CS193X: Web Programming Fundamentals

Spring 2017

Victoria Kirst (vrk@stanford.edu)
CS193X schedule

Today
- Middleware and Routes
- Single-page web app
- More MongoDB examples
- Authentication
  - Victoria has office hours after class

Monday
- Our last lecture!
- A sprint through some important things I didn't cover
- What's next: An opinionated guide
- Maybe snacks?!?
Final Project Proposal due today 6/2
- No late cutoff! Must turn in on time.

Final Project is due Mon, June 12
- No late cutoff! Must turn in on time.
  - More details posted
    - You will need to create a Video Walkthrough, which will not be graded (aside from completion)
    - Required for every project, Diary app or original
- Additional hints posted too!
Web topic requests

Post to our Piazza post:
- https://piazza.com/class/j0y7gmnuoh167p?cid=184
Modules and Routes
Routes

So far, our server routes have all been defined in one file.

Right now, server.js:
- Starts the server
- Sets the template engine
- Serves the public/ directory
- Defines the JSON-returning routes
- Defines the HTML-returning routes

As our server grows, it'd be nice to split up server.js into separate files.
Recall: Modules
NodeJS modules

You can include it in another JavaScript file by using the require statement:

```javascript
scripts.js

1 require('./silly-module.js');
2
```

- Note that you **MUST** specify "./", "../", "/", etc.
- Otherwise NodeJS will look for it in the node_modules/directory. See [require() resolution rules](#)
module.exports

- **module** is a special object automatically defined in each NodeJS file, representing the current module.

- When you call `require('./fileName.js')`, the `require()` function will return the value of `module.exports` as defined in `fileName.js`

  - `module.exports` is initialized to an empty object.
function-module.js

1 function printHello() {
2     console.log('hello');
3 }
4 module.exports = printHello;
5

scripts.js

1 const result = require('./function-module.js');
2 console.log(result);
3 result();

$ node scripts.js
[Function: printHello]
hello

- We can export a function by setting it to module.exports.
print-util.js
1  function printHello() {
2     console.log('hello');
3  }
4
5  function greet(name) {
6     console.log(`hello, ${name}`);
7  }
8
9  module.exports.printHello = printHello;
10 module.exports.greet = greet;

scripts.js
1  const printUtil = require('./print-util.js');
2  printUtil.printHello();
3  printUtil.greet('world');
4  printUtil.greet("it's me");

$ node scripts.js
hello
hello, world
hello, it's me

- We can export multiple functions by setting fields of the module.exports object
Back to Routes
Routes

So far, our server routes have all been defined in one file.

Right now, server.js:
- Starts the server
- Sets the template engine
- Serves the public/ directory
- Defines the JSON-returning routes
- Defines the HTML-returning routes

As our server grows, it'd be nice to split up server.js into separate files.
Goal: HTML vs JSON routes

Let's try to split server.js into 3 files.

Right now, server.js does the following:

- Starts the server
- Sets the template engine
- Serves the public/ directory
- Defines the JSON-returning routes
- Defines the HTML-returning routes

→ We'll continue to use server.js for the logic in blue
→ We'll try to move JSON routes to api.js
→ We'll try to move the HTML routes to view.js
Goal: HTML vs JSON routes

→ We'll continue to use server.js for the logic in blue
→ We'll try to move JSON routes to api.js
→ We'll try to move the HTML routes to view.js

Desired directory structure:

- dictionary-with-routes
  - node_modules
  - public
  - routes
    - api.js
    - views.js
  - views
    - package.json
    - server.js
Desired: server.js

```javascript
const express = require('express');
const MongoClient = require('mongodb').MongoClient;
const exphbs = require('express-handlebars');

const app = express();
const hbs = exphbs.create();
app.engine('handlebars', hbs.engine);
app.set('view engine', 'handlebars');

app.use(express.static('public'));

const DATABASE_NAME = 'eng-dict2';
const MONGO_URL = `mongodb://localhost:27017/${DATABASE_NAME}`;

let db = null;
let collection = null;

async function startServer() {
  // Set the db and collection variables before starting the server.
  db = await MongoClient.connect(MONGO_URL);
  collection = db.collection('words');
  // Now every route can safely use the db and collection objects.
  await app.listen(3000);
  console.log('Listening on port 3000');
}
startServer();
```

We'd like to keep all set-up stuff in server.js...
Desired api.js (DOESN'T WORK)

And we'd like to be able to define the /lookup/:word route in a different file, something like the following:

```javascript
async function onLookupWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word;

  const query = { word: word.toLowerCase() };
  const result = await collection.findOne(query);

  const response = {
    word: word,
    definition: result ? result.definition : ''
  };
  res.json(response);
}
app.get('/lookup/:word', onLookupWord);
```

Q: How do we define routes in a different file?
Router

Express lets you create Router objects, on which you can define modular routes:

```javascript
const express = require('express');
const router = express.Router();

async function onLookupWord(req, res) {
  ...
}

router.get('/lookup/:word', onLookupWord);

module.exports = router;
```
Router

```
const express = require('express');
const router = express.Router();

async function onLookupWord(req, res) {
  ...
}

router.get('/lookup/:word', onLookupWord);

module.exports = router;
```

- Create a new Router by calling `express.Router()`
- Set routes the same way you'd set them on App
- Export the router via `module.exports`
Using the Router

Now we include the router by:

- Importing our router module via `require()`
- Calling `app.use(router)` on the imported router

```javascript
const api = require('./routes/api.js');
const app = express();
app.use(api);
```

Now the app will also use the routes defined in `routes/api.js`!

However, **we have a bug** in our code...
We need to access the MongoDB collection in our route...

code snippet:
```javascript
const express = require('express');
const router = express.Router();

async function onLookupWord(req, res) {
    const routeParams = req.params;
    const word = routeParams.word;

    const query = { word: word.toLowerCase() };
    const result = await collection.findOne(query);

    const response = {
        word: word,
        definition: result ? result.definition : ''
    };
    res.json(response);
}
router.get('/lookup/:word', onLookupWord);
module.exports = router;
```
...Which used to be defined as a global variable in server.js.

Q: What's the right way to access the database data?

```javascript
let db = null;
let collection = null;

async function startServer() {
  // Set the db and collection variables before
  db = await MongoClient.connect(MONGO_URL);
  collection = db.collection('words');
  // Now every route can safely use the db and
  await app.listen(3000);
  console.log('Listening on port 3000');
}

startServer();
```
Middleware

In Express, you define middleware functions that get called certain requests, depending on how they are defined.

The app.METHOD routes we have been writing are actually middleware functions:

```javascript
function onViewIndex(req, res) {
    res.render('index');
}
app.get('/', onViewIndex);
```

onViewIndex is a middleware function that gets called every time there is a GET request for the "/" path.
Middleware: `app.use()`

We can also define middleware functions using `app.use()`:

```javascript
// Middleware function that prints a message for every request.
function printMessage(req, res, next) {
  console.log('request to server!');
  next();
}

app.use(printMessage);
```

Middleware functions receive 3 parameters:
- `req` and `res`, same as in other routes
- `next`: Function parameter. Calling this function invokes the next middleware function in the app.
  - If we resolve the request via `res.send`, `res.json`, etc, we don't have to call `next()`
Middleware: app.use()

We can write middleware that defines new fields on each request:

```javascript
const db = await MongoClient.connect(MONGO_URL);
const collection = db.collection('words');

// Adds the "words" collection to every MongoDB request.
function setCollection(req, res, next) {
    req.collection = collection;
    next();
}

app.use(setCollection);
```
Now if we load this middleware on each request:

```javascript
async function startServer() {
    const db = await MongoClient.connect(MONGO_URL);
    const collection = db.collection('words');

    // Adds the "words" collection to every MongoDB request.
    function setCollection(req, res, next) {
        req.collection = collection;
        next();
    }
    app.use(setCollection);
    app.use(api);

    await app.listen(3000);
    console.log('Listening on port 3000');
}
```
Now if we load this middleware on each request:

```javascript
async function startServer() {
    const db = await MongoClient.connect(MONGO_URL);
    const collection = db.collection('words');

    // Adds the "words" collection to every MongoDB request.
    function setCollection(req, res, next) {
        req.collection = collection;
        next();
    }
    app.use(setCollection);
    app.use(api);

    await app.listen(3000);
    console.log('Listening on port 3000');
}
```

Note that we need to use the api router AFTER the middleware.
Middleware: app.use()

Then we can access the collection via req.collection:

```javascript
async function onLookupWord(req, res) {
    const routeParams = req.params;
    const word = routeParams.word;

    const query = { word: word.toLowerCase() };
    const result = await req.collection.findOne(query);

    const response = {
        word: word,
        definition: result ? result.definition : ''
    };
    res.json(response);
}

router.get('/lookup/:word', onLookupWord);
```
Middleware: app.use()

Then we can access the collection via req.collection:

```javascript
async function onLookupWord(req, res) {
    const routeParams = req.params;
    const word = routeParams.word;

    const query = { word: word.toLowerCase() };
    const result = await req.collection.findOne(query);

    const response = {
        word: word,
        definition: result ? result.definition : ''
    };
    res.json(response);
}
router.get('/lookup/:word', onLookupWord);
```
Views router

We can similarly move the HTML-serving logic to views.js and require() the module in server.js:

```javascript
const api = require('./routes/api.js');
const views = require('./routes/views.js');

app.use(setCollection);
app.use(api);
app.use(views);
```
const express = require('express');
const router = express.Router();

async function onViewWord(req, res) {
    ...
    res.render('word', placeholders);
}
router.get('/:word', onViewWord);

function onViewIndex(req, res) {
    res.render('index');
}
router.get('/', onViewIndex);

module.exports = router;
Routes and middleware

Simple middleware example code is here:
- simple-middleware
- Run instructions

Dictionary with routes example code here:
- dictionary-with-routes
- Run instructions

Express documentation:
- Router
- Writing / Using Middleware
Recall: Web app architectures
Structuring a web app

There are roughly 4 strategies for architecting a web application:

1. **Server-side rendering:**
   Server sends a new HTML page for each unique path

2. **Single-page application:**
   Server sends the exact same web page for every unique path (and the page runs JS to change what it look like)

3. Combination of 1 and 2 ("Isomorphic" / "Universal")

4. **Progressive Loading**
Structuring a web app

There are roughly 4 strategies for architecting a web application:

1. **Server-side rendering:**
   Server sends a new HTML page for each unique path

2. **Single-page application:**
   Server sends the exact same web page for every unique path (and the page runs JS to change what it look like)

→ Let's talk about this one now
Single-page web app
Single page web app

- The server always sends the same one HTML file for all requests to the web server.

- The server is configured so that requests to /<word> would still return e.g. index.html.

- The client JavaScript parses the URL to get the route parameters and initialize the app.

GET localhost:3000/

index.html
Single page web app

- The server always sends the **same one HTML file** for all requests to the web server.
- The server is configured so that requests to `/<word>` would still return e.g. index.html.
- The client JavaScript parses the URL to get the route parameters and initialize the app.

```
GET localhost:3000/dog
```

```
index.html
```
Single page web app

Another way to think of it:

- You embed **all your views** into `index.html`
- You use JavaScript to switch between the views
- You configure JSON routes for your server to handle sending and retrieving data

```
GET localhost:3000/dog
```
Dictionary example

Let's write our dictionary example as a single-page web app.

**English dictionary**

Look up a word:  

**dog**

The definition of **dog** is:

A quadruped of the genus Canis, esp. the domestic dog (C.familiaris).
Recall: Handlebars

For our multi-page dictionary app, we had two handlebars files: index.handlebars and word.handlebars

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Dictionary lookup</title>
    <link rel="stylesheet" href="style.css">
    <script src="fetch.js" defer></script>
  </head>
  <body>
    <h1>English dictionary</h1>

    <form id="search">
      Look up a word: <input type="text" id="word-input"/>
      <input type="submit" value="Search!">
    </form>

    <hr />

    <div id="results" class="hidden">
      The definition of <a href="" id="word"></a> is:
      <blockquote id="definition"></blockquote>
    </div>
  </body>
</html>
```

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Define: {{ word }}</title>
    <link rel="stylesheet" href="/css/style.css">
  </head>
  <body>
    <h1>{{ word }}</h1>
    <div id="results" class="hidden">
      The definition of <strong id="word">{{ word }}</strong> is:
      <blockquote id="definition">{{ definition }}</blockquote>
    </div>
  </body>
</html>
```
In a single-page web app, the HTML for both the Search page and the Word page are in index.html:

```html
<!-- View for the search page -->
<section id="main-view" class="hidden">
  <h1>English dictionary</h1>
  
  <form id="search">
    Look up a word: <input type="text" id="word-input"/>
    <input type="submit" value="Search!">
  </form>

  <hr />

  <div id="results" class="hidden">
    The definition of <a href="" id="word"></a> is:
    <blockquote id="definition"></blockquote>
  </div>
</section>

<!-- View for a single word -->
<section id="word-view" class="hidden">
  <h1></h1>
  The definition of <strong id="wv-word"></strong> is:
  <blockquote id="wv-def"></blockquote>
</section>
```
Server-side routing

For all requests that are not JSON requests, we return "index.html"

const path = require('path');

async function onAllOtherPaths(req, res) {
    res.sendFile(path.resolve(__dirname, 'public', 'index.html'));
}

app.get('*', onAllOtherPaths);
Client-side parameters

All views are hidden at first by the client.

<!-- View for the search page -->
<section id="main-view" class="hidden">
  ...
</section>

<!-- View for a single word -->
<section id="word-view" class="hidden">
  ...
</section>
Client-side parameters

When the page loads, the client looks at the URL to decide what page it should display.

```javascript
const urlPathString = window.location.pathname;
const parts = urlPathString.split('/');
if (parts.length > 1 && parts[1].length > 0) {
  const word = parts[1];
  this._showWordView(word);
} else {
  this._showSearchView();
}
```
Client-side parameters

To display the word view, the client makes a fetch() requests for the definition.

class WordView {
    constructor(containerElement, word) {
        this.containerElement = containerElement;
        this._onSearch(word);
    }

    async _onSearch(word) {
        const result = await fetch('/lookup/' + word);
        const json = await result.json();
Completed example

Completed example code:

- [dictionary-spa](#)
- See [run instructions](#)
More MongoDB examples
Example: Cross-stitch

Let's say that we want to write a Cross-stitch App, that lets us create and save a cross-stitch drawing (called a "hoop").

→ Simplest version: 1 global drawing
Implementation

There are 625 (25x25) small divs that make up each square of the drawing.

Q: How do we save the drawing to a database?
You need to figure out a way to represent your data, in a way that lets us load the drawing later.

For each colored square, need to know:
- Color
- Position

For the hoop, need to know:
- Name of the hoop
Data representation options

One option: Give every pixel a number, 0-625, and assign each number a color

hoopData = {
    title: "smile",
    pixelData: [
        'white',
        'white',
        ...,
        ]
}
Data representation options

One option: Give every pixel a number, 0-625, and assign each number a color.

**Drawbacks:**

- Would be hard to support grids of different sizes.
- You don't need to store information for every pixel, only the changed pixels.
Data representation options

One option: Give every pixel a number, 0-624, and assign each number a color

Drawbacks:
- Would be hard to support grids of different sizes
- You don't need to store information for every pixel, only the changed pixels
Another option: Give every non-empty pixel a row number, column number, and a color

hoopData = {
    title: "smile",
    pixelData: [
        { row: 12, col: 8, color: 'black' },
        { row: 11, col: 15, color: 'black' },
        ...
    ]
}
Saving data

Since there is only one hoop, saving and retrieving the data is pretty easy:

- On the **client side**, make a fetch POST request:

```javascript
const data = {
  id: this.id,
  name: title,
  data: this.hoop.getData()
};

const fetchOptions = {
  method: 'POST',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(data)
};

await fetch('/save', fetchOptions);
```

(The first time we save, we won't have an id value.)
Saving data

Since there is only one hoop, saving and retrieving the data is pretty easy:

- On the **server side**, upsert the entry to the database:

```javascript
async function onSaveHoop(req, res) {
    const id = req.body.id;
    const name = req.body.name;
    const data = req.body.data;
    let query = {};
    if (id) {
        query = { _id: ObjectId(id) }
    }
    const newEntry = { name: name, data: data }
    const params = { upsert: true }
    const response = await hoops.update(query, newEntry, params);

    res.json({ success: true });
}
app.post('/save', jsonParser, onSaveHoop);
```
On the **client side**, make a fetch GET request when the page first loads:

```javascript
async _loadFromDb() {
  const response = await fetch(':/load');
  const result = await response.json();
  if (result) {
    const nameInput = document.querySelector('#hoop-name');
    nameInput.value = result.name;
    this.hoop.loadData(result.data);
    this.id = result.id;
  }
}
```
Loading data

On the **server side**, retrieving the data is *really* easy, since there is only one hoop:

```javascript
async function onLoadHoop(req, res) {
  const result = await hoops.findOne();
  res.json(result);
}

app.get('/load', onLoadHoop);
```
Completed example

One global cross-stitch hoop:
- [cross-stitch-one-hoop](#)
- See [run instructions](#)
Example: Cross-stitch

Let's extend the cross-stitch App to create and save multiple cross-stitch drawings.

- Each drawing is loaded at localhost:3000/id/<id>
Data representation

We can use the same data representation for each drawing; we're just going to have more than one:
Two screens

A trickier decision is figuring out how to design the two screens, including a unique URL for each image:

**Cross-Stitch**

- Create New Hoop
- smile
- hearts
Let's say that you have a URL at id/59314cc12ac3cce39cd98bf1.

Q: How do we load the URL for this id?
Loading data

One solution: Look at the URL on the client and get the ID

```javascript
const urlPathString = window.location.pathname;
const parts = urlPathString.split('/');
if (parts.length > 2) {
    const id = parts[2];
    new EditView(id);
}
```

(Some other solutions:

- Build the page completely in Handlebars template (icky)
- Inject a global JavaScript variable into the Handlebars template (icky and tricky))
Loading data

The server-side lookups are easy:

```javascript
async function onLoadHoop(req, res) {
    const id = req.params.id;
    const query = { _id: ObjectId(id) };
    const result = await req.hoops.findOne(query);
    res.json(result);
}
router.get('/load/:id', onLoadHoop);

async function onLoadAllHoops(req, res) {
    const result = await req.hoops.find().toArray();
    res.json({ response: result });
}
router.get('/load', onLoadAllHoops);
```
Aside: HashIds

In the cross-stitch app and the e-cards app, we used the raw MongoDB ids in the URL.

That's not great:
- Can be pretty guessable, since they don't change much between objects
- Very long
- Exposes database internals (the id) to the user

→ Try using the HashIds library
Completed example

Multiple cross-stitch hoops

- cross-stitch-one-user
- See run instructions
Authentication
What if you want to add user login to your web page?

- For example, what if we extended the Cross-stitch app so that you had to log in before you could create a new cross-stitch drawing?
Authentication is hard

Trying to write your own login system is difficult:
- How are you going to save passwords securely?
- How do you help with forgotten passwords?
- How do you make sure users set a good password?
- Etc.

Luckily, you don't have to build your own login.
OAuth2

- **OAuth2** is a standard for user authentication
- For users:
  - It allows a user to log into a website like AirBnB via some other service, like Gmail or Facebook
- For developers:
  - It lets you authenticate a user without having to implement log in
- Examples: "Log in with Facebook"
OAuth2 APIs

Companies like Google, Facebook, Twitter, and GitHub have OAuth2 APIs:

- [Google Sign-in API](#)
- [Facebook Login API](#)
- [Twitter Login API](#)
- [GitHub Apps/Integrations](#)

- OAuth2 is standardized, but the libraries that these companies provide are all different.
- You must read the documentation to understand how to connect via their API.
Using OAuth2

All OAuth2 libraries are going to be different, but they work like the following:

1. Get an API key
2. Whitelist the domains that can call your API key
3. Insert a `<script>` tag containing `<company>'s API
4. In the **frontend** code:
   a. Use `<company>'s API to create a login button
   b. When the user clicks the login button, you will get information like:
      i. Name, email, etc
      ii. Some sort of **Identity Token**
Aside: API keys

Generally you're not supposed to store API keys in your GitHub repo, even though we did in HW5 and in some lecture examples.

→ How are you supposed to store API keys?
API keys: Store in Env Vars

Generally you're not supposed to store API keys in your GitHub repo, even though we did in HW5 and in some lecture examples.

→ How are you supposed to store API keys?
→ Best practice: **Use Environment Variables**
  - Set the environment variable on your host, such as [Heroku](https://heroku.com)
  - Can access the environment variable's value in NodeJS via `process.env.VAR_NAME`
Using OAuth2

You need to authenticate the identity of the client on the backend as well:
- In the **backend** code:
  - Use `<company>'s libraries to verify the token from the client is a valid token
Using OAuth2: frontend

- Load the Google API by calling Google's library functions with the client id
- Add a button that, when clicked, prompts the user to log into Google

Login with Google
Using OAuth2: Frontend

Login with Google.
Using OAuth2: Frontend

- When the user logs in, the login callback will fire with information about the user
  - Name, email, etc
  - Will also include an IdentityToken, which will expire after a certain amount of time
Using OAuth2: Backend

- When we want to save information to the client, we should send along the IdentityToken.
Using OAuth2: Backend

- NodeJS can then call into Google's Login endpoint to verify the **IdentityToken** is valid and to get the user's email, name, etc.
Adding user login to Cross-stitch:

- Now we have two collections: Users and Hoops

```javascript
const hoops = db.collection('hoops');
const users = db.collection('users');
```
Saving hoops

Every Hoop now has an author associated with it:

```javascript
async function onSaveHoop(req, res) {
    const idToken = req.body.idToken;
    const userInfo = await auth.validateToken(idToken);

    const userQuery = { email: userInfo.email };
    const userResponse = await req.users.findOne(userQuery);
    const id = req.body.id;
    const name = req.body.name;
    const data = req.body.data;
    let query = {};
    if (id) {
        query = { _id: new ObjectID(id), authorId: new ObjectID(userResponse._id) };
    }
    const newEntry = { name: name, data: data, authorId: new ObjectID(userResponse._id) };
    const params = { upsert: true };
    const response = await req.hoops.update(query, newEntry, params);
    const updatedId = id || response._id;

    res.json({ id: updatedId });
}
router.post('/save', jsonParser, onSaveHoop);
```
Loading hoops

You also need to load hoops by author:

```javascript
async function onLoadAllHoops(req, res) {
  const idToken = req.params.idToken;
  const userInfo = await auth.validateToken(idToken);
  const userQuery = { email: userInfo.email };
  const userResponse = await req.users.findOne(userQuery);
  let result = null;
  if (userResponse) {
    result = await req.hoops.find({authorId: ObjectId(userResponse._id)}).toArray();
  }
  res.json({ response: result });
}
router.get('/load/:idToken', onLoadAllHoops);
```

This is also called an "application-level join"
Completed example

User login for cross-stitch:
- [cross-stitch-user-login](#)
- See [run instructions](#)
MongoDB database design

For more on MongoDB database design, MongoDB wrote a short, helpful blog series:

- [6 Rules of Thumb for MongoDB Schema Design:](#)
  - [Part 1:](#) Basic modeling techniques
  - [Part 2:](#) Referencing
  - [Part 3:](#) Design recommendations

For *a lot* more on database design, take a database class!