Web Applications

CSE183

Fall 2020

Storage Tier I

Baskin Engineering
UC SANTA CRUZ

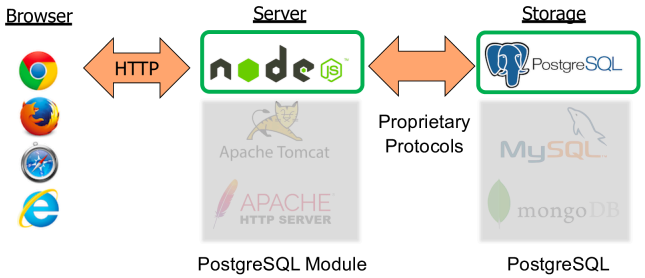
Notices

- **Assignment 7** due 23:59 **Thursday November 19**
- **Assignment 8** available **10:25 Friday November 20**
  - **Due 23:59 Saturday November 28**
    - **48 Hour extension to accommodate Thanksgiving**
- **Assignment 9** available **10:25 Monday November 23**

Today's Lecture

- Relational Databases
- Structured Query Language
- Keys and Indexes
- Object Relational Mapping
- NoSQL Databases
- Hybrid Databases - PostgreSQL
- The Books Database Example - Part I
- Assignment 7 - Secret Sauce
- Questions

Our Full Stack Web Application



Web App Storage Tier

- Essential Properties:
  - Reliable
  - Available
  - Scalable
- Desirable Properties:
  - Handle app evolving over time
  - Easy to use and reason about

Relational Databases

Relational Databases

- Prior to mid 1980's many different data storage structures:
  - File systems, hierarchical databases, index sequential access…
- In 1971 Edgar Codd of San Jose introduced **'The Relational Model'**
  - Immediately a huge success
    - Seen as the answer to almost all data storage problems
    - Many still think it is
- Data is organized as a series of relations (a.k.a. **tables**)
- A relation is made of up of tuples (a.k.a. **rows**)
- A tuple is a set of strongly typed **columns**
  - String: VARCHAR(20)
  - Integer: INTEGER
  - Floating-point: FLOAT, DOUBLE
  - Date/time: DATE, TIME, DATETIME
  - Many others…

https://history-computer.com/ModernComputer/Software/Codd.html

Database Schemas

- Relational Database structure described in a **Schema**
  - Table names
  - The names and types of columns within each table
  - How the tables relate to each other
    - E.g. order table is linked to the user by the user id
  - Many other constraints and optional features

Schema Example

- A table to hold user details
- Columns:

id	INTEGER	Unique identifier of the user
name	VARCHAR(64)	User's name up to 64 bytes
address	VARCHAR(256)	User's address up to 256 bytes

id	name	address
145689	Big Bird	128 Sesame Street, New York, NY
819657	J.R. Biden Jr.	1600 Pennsylvania Avenue NW, Washington, DC
716526	Queen Elizabeth II	Buckingham Palace, Westminster, UK

Structured Query Language (SQL)

- Standard for accessing relational data
  - Based on **Relational Algebra** (super interesting ☺)
- Queries: The Power behind Relational Databases
  - Many ways to extract information
  - You specify what you want…
  - The database system figures out how to get it ***efficiently***
  - Can refer to data by value, not just name

https://en.wikipedia.org/wiki/Relational_algebra

SQL Examples

- Create entities in the schema:

```
CREATE TABLE user (
   id      SERIAL PRIMARY KEY,
   name    VARCHAR(64),
   address VARCHAR(256)
);
```

- Manipulate data:

Create
```
INSERT INTO user (name, address)
VALUES ('Miss Piggy', 'Sesame Street');
```

Retrieve
```
SELECT FROM user
WHERE name = 'Big Bird';
```

Update
```
UPDATE user
SET name = 'Oscar the Grouch'
WHERE id = 123456;
```

Delete
```
DELETE FROM user
WHERE address LIKE = '%Washington%';
```


Keys and Indexes

- Consider: `SELECT * FROM user WHERE id = 2`
- Database could implement this by:
  - Scanning the user table and return all rows with `id = 2`
  - Pre-building an index that maps `id`s to internal table row identifiers
    - Then lookup matching rows from index ☺
- We define **keys** in the DDL to tell databases that building an index may be a good idea
  - Primary Key:
    - Principal index for a table
    - Use `PRIMARY KEY` in `CREATE TABLE` DDL
  - Secondary Keys:
    - Other indexes on data in the table
  - Foreign Keys:
    - Links to other tables

Object Relational Mapping

- Relational model and SQL was a bad match for Web Applications
  - Object versus tables
  - Need to evolve quickly
- 2nd generation web frameworks (e.g. Rails) handled mapping objects to SQL DB
- Rail's Active Record
  - Objects map to database records
  - One class for each table in the database (called Models in Rails)
  - Objects of the class correspond to rows in the table
  - Attributes of an object correspond to columns from the row
- Handled all the schema creation and SQL commands behind object interface

NoSQL Databases

- Relational Databases and SQL provided fast, reliable storage for early web applications - but not very flexible
  - Changing the database schema was a big job ☹
- Led to databases more closely matching the typical Web Application object model
  - Known collectively as **NoSQL Databases**
- MongoDB - Arguably the most prominent NoSQL database
  - Stores collections containing documents (JSON objects)
  - Expressive query language
  - Can use indexes for fast(er) lookups
  - Tries to handle scalability, reliability, etc.
    - But does not always do it very well ☹

Hybrid Databases

- SQL is powerful, but (arguably) inflexible
- NoSQL is flexible, but (arguably) lacks power
- Hey, I know… let's combine the two!

- PostgreSQL supports:
  - Traditional Relational Modeling
  - Unstructured data storage in JSON BLOBS (Binary Large Objects)
  - JSON Queries in SQL (woo-hoo! ☺ ☺ ☺)

Storing JSON

Create a table:

```
CREATE TABLE user (
  id   SERIAL PRIMARY KEY,
  user JSONB
);
```

Insert a row:

```
INSERT INTO user (user) VALUES ('{
  "name":"Big Bird",
  "address":{
    "number": "128",
    "street": "Sesame Street"
  }
}');
```

Note difference between -> and ->>

Find a row:

```
SELECT id, user FROM user
WHERE user->>'name' LIKE '%Bird%';
```

Find a row:

```
SELECT id, user FROM user
WHERE user->'address'->>'street' = 'Sesame Street';
```


https://www.postgresqltutorial.com/postgresql-json/

The Books Database Example

Part I

The Books Database API Schema

- Like The Books Example (in-memory) version

```
paths:
  /v0/books:
    get:
      description: Returns an array of books
      parameters:
        - in: query
          name: author
          schema:
            type: string
          required: false
          description: Author's name, in part or full
      responses:
        200:
          description: Books Response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/BooksResponse'
```

Optional **Query Parameter** on `GET /v0/books` so we can search for books by author

The Books Database SQL Schema

- Single table with two columns:

```
CREATE TABLE book(isbn VARCHAR(16) PRIMARY KEY, book jsonb);
```

Table name

Book details are stored as JSON

ISBNs must be unique and an index of ISBNs will be built for fast searching

The Books Database Example

Still service the Express Routes by mapping them to request handler functions implemented in `books.js`

```
app.get('/v0/books',     books.getAll);
app.get('/v0/books/:isbn', books.getByISBN);
app.post('/v0/books',    books.post);
```

New module to read/write from/to database

Pass the new 'author' query parameter (if any) to the new database module

Delegate getting hold of a single book to the new database module after checking the in-bound ISBN is correctly formatted

Delegate inserting the new book to the database module after checking the ISBN is correctly formatted

```
const db = require('./db');

exports.getAll = async (req, res) => {
  const books = await db.selectBooks(req.query.author);
  res.status(200).json(books);
}

exports.getByISBN = async (req, res) => {
  if (req.params.isbn.match(isbn10) || req.params.isbn.match(isbn13)) {
    const book = await db.selectBook(req.params.isbn);
    if (book) {
      res.status(200).json(book);
    } else {
      res.status(404).send();
    }
  } else {
    res.status(400).send();
  }
}

exports.post = async (req, res) => {
  if (req.body.isbn.match(isbn10) || req.body.isbn.match(isbn13)) {
    const book = await db.selectBook(req.body.isbn);
    if (book) {
      res.status(409).send();
    } else {
      await db.insertBook(req.body);
      res.status(201).send(req.body);
    }
  } else {
    res.status(400).send();
  }
}
```

The Database Module `src/db.js`

Use the PostgreSQL Node.js Module

Create a pool of database connections for SQL queries
Which database to connect to and what credentials to use are stored in environment variables

If the author query parameter was sent to us, use it to restrict the query by doing a case insensitive wildcard search against JSON column
Otherwise select and return all the books

Select a single book by it's ISBN
ISBNs are the book table's primary key, so we can use a standard SQL SELECT

Insert a new row (tuple) into the book table

```
const { Pool } = require('pg');

const pool = new Pool({
  host: 'localhost',
  port: 5432,
  database: process.env.POSTGRES_DB,
  user: process.env.POSTGRES_USER,
  password: process.env.POSTGRES_PASSWORD
});

exports.selectBooks = async (author) => {
  let select = 'SELECT book FROM book';
  if (author) {
    select += ' WHERE book->>'author' ~* $1'
  }
  const query = {
    text: select,
    values: author ? [ `${author}` ] : [ ]
  };
  const { rows } = await pool.query(query);
  const books = [];
  for (const row of rows) {
    books.push(row.book);
  }
  return books;
}

exports.selectBook = async (isbn) => {
  const select = 'SELECT book FROM book WHERE isbn = $1';
  const query = {
    text: select,
    values: [ isbn ]
  };
  const { rows } = await pool.query(query);
  return rows.length == 1 ? rows[0].book : undefined;
}

exports.insertBook = async (book) => {
  const insert = 'INSERT INTO book(isbn, book) VALUES ($1, $2)';
  const query = {
    text: insert,
    values: [ book.isbn, book ]
  };
  await pool.query(query);
}
```


Assignment 7 - Secret Sauce

Assignment 7 - Secret Sauce

- `/v0/mail` and `/v0/mail?mailbox={mailbox}`
- How many URLs?
  - Just one: `/v0/mail`
  - `?mailbox` Is a **query parameter**
- In OpenAPI Schema:

```
parameters:
  - in: query
    name: mailbox
    schema:
      type: string
    required: false
    description: Mailbox name
```


Assignment 7 - Secret Sauce

- Define 3 types of E-Mail in the OpenAPI Schema

  `EMail`
  All properties including `content`
  Returned from `GET /v0/mail/{id}`

  `MailboxEMail`
  All properties *except* `content`

  `NewEMail`
  No `id`, `from-name`, `from-email` properties
  Input to `POST /v0/mail`

- Return from `GET /v0/mail` is an **array of mailboxes**

  `Mailbox`
  `name` string
  `mail` array of `MailboxEmail`

Assignment 7 - Secret Sauce

- Basic:
  - Mailboxes are in-memory arrays of `EMail` read from disk
  - All known mail is a Map of mailboxes, keyed by mailbox name
  - When returning mailbox(es) generate `MailboxEMail` objects from the base `EMail` objects on-the-fly
  - When an email is PUT, create new mailbox in Map if necessary
- Advanced:
  - Follow the model of The Book Example tests
- Stretch:
  - Same as basic except load all files in a directory into the Map
  - Save individual mailboxes only when they change
    - When 'sent' gets a new mail
    - Or when an email is moved between mailboxes (save them both)

Upcoming Lectures

- Friday 20th — Storage Tier II (Assignment 8 Introduction & Quiz 3 Solutions)
- Monday 23rd — Cookies & Session Data (Assignment 9 Introduction)
- Wednesday 25th — Input Validation
- **Friday 27th** — **No Class - Day After Thanksgiving** 

Tasks

- **Assignment 7** due 23:59 **Thursday November 19**

Web Applications

CSE183

Fall 2020

Storage Tier II

Baskin Engineering UC SANTA CRUZ

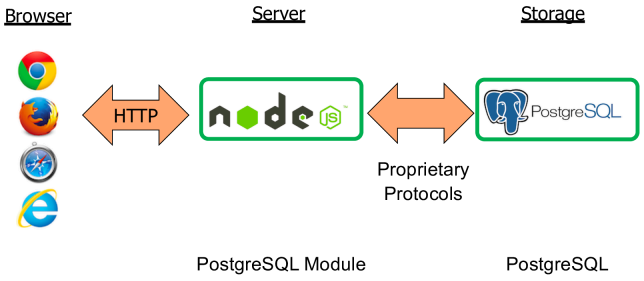
Notices

- **Assignment 8** available **10:25 Friday November 20**
  - **Due 23:59 Saturday November 28**
    - **48 Hour extension to accommodate Thanksgiving**
- **Assignment 9** available **10:25 Monday November 23**
- Class-wide **5 points of extra credit** for helping break-in the new assignments - thank you!

Today's Lecture

- Storage Tier Recap
- Containerisation
- The Books Database Example - Part II
- Quiz 3 - Solutions
- Assignment 8 - Introduction
- Questions

Our Full Stack Web Application



Web App Storage Tier

- Properties:
  - Essential: Reliable / Available / Scalable
  - Desirable: Handle app evolution, easy to use and reason about
- Relational Databases:
  - Data stored as **rows** in **tables** with strongly types **columns**
  - **SQL** for querying the data is enormously powerful
  - Rigid Schema makes it hard to rapidly evolve the Web App model
- NoSQL Databases
  - Store data in an unstructured form
  - More direct mapping to Web App objects in MVC models
- Hybrid Databases
  - Arguably, the best of both worlds
  - Store unstructured data, but query with SQL

Containerisation

- What?
  - Running additional operating systems “on top” of another
  - Distinct from virtualization (CMPS130) but similar
  - Containerised OSs are optimsied for the software they contain
    - Only what is needed - No GUI, no games, not ‘productivity’ apps, etc. etc.
  - Docker is the market leader
- Why?
  - Installing complex, resource intensive software with multiple dependencies (like PostgreSQL) on developer machines is time consuming and error prone ☹
  - Having the same environment in automated test systems as on developer machines reduces the chance of hard to track down problems ☺

The Books Database Example

Part II

The Books Database API

- OpenAPI Schema

```yaml
paths:
  /v0/books:
    get:
      description: Returns an array of books
      parameters:
        - in: query
          name: author
          schema:
            type: string
          required: false
          description: Author's name, in part or full
      responses:
        200:
          description: Books Response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/BooksResponse'
```

Optional **Query Parameter** on `GET /v0/books` so we can search for books by author

- Database Schema

Book details are stored as JSON

```
CREATE TABLE book(isbn VARCHAR(16) PRIMARY KEY, book jsonb);
```

The Books Database Example

Still service the Express Routes by mapping them to request handler functions implemented in `books.js`

```
app.get('/v0/books',       books.getAll);
app.get('/v0/books/:isbn', books.getByISBN);
app.post('/v0/books',      books.post);
```

New module to read/write from/to database

Pass the new 'author' query parameter (if any) to the new database module

Delegate getting hold of a single book to the new database module after checking the in-bound ISBN is correctly formatted

Delegate inserting the new book to the database module after checking the ISBN is correctly formatted

```
const db = require('./db');

exports.getAll = async (req, res) => {
  const books = await db.selectBooks(req.query.author);
  res.status(200).json(books);
}

exports.getByISBN = async (req, res) => {
  if (req.params.isbn.match(isbn10) || req.params.isbn.match(isbn13)) {
    const book = await db.selectBook(req.params.isbn);
    if (book) {
      res.status(200).json(book);
    } else {
      res.status(404).send();
    }
  } else {
    res.status(400).send();
  }
}

exports.post = async (req, res) => {
  if (req.body.isbn.match(isbn10) || req.body.isbn.match(isbn13)) {
    const book = await db.selectBook(req.body.isbn);
    if (book) {
      res.status(409).send();
    } else {
      await db.insertBook(req.body);
      res.status(201).send(req.body);
    }
  } else {
    res.status(400).send();
  }
}
```


The Database Module `src/db.js`

Use the PostgreSQL Node.js Module

Create a pool of database connections for SQL queries
Which database to connect to and what credentials to use are stored in environment variables

If the author query parameter was sent to us, use it to restrict the query by doing a case insensitive wildcard search against JSON column
Otherwise select and return all the books

Select a single book by it's ISBN
ISBNs are the book table's primary key, so we can use a standard SQL SELECT

Insert a new row (tuple) into the book table

```
const { Pool } = require('pg');

const pool = new Pool({
  host: 'localhost',
  port: 5432,
  database: process.env.POSTGRES_DB,
  user: process.env.POSTGRES_USER,
  password: process.env.POSTGRES_PASSWORD
});

exports.selectBooks = async (author) => {
  let select = 'SELECT book FROM book';
  if (author) {
    select += ` WHERE book->>'author' ~* $1`
  }
  const query = {
    text: select,
    values: author ? [ `${author}` ] : [ ]
  };
  const { rows } = await pool.query(query);
  const books = [];
  for (const row of rows) {
    books.push(row.book);
  }
  return books;
}

exports.selectBook = async (isbn) => {
  const select = 'SELECT book FROM book WHERE isbn = $1';
  const query = {
    text: select,
    values: [ isbn ]
  };
  const { rows } = await pool.query(query);
  return rows.length == 1 ? rows[0].book : undefined;
}

exports.insertBook = async (book) => {
  const insert = 'INSERT INTO book(isbn, book) VALUES ($1, $2)';
  const query = {
    text: insert,
    values: [ book.isbn, book ]
  };
  await pool.query(query);
}
```


Database Bootstrapping

- Containerise PostgreSQL with Docker
- Some setup files are needed:
  - Docker Configuration: `docker-compose.yml`
  - PostgreSQL Schema: `sql/schema.sql`
  - Existing Books: `sql/books.sql`
  - Environment Variables: `.env`
- And some changes to `package.json`
  - Bring up Docker container before running our RESTful API Server
  - If first time the container has been brought up:
    - Download and install a PostgreSQL Docker Image:
      - Stripped down Linux with PostgreSQL pre-installed
    - Create the database schema we need
    - Populate the `book` table

`.env` and `docker-compose.yml`

```
POSTGRES_DB=dev
POSTGRES_USER=postgres
POSTGRES_PASSWORD=postgres
```

PostgreSQL Connection properties, default database is `dev`

```
------- 8< ------------------------------

version: '3.7'

services:
  postgres:
    container_name: cse183-books-database
    image: postgres
    environment:
      POSTGRES_DB: ${POSTGRES_DB}
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
    ports:
      - "5432:5432"
    volumes:
      - ./sql/databases.sql:/docker-entrypoint-initdb.d/1.databases.sql
      - ./sql/schema.sql:/docker-entrypoint-initdb.d/2.schema.sql
      - ./sql/books.sql:/docker-entrypoint-initdb.d/3.books.sql
```

Name of Docker Container

The pre-built Docker image to use

Read from `.env`

SQL to run after database is created

sql/schema.sql & sql/books.sql

```
DROP TABLE IF EXISTS book;

CREATE TABLE book(isbn VARCHAR(16) PRIMARY KEY, book jsonb);

------- 8< ------------------------------

DELETE FROM book;

INSERT INTO book(isbn, book) VALUES ('3628598842',
'{"isbn":"3628598842","author":"Roosevelt Meale","title":"Forever,
Darling","publisher":"GutmannFrami"}');

INSERT INTO book(isbn, book) VALUES ('8536737085',
'{"isbn":"8536737085","author":"Avigdor
Paulisch","title":"Ragtime","publisher":"Treutel, Yundt and
Gerhold"}');

INSERT INTO book(isbn, book) VALUES ('7747252757',
'{"isbn":"7747252757","author":"Myron
Blakiston","title":"Cookers","publisher":"Parker, Stamm and
Jacobs"}');

...
```


Testing Support

- Don't want to use the `dev` database for testing
  - Its contents will be unpredictable ☹
- Need a `test` database:
  - Clear it out and install fresh data before each test run ☺
- Also need to:
  - Automatically start the Docker container before testing
  - Automatically stop the Docker container after testing

Books Database Example Tests

```
const supertest = require('supertest');
const http = require('http');

const db = require('./db');
const app = require('../src/app');

let server;

beforeAll(() => {
  server = http.createServer(app);
  server.listen();
  request = supertest(server);
  return db.reset();
});

afterAll((done) => {
  server.close(done);
});

test('GET All', async () => {
  await request.get('/v0/books')
    .expect(200)
    .expect('Content-Type', /json/)
    .then(res => {
      expect(res).toBeDefined();
      expect(res.body).toBeDefined();
      expect(res.body.length).toEqual(256);
    })
});
```

Tests are identical to those for the original books example

```
var fs = require('fs');
const { Pool } = require('pg');

require('dotenv').config();
process.env.POSTGRES_DB='test';

const pool = new Pool({
  host: 'localhost',
  port: 5432,
  database: process.env.POSTGRES_DB,
  user: process.env.POSTGRES_USER,
  password: process.env.POSTGRES_PASSWORD
});

const run = async (file) => {
  const content = fs.readFileSync(file, 'utf8');
  const statements = content.split(/\r\n/);
  for (statement of statements) {
    await pool.query(statement);
  }
}

exports.reset = async () => {
  await run('sql/schema.sql');
  await run('sql/books.sql');
}
```

Set an environment Variable so API Server connects to the `test` database, not `dev`

Create a connection pool so the tests can talk to the database to set up 'known' books.

Utility function to execute lines of SQL in a file one-after-the-other

Create the books table and insert the known books

Quiz 3 - Solutions

Which of the following are recent CSS extensions supporting Responsive Web Design?

- Flexbox grid layout with relative positioning ✔
- Breakpoints based on number of screen and screen size ✔
- Auto scaling of visual media ✔
- Style Polymorphism ✘
- Font anti-aliasing for high resolution screens ✘

Which of the following are legitimate reasons for writing React Function Components rather the Class Components?

- Classes are harder to test ✔
- Classes typically contain more code ✔
- Functions make it easier to adopt best practices ✔
- Functions natively support state encapsulation ✘
- Functions support prototype-based inheritance ✘

Which of the following are provided by Material-UI specifically to make writing a Responsive Web Design React Web App easier?

- Hidden Component ✔
- Grid Component ✔
- Media Queries ✔
- Drawer Components ✘
- Global State via Context ✘

Select all true statements about JavaScript Promises:

- Retrieving a Promise value using then() will block until the Promise is fulfilled ✔
- Promises are a replacement for callbacks ✘
- Promises are fulfilled when the function returning them exits ✘
- Once returned, Promises must immediately be waited on ✘
- Promise chaining is possible because the Promise then() function always returns another Promise ✘

Outline the pros and cons of having a web server return the view components of a web application using an HTTP response header including:

```
Cache-Control: max-age=7200
```

Pro: Using caching reduces server load for resources (images, html, css, etc.) that do not change very often.

Con: If the resource changes, the updated version is not sent until the cache expires.

The Node.js filesystem module 'fs' has a readFileSync function that returns the entire contents of a file directly into a program variable. Whilst this call can be convenient, explain why it is not recommended inside a Web App server.

The fs.readFileSync execution can take a relatively long time if the file must be read from a slow disk.

During that time, the Node.js **event loop** will be waiting for the current function call to return and can't move on to another function, removing any chance of concurrency from overlapping computation with I/O.

Assume you are developing the API for an auto parts company.

An example for listing all parts orders might be:

```
HTTP Method:      GET
Endpoint:         /v0/order
Request Body:     None
Response Body:    JSON
```

Following that example, write API calls for each CRUD operation on a single parts order while adhering to RESTful API design principles.

Your endpoints should use :id to represent the unique identifier of an order.

	Method	Endpoint	Request	Response
C:	POST	/v0/order	JSON	JSON
R:	GET	/v0/order/:id	None	JSON
U:	PUT	/v0/order/:id	JSON	JSON
D:	DELETE	/v0/order/:id	None	JSON

Beyond support for Responsive Web Design, briefly outline three other considerations developers should keep in mind when developing a Web App User Interface.

Accessibility

Internationalisation

Testing

(with accurate descriptions)

With regard to React Hooks:

- **Why are the called hooks?**
- **Why are they necessary?**
- **What are the two main ones?**

Because they let functional Components "hook into" React state changes and lifecycle events.

Because functional Components do not have the "built in' support for state changes and lifecycle events Class Components have.

useState and useEffect.

Assignment 8

Assignment 8

- E-Mail API in Node.js / Express / OpenAPI
- **Backed by Dockerised PostgreSQL ☺**
- Database has three mailboxes full of E-Mails
  - Inbox, Sent, Trash
- Endpoints:

`GET /v0/mail`	Retrieve all the mail
`GET /v0/mail?mailbox={mailbox}`	Retrieve one mailbox
`GET /v0/mail/{id}`	Retrieve one E-Mail
`POST /v0/mail`	Send an E-Mail (goes in Sent)
`PUT /v0/mail/{id}?mailbox={mailbox}`	Move an E-Mail to different mailbox
`GET /v0/mail?from={from}`	**Search for E-Mail (Stretch)**

- Base your submission on your Assignment 7 solution and the Books Database Example ☺

Assignment 8 - Requirements

- **Basic** - 4 Points
  - Write the OpenAPI Schema
    - **Note that it's slightly different to the one in Assignment 7**
  - Implement all the Assignment 7 endpoints
- **Advanced** - 2 Points
  - Write tests for all the Assignment 7 endpoints
  - 100% code coverage
  - Zero linter errors
- **Stretch** - 1 Point
  - **Add the new endpoint**
  - Write tests to demonstrate it works
  - Maintain 100% code coverage and Zero linter errors

Running `npm zip` will create a correctly formatted submission archive for upload to Canvas ☺

Upcoming Lectures

- Monday 23rd — Cookies & Session Data
  (Assignment 9 Introduction & Assignment 6 Simplest Solution)
- Wednesday 25th — Input Validation
- **Friday 27th** — **No Class - Day After Thanksgiving** 

Tasks

- **Assignment 8** due 23:59 **Saturday November 28**

Web Applications

CSE183

Fall 2020

Authentication

Baskin Engineering UC SANTA CRUZ

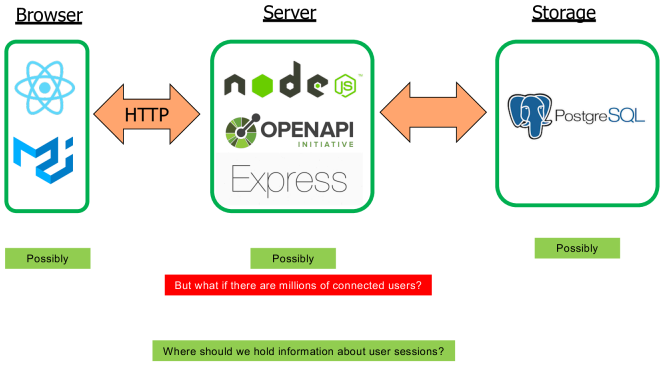
Notices

- **Assignment 8** due 23:59 **Saturday November 28**
- **In-Class Quiz 4** 09:25 **Monday November 30**
- **Assignment 9** due 23:59 **Thursday December 10**
- **In-Class Quiz 5** 09:25 **Friday December 11**
- **Make-up Quiz** 09:55 **Friday December 11**
- **Final Examination** 12:00 - 15:00 **Wednesday December 16**

Today's Lecture

- Cookies
- Express Sessions
- Web Storage API
- JSON Web Token (JWT)
- The Authenticated Books Example
- Assignment 8 - Secret Sauce 
- Questions

Our Full Stack Web Application



Which user sent the HTTP Request?

- Desirable for server to authenticate user once - and have that information available each time we process a request
- More generally web apps would like to keep various pieces of state per “active” browser connection
- Generically known as **Session State**
- Concretely in Express:

```
app.get('/something/:id', routeHandler);
```

- Need to decide to accept or reject the request
- Can use:

```
const sessionState = GetSessionState(req);
```


Session State Lookup Problem

- HTTP request arrives at web server
- Not a lot of information to uniquely identify browser "session"
- Solution:
  - Include something in the request to identify the session
  - Care must be taken to avoid forgeries
- Early HTTP Solution: Cookies
  - State set by web server that browser attaches to every request
  - Useful but with a checkered history
- Modern Solution:
  - Browsers now support **Local Storage API**

HTTP Cookies - Basic Idea

- Web server adds `Set-Cookie:` to HTTP response header

```
Set-Cookie: cookie_name1=cookie_value1
Set-Cookie: cookie_name2=cookie_value2; expires=Sun, 16 Jul 2016 06:23:41 GMT
```

- Each cookie is simply a name-value pair
- Future browser requests to same server should include the HTTP Request `Cookie` header:

```
Cookie: cookie_name1=cookie_value1; cookie_name2=cookie_value2
```

- Cookie Contents
  - Domain for this cookie: server, port (optional), URL prefix (optional)
  - The cookie is only included in requests matching its domain
  - Expiration date: browser can delete old cookies
- Cookie Limits:
  - Data size limited by browsers (typically < 4 KB)
  - Browsers limit the number of cookies per server (around 50)

Cookies as Web App Storage

- User can:
  - View cookies ☺
  - Modify/corrupt cookies ☹
  - Delete cookies ☹
  - Create cookies ☺
  - Lose cookies to hackers ☹
- Simply switching browser tabs looks like you deleted all the app's cookies
  - Cookies have been used in many bad ways (more later in class)
  - Users are suspicious of them
- Pretty unreliable Web App storage ☹
  - Limited to hint, shortcut, etc. that can be recovered if missing
  - While actively communicating with web app: Session cookies

Session State with Cookies

- Early web frameworks supported session state in cookies
- Rails provided `session`, a JavaScript-like object, that you could store anything

```
session[:userName] = "david"
```

- Rails packaged session into cookies and added to HTTP response
  - Data is available in all future requests from the same browser
- Rails automatically checks for a session cookie at the start of each request:
  - Cookie exists? use it to find session data
  - No cookie? Create new session, new cookie
- At end of each request, save session data where it can be found by future requests (must decide where)

Session State in Cookies

- Early approach: Store session state in cookie
  - Since cookies can be viewed, changed, deleted, stolen, etc. care must be taken. E.g.:

```
session.userName = "david";
session.password = "secret";
```

- Using cryptography helps to:
  - Hide content from viewers, hackers
  - Detect forgeries and changes
  - But can't do much about deletions ☹
- Alternative is to insert a reference to the session state:

```
Set-Cookie: session=0x4137fd6a; Expires=Wed, 09 Jun 2012 10:18:14 GMT
```

Less transfer overhead but still need to protect with cryptography

Session State Storage Options

- Web Server memory
  - Fastest access
  - May be too large (many active users)
  - Makes load balancing across web servers hard
- Storage tier
  - Easy shared across all the web servers
  - May be overkill: Don't need the super reliability of storage system
  - May be too much load for the storage tier (need on every request)
- Specialized storage system
  - Support fast fetching of small, short-lived data
  - E.g: memcache, redis - in memory key-value stores

Express Sessions

- Express has a middleware layer to deal with the session state
  - Stores a sessionID safely in a cookie
  - Store session state in a session state store
  - Like Rails, handles creation and fetching of session state for your request handlers
- Usage:

```
app.use(session({secret: 'badSecret'}));
```

  - secret is used to cryptographically sign the sessionID cookie

```
app.get('/something/:id', function (req, res) …
```

  - `req.session` is an object we can read and write

Express Sessions

- Login handler route can store into `req.session.userName`
- All other handlers read `req.session.userName`
  - If not set error or redirect to login page
  - Otherwise we know who is logged in
- Can put other per-session state in `req.session`
- On logout, destroy the session

```
req.session.destroy(function (err) { } );
```

- Default session store is in the Node.js memory
  - OK for development but not production
  - Has session store backends for storage tiers

Web Storage API

- Browser Side Replacement for Cookies
- `sessionStorage`
  - Per-origin storage available when page is open
- `localStorage`
  - Per-origin storage with longer lifetime
- Standard key-value interface:

```
localStorage.appSetting = 'Anything';
localStorage.setItem('appSetting', 'Anything');
sessionStorage['appSetting2'] = 2;
```

- Limited space (~10MB) and similar reliability issues to cookies

JSON Web Token

JSON Web Token (JWT)

- An open standard (RFC 7519) for a compact and self-contained way for securely transmitting information between parties as a JSON object
- This information can be verified and trusted because it is digitally signed
- JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA
- Signed tokens can verify the integrity of the claims contained within it, while encrypted tokens hide those claims from other parties
- When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it
- Say what? ¯_(ツ)_/¯ (Take CSE132 - Computer Security)
- Bottom line:
  - A JWT is a secure way of transmitting information, even over insecure networks
  - If we assume ToFU (Trust on First Use - CSE132 Again)
    - Secure way of authenticating HTTP communications
  - But better if we have HTTPS to encrypt all browser-server communications

https://jwt.io/introduction/

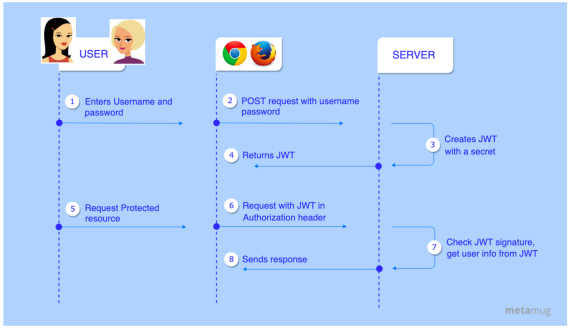
JWT for Authentication

- Say we want to have users “log in” to our Web App
- We store username and hashed password in storage tier
- When user logs in, they supply username and password
- Web App Server checks to see if a user with username exists and checks a hash of the supplied password against the stored hash
- If everything checks out, server issues a JWT to the user’s browser by sending it back in the HTTP Response
- The JWT is sent back to the server with every HTTP Request
- Upside: The user only has to log in once ☺

Books Scenario

- Users:
  - Anna A book club administrator 
  - Molly A member of the book club 
- API:
  - GET /v0/books available to any authenticated user 
  - GET /v0/books/:isbn available to any authenticated user 
  - POST /v0/books only available to administrators 

JWT Process Flow



https://metamug.com/article/security/jwt-java-tutorial-create-verify.html

The Authenticated Books Example

openapi.yaml

```yaml
    ISBN:
      type: string
      pattern: '^(?:ISBN(?:-10)?:?●)?(?=[0-9X]{10}$|(?=(?:

    Book:
      type: object
      properties:
        isbn:
          $ref: '#/components/schemas/ISBN'
        title:
          type: string
        author:
          type: string
        publisher:
          type: string
      required:
        - title
        - author
        - isbn
        - publisher

  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT

security:
  - bearerAuth: []
```

ISBN Format now enforced here, not in code

A JWT Security Scheme enforced by a bearer token in HTTP requests

Use defined security scheme for all end-points

app.js

```js
const express = require('express');
const yaml = require('js-yaml');
const swaggerUi = require('swagger-ui-express');
const fs = require('fs');
const path = require('path');
const OpenApiValidator = require('express-openapi-validator');

const auth = require('./auth');
const books = require('./books');

const app = express();
app.use(express.json());
app.use(express.urlencoded({ extended: false }));

const apiSpec = path.join(__dirname, '../api/openapi.yaml');

let apidoc = yaml.safeLoad(fs.readFileSync(apiSpec, 'utf8'));
app.use('/v0/api-docs', swaggerUi.serve, swaggerUi.setup(apidoc));

app.post('/authenticate', auth.authenticate);

app.use(
  OpenApiValidator.middleware({
    apiSpec: apiSpec,
    validateRequests: true,
    validateResponses: true,
  }),
);

app.get('/v0/books',       auth.check, books.getAll);
app.get('/v0/books/:isbn', auth.check, books.getByISBN);
app.post('/v0/books',      auth.check, books.post);

app.use((err, req, res, next) => {
  console.log('Message: ' + err);
  res.status(err.status).json({
    message: err.message,
    errors: err.errors,
    status: err.status,
  });
});
```

Custom authentication module

End-point clients call to get a JWT, dealt with by a route handler in `auth.js`

Middleware to prevent access to Express Routes when not authenticated

users.json & secrets.json

```json
[
  {
    "email": "molly@books.com",
    "password": "$2b$10$Y00XOZD/f5gBSpDusPUgU.iJufk6Nxx6gAoHRG8t2eMyGgoP2bK4y",
    "role": "member",
    "name": "Molly Member"
  },
  {
    "email": "anna@books.com",
    "password": "$2b$10$Y00XOZD/f5gBSpDusPUgU.G1ohpR3oQbbBHK4KzX7dU219Pv/lzze",
    "role": "admin",
    "name": "Anna Admin"
  }
]
```

Users, their roles and hashed passwords

```json
{
  "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6
}
```

Hard to guess secret for JWT generation and validation

auth.js

```js
const jwt = require('jsonwebtoken');
var bcrypt = require('bcrypt');

const secrets = require('../data/secrets');
var users = require('../data/users.json');

exports.authenticate = async (req, res) => {
  const { email, password } = req.body;
  const user = users.find((user) => {
    return user.email === email &&
    bcrypt.compareSync(password, user.password);
  });
  if (user) {
    const accessToken = jwt.sign(
      {email: user.email, role: user.role},
      secrets.accessToken, {
        expiresIn: '1m',
        algorithm: 'HS256'
      });
    res.json({name: user.name, accessToken: accessToken});
  } else {
    res.status(401).send('Username or password incorrect');
  }
};

exports.check = (req, res, next) => {
  const authHeader = req.headers.authorization;
  if (authHeader) {
    const token = authHeader.split(' ')[1];
    jwt.verify(token, secrets.accessToken, (err, user) => {
      if (err) {
        return res.sendStatus(403);
      }
      req.user = user;
      next();
    });
  } else {
    res.sendStatus(401);
  }
};
```

JWT Module

Hashing Module for passwords

Initial secrets to seed JWT algorithms

"Database" of known users

Get email and password from HTTP request body
Find a user with the supplied email and matching password
if user found
  return a signed JWT
else
  return HTTP 401 "Unauthorized"

If no authorization header
  return HTTP 401 "Unauthorized"
else
  verify the supplied JWT
  if valid
    set the user into the request for downstream components
    call the next middleware component
  else
    return HTTP 403 "Forbidden"

books.js

```
1  const books = require('../data/books.json');
2
3  exports.getAll = async (req, res) => {
4    res.status(200).json(books);
5  }
6
7  exports.getByISBN = async (req, res) => {
8    const book = books.find(book => book.isbn == req.params.isbn);
9    if (book) {
10     res.status(200).json(book);
11   } else {
12     res.status(404).send();
13   }
14 }
15
16 exports.post = async (req, res) => {
17   const { role } = req.user;
18   if (role !== 'admin') {
19     return res.sendStatus(403);
20   }
21   const book = books.find(book => book.isbn == req.body.isbn);
22   if (book) {
23     res.status(409).send();
24   } else {
25     books.push(req.body);
26     res.status(201).send(req.body);
27   }
28 }
```

Only allow admin users to create new books
Return HTTP 403 "Forbidden" for other users

Login.js

from app.js

```
20  app.post('/authenticate', auth.authenticate);
```

```
15  const onSubmit = (event) => {
16    event.preventDefault();
17    fetch('/authenticate', {
18      method: 'POST',
19      body: JSON.stringify(user),
20      headers: {
21        'Content-Type': 'application/json'
22      }
23    })
24    .then(res => {
25      if (!res.ok) { throw res }
26      return res.json()
27    })
28    .then((json) => {
29      localStorage.setItem('user', json);
30      history.push('/');
31    })
32    .catch(err => {
33      alert('Error logging in, please try again');
34    });
35  }
```

POST to the server authentication end-point

Include in the HTTP Request Body the username (email address) and password of the user attempting to log in

If username and password were correct, we'll get an object back containing the use's full name (e.g. "Anna Admin") and the JWT

Store the returned name/JWT object in local storage

Home.js

```
4  const fetchBooks = (setBooks, setError) => {
5    const user = localStorage.getItem('user');
6    const bearerToken = user ? user.token : '';
7    fetch("/v0/books", {
8        method: 'get',
9        headers: new Headers({
10         'Authorization': `Bearer ${bearerToken}`,
11         'Content-Type': 'application/x-www-form-urlencoded'
12       })
13     })
14     .then((response) => {
15       if (!response.ok) { throw response }
16       return response.json()
17     })
18     .then((json) => {
19       setError('');
20       setBooks(json);
21     })
22     .catch((error) => {
23       setBooks([]);
24       setError(`${error.status} - ${error.statusText}`);
25     })
26 };
```

Get the name/JWT object from local storage

GET the API end-point passing the JWT as the Bearer token in the Authorization header to the HTTP Request

If JWT is valid, server will return the requested data

And React state can be set

Assignment 8 - Secret Sauce

Selecting all the Mail

Database Schema:

```
CREATE TABLE mail(id UUID UNIQUE PRIMARY KEY DEFAULT gen_random_uuid(), mailbox VARCHAR(32), mail jsonb);
```

Needs to be on mail when returned

Only a limited number of mailboxes

SQL:

```
SELECT id, mailbox, mail FROM mail;
```

Pseudo JavaScript Code:

```
create map keyed by mailbox name
for each row returned from database
  lookup mailbox array in map
  if does not exist
    create it and insert into map
  mail.id = row.id
  insert mail into mailbox array
for each key in map
  create named mailbox object in mailboxes array
return mailboxes
```


Upcoming Lectures

- **Friday 27th** **No Class - Day After Thanksgiving** 
- Monday 30th In-Class Quiz 4
- Wednesday 2nd Input Validation
- Friday 5th Introduction to Web App Security

Tasks

- **Assignment 8** due 23:59 **Saturday November 28**
- **Assignment 9** due 23:59 **Thursday December 10**