JSON, APIs, and async/await

Michael Chang

Spring 2020

Plan for today

JSON and REST APIs

- Examples of some APIs

Recap: fetch and Promises

- Reading an external text file

async and await

- Asides: debugging, exceptions
- Accessing REST APIs using fetch

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)

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

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. reach 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
}
```

HTTP requests

Example

GET /students/mchang91?full=1

Method: what we want to do

GET: get some information

POST: send some information (and get response)

(For REST APIs)

PATCH: update an object

DELETE: delete an object

HTTP requests

Example

GET /students/mchang91?full=1

Path: the resource we're accessing

A URI (parts separated by /)

Some parts are fixed (e.g. "students")

Some are identifiers (e.g. "mchang91")

HTTP requests

Example

GET /students/mchang91?full=1

Query string: additional info about resource

Describe what you're looking for

Key/value pairs, separated by &

Keys and values are URI encoded

E.g. ?q=search+string&lang=en

Would encode two key/value pairs

q: "search string"

lang: "en"

HTTP requests

Example

POST /students/mchang91/enroll

Content-Type: application/json

{"course":"cs106a"}

Headers: info about request

What browser we're using (User-Agent)

What type of data we're sending (Content-Type)

HTTP requests

Example

POST /students/mchang91/enroll

Content-Type: application/json

{"course":"cs106a"}

Body: data send to server

Only for non-GET requests

Used when sending full objects

May be the full object (e.g. to create it), partial object (to update), or specific parameters (for custom actions)

HTTP response

Example

HTTP/1.1 200 OK

Content-Type: application/json

{"course":"cs106a"}

Status code: result of request

Gives a general indication of success/failure

Text after the number is generic, specified by HTTP

E.g. 200 will always be "OK", 404 will be "Not Found"

HTTP response

Example

HTTP/1.1 200 OK

Content-Type: application/json

{"course":"cs106a"}

Headers: info about the response

The type of server (Server)

The type of response data (Content-Type)

Body: the resource, error message, etc.

When GETting a resource, probably the object

When an error occurs, often contains a message

When taking an action, info on success/failure

Common HTTP statuses

200 OK

Request was successful

400 Bad Request

Server couldn't understand the request, or couldn't do the thing

401 Unauthorized

Need to log in or send some credentials

403 Forbidden

Credentials provided, but you don't have access

404 Not Found

The thing you asked for isn't there

500 Server Error

Problem on the server side

So far

JSON and REST APIs

- Examples of some APIs

Recap: fetch and Promises

- Reading an external text file

async and await

- Asides: debugging, exceptions
- Accessing REST APIs using fetch

Recap: fetch text file

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

Recall: Promises

Request happens in background

Callback passed to .then() executed when finished

Two steps: first get response, then read body

"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(...);
    ...
  });
};
```

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

Aside: exceptions

try/catch blocks

```
try {
  ...
  throw new Error("Boom");
  ...
} catch (e) {
  console.log(e.stack);
}
```

Aside: exceptions

`throw <expression>`

- Can technically throw anything
- But probably should throw `Errors`

`new Error(message)`

- Automatically builds a stack trace
- Displays nicely in the console
- Can have subclasses of errors

Summary

At this point

- Covered all the core pieces of client size
- Can interact with the page, with data, with servers and APIs

Next time

- Tie up any loose ends
- Start talking about servers and backends

Later

- Come back to client side to talk about cool things
  - Mobile, accessible techniques, animations, login, security