-async function

- If you make a callback that uses await, it has to be async too

```
const main = () => {
  let elem = ...;
  elem.addEventListener("click", async (event) => {
    let res = await fetch(...);
    ...
  });
};
```

- Exception: you can use `await` on the console
  - Useful for debugging `fetch` calls

async/await gotchas

async functions return Promises

Even if you don't use await

```
const foo = async () => {
  return 42;
};

/* Can mix/match async and Promise.then */
foo().then(num => {
  console.log(num); // -> 42
});
```

async/await gotchas

If you leave off await, bad things happen

You'll get a Promise, which is probably not what you want

```
const foo = async () => {
  let response = fetch(...); // No await!!
  let text = response.text();
  // Error: Promise has no text() method
};
```

Unfortunately, this can be really hard to debug

REST APIs

Representational state transfer

- Defines certain rules the API will follow

Resources

- Each "thing" we want to send/receive is a "resource"
- Identified by a URI (path)
  - E.g. `/courses/CS193X` or `/users/mchang91`
- Servers return "representation" of the resource
- Clients send (possibly partial) representations to update resources

Statelessness

- Server doesn't "remember" clients
  - I.e. each request includes URI, other info

Representing objects

JSON (JavaScript Object Notation)

Based on JS object syntax, but stricter

E.g. keys must be quoted, only primitive types

```
{
  "id": 1206,
  "courses": [
    { "dept": "CS", "num": "106A" },
    { "dept": "CS", "num": "106A" },
  ],
  "current": true
}
```

Classes and REST APIs

Classes can model resources

E.g. a Student or User class

Loading (reading) a resource

```
class Student {
  /* Can’t make constructor async */
  static async load(id) {
    let data = await ...;
    return new Student(data);
  }
}
```

Classes and REST APIs

Classes can model resources

E.g. a Student or User class

Loading (reading) a resource

```
class Student {
  constructor(data) {
    /* Copy key/values from data to this */
    Object.assign(this, data);
    /* ... init private instance vars */
  }
}
```

Summary

Today

Managing data in the client, interacting with servers

Before next time

assign2.1

Next time

Structure of a REST API

Sending data back to the server

Parts of an HTTP request/response