Homework #5

More Colors / Let's Watch Some Movies! (Sessions and Storing Data)

Overview

Repository Creation

A GitHub repository is required to upload your assignment to Gradescope.

👀 Click on this link: https://classroom.github.com/a/6tKmkdAZ (https://classroom.github.com/a/6tKmkdAZ) to accept this assignment in GitHub classroom.

- This will create your homework repository
- Clone it once you've created it.

Goals

There are two main parts to this assignment

1. Session Management / Handling Cookies (Part 1)
    - build your own middleware to parse cookies
    - build your own middleware to create an in-memory session store
2. Storing and Retrieving Data (Parts 2 - 6)
    - use the commandline mongodb client to create a database, collection and several documents
    - use mongoose to read data from mongodb
    - use mongoose to write data to mongodb
    - use pre-built session middleware (Part 6)

Description

By the end of this project… you should be familiar with:

- writing middleware
- setting and reading cookies
- some basic read and write operations with mongodb…
- integrating mongodb with an Express web application using Mongoose (See the example interaction at the end of this page).

You'll create / modify two express apps:

1. A simple demo site that has two pages (partially written for you):
    - a page to change the background color of the site: `/preferences`
        - has a simple form
        - changes made in the form are stored in the user's session
    - a page showing the current data that's stored in that user's session: `/`
2. A site that has a list of movies:
    - the first page is a list of movies: `/movies`
        - displays a table of movies
        - has a form to filter the movies by director name

- the second page allows the user to add movies: `/movies/add`

We'll have a non-standard layout for our project since it will have two separate sites. When you're done with all of the directions, the folder hierarchy may look like the tree below.

👀 **Not all of these files will be present when you first clone (for example, `package.json` must be generated, a `.gitignore` should be supplied), and some files can be named differently - (for example, the naming of template / hbs files is your choice)**

```
├── .eslintrc.cjs
├── .gitignore
├── package.json
├── package-lock.json
├── node_modules
└── src
    ├── cookied
    │   ├── app.mjs
    │   ├── colors.mjs
    │   ├── cookied.mjs
    │   └── views
    │       ├── index.hbs
    │       ├── layout.hbs
    │       └── preferences.hbs
    └── movies
        ├── .env
        ├── app.mjs
        ├── config.mjs
        ├── db.mjs
        └── views
            ├── layout.hbs
            ├── movies-add.hbs
            └── movies.hbs
```

⚠️ **Note that `node_modules` and `.env` should be in your `.gitignore`!**

Submission Process

Submit **homework through gradescope** by uploading your GitHub repository:

- gradescope can be accessed through Brightspace
- gradescope will only accept submissions via GitHub

You will be given access to a private repository on GitHub, and it will contain some initial files (such as a linter configuration and starter files).

Push your code to the homework repository on GitHub so that your latest code is present.

Submit your repo through gradescope. The gradescope assignment will close, so make sure you submit before the deadline.

Note that if you make changes to your repo after you submitted through gradescope, you must resubmit through gradescope (gradescope does not sync changes).

Make at Least 4 Commits

- Commit multiple times throughout your development process.
- Make at least 4 separate commits - (for example, one option may be to make one commit per part in the homework).

Part 1 - Session Management

In this part of the assignment, you'll write two middleware functions: one to parse cookies, and one to manage sessions.

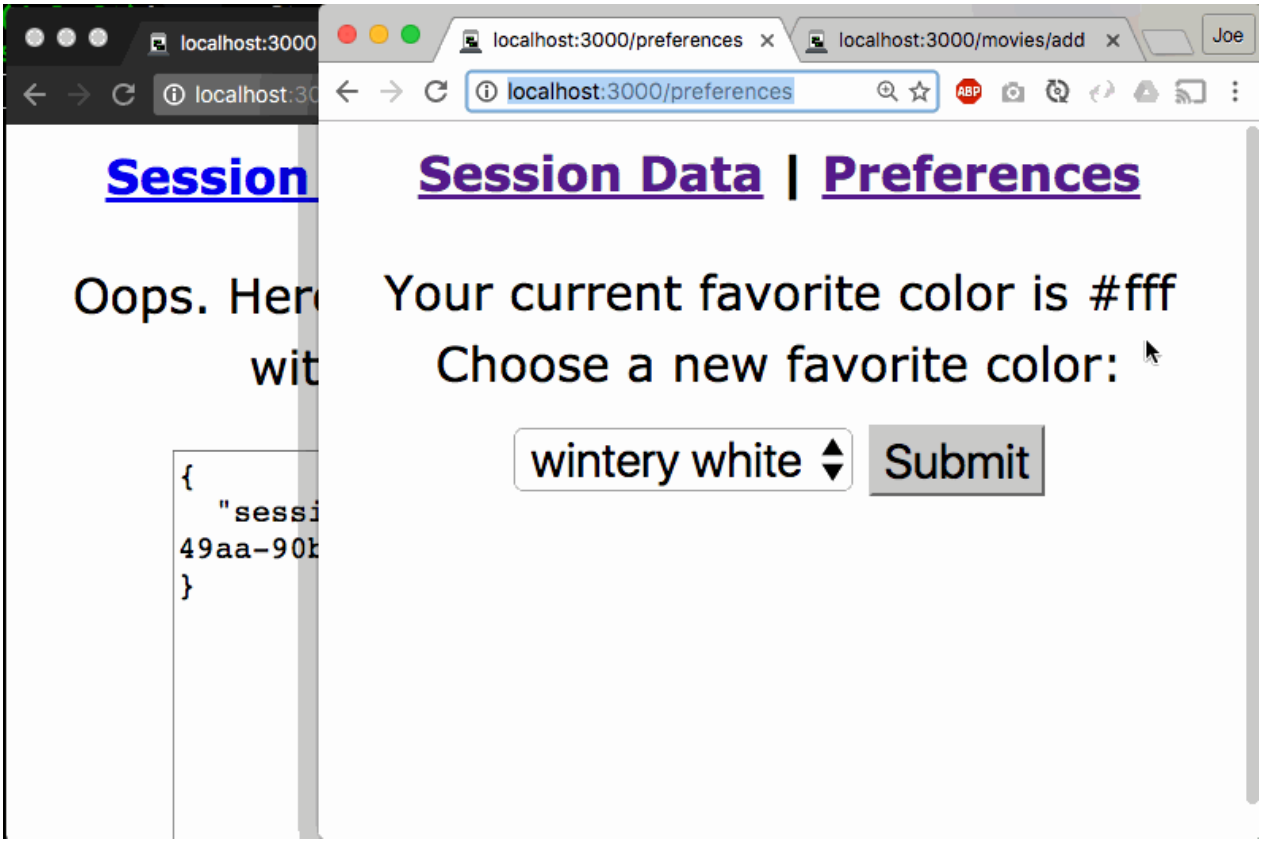
The end goal of the program is to create session management where a session id is assigned to a client via cookies… and that session id is associated with data on the server.

For this program the data that is stored is the user's session id and their favorite color preference (which changes the background color of the site). **The majority of the site is already written for you; you are just implementing middleware**.

Requirement

- create your own cookie handling middleware
- create your own session handling middleware
- ⚠️ **DO NOT USE** the prebuilt `express-session` or `cooke-parser` middleware

See the animation below for the site works (note that the 2nd browser is in incognito mode).



1. in the root of your project (outside of `src`), create the following files with appropriate configuration:
    - `.eslintrc.cjs` (you can use the one that's present or if there isn't one there, use one from our previous projects and customize it if you like)
    - `.gitignore` (minimally ignore `node_modules` and `.env` - which we'll use for configuration for one of our apps)
    - `package.json` (should be auto-generated)
    - `package-lock.json` (should be auto-generated)
2. again, in the root of your project, install the appropriate modules for:
    - the express framework `express`

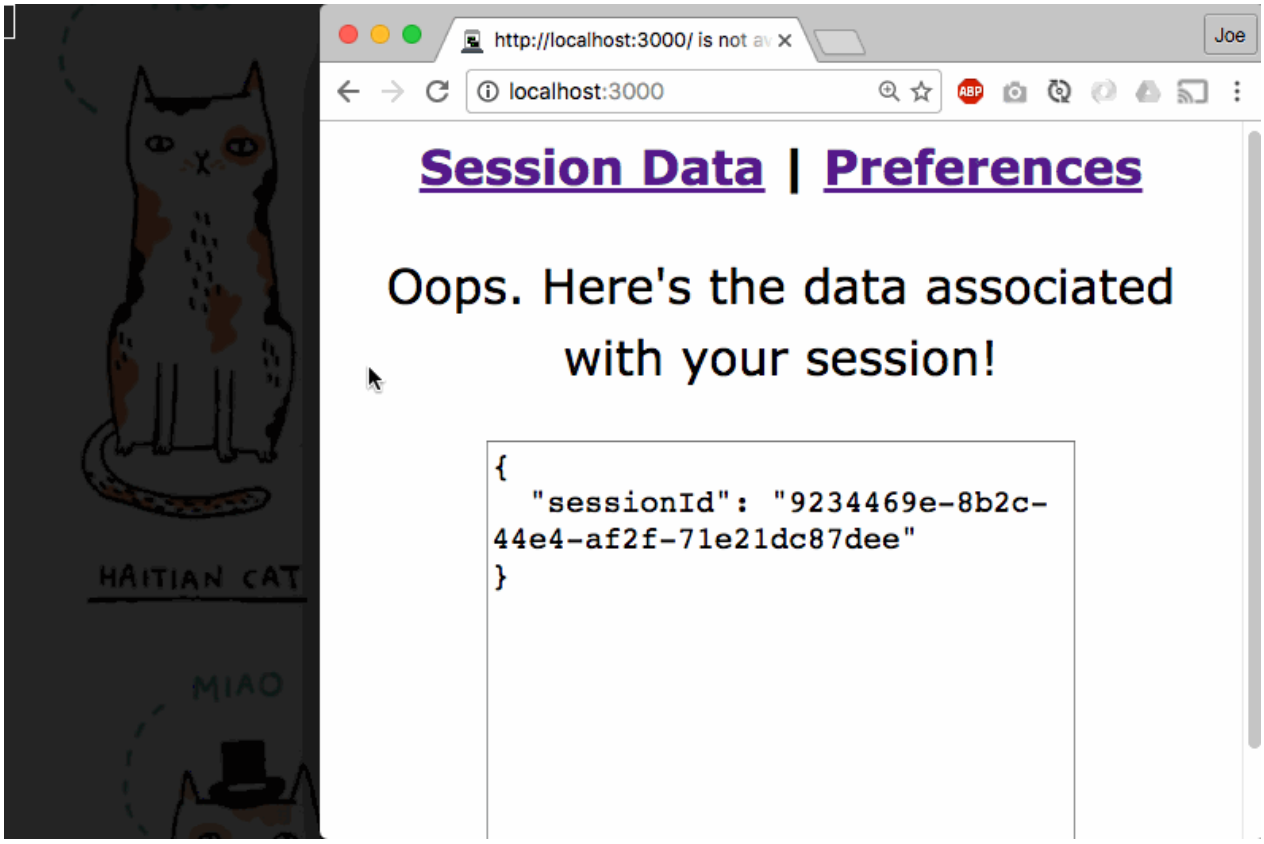
- templating (`hbs`)
- `uuid` (for generating a session id): `npm install uuid`
- double check to make sure that your dependencies are in `package.json`
- there will be more modules to install in the next part…

3. open up `src/cookied/app.mjs` and read through the code to get an idea of what it's doing:
   - it only has 3 routes that render 2 *actual* pages
   - it makes use of two middleware function (both of which don't exist yet; you'll be writing these functions!)
     1. `parseCookies`
     2. `manageSession`
   - finally, note that it listens on a port defined by an environment variable, but defaults to 3000 if the variable doesn't exist:
     - `app.listen(process.env.PORT ?? 3000);`
4. write a `parseCookies` middleware
   - this middleware…
     - checks the incoming request for a `Cookie` header …
     - and parses name value pairs from the value into a property on the 'req' object called `hwCookies`
   - open up `src/cookied/cookied.mjs
   - create a function called `parseCookies`
   - it should have 3 arguments (see middleware (../slides/09/middleware.html))
   - SPOILERS / WALK THROUGH (feel free to implement on your own rather than use the details below)
     - use `req.get` to retreive the `Cookie` header from the request
     - check out our readings to parse the names and values out of the cookie header (MDN on Cookies (https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies) and nczonline's article on cookies (https://www.nczonline.net/blog/2009/05/05/http-cookies-explained/))
     - create a property called `hwCookies` on the request object; it should be initialized as an empty object
     - add the names and values parsed from the `Cookie` as properties and values on `req.hwCookies`
     - don't forget to call `next` when you're done!
     - export the function from the module so that it's availabe when `require` 'd
5. write a `manageSession` middleware
   - this middleware…
     - checks cookies from the request for a session id
     - tries to retrieve data for that session id from an in-memory (*read: global variable*)
     - if it's not an existing session id (or if no session id came through), generate a new session id and send it back to the browser
   - open up `src/cookied/cookied.mjs
   - create a function called `manageSession`
   - it should have 3 arguments (see middleware (../slides/09/middleware.html))
   - when it is done running, `req` should have a property called `hwSession` where data retrieved from the session store is placed
   - additionally, `req.hwSession` should have a property called `sessionId` that stores the current session id (just for debugging purposes)
   - if it generates a session id, the id that it generates should be created by the `uuid` module using `import { v4 } from ('uuid';` , then… `v4()`

- when it tells the browser to set a session id cookie, the name of the cookie should be `sessionId` and it should be `HttpOnly`
- it should also log out `session already exists: [session id]` if the middleware found an existing session
- finally, it should log out `session generated: [session id]` if it creates a new session id
- SPOILERS / WALK THROUGH (feel free to implement on your own rather than use the details below)
  - create a global variable in your module, `cookied.mjs`; initialize it to an empty object (for example `const sessions = {}`
  - this will be your session store - session ids will be keys, and values will be an object of name / value pairs associated with that session
  - within your `manageSession` function, check if `sessionId` is in `req.hwCookies` and check if that session id exists within your session store
    - if the above conditions are true, we know we can set `req.hwSession` to the data that's in our session store for that session id
  - however, if there is no `sessionId` in `hwCookies` or if the `sessionId` isn't in our session store, then generate a new session id and create an empty object for that id's data in the session store
    - again, `req.hwSession` can be set to the data associated with the id from the session store (which, of course, is just an empty object)
    - for example: `req.hwSession = sessions[sessionId]` (assuming `sessions` is the global containing session data)
    - add a `Set-Cookie` header to the response useing `res.append` ... so that the browser will send back the session id in every subsequent request
  - finally, add a property called `sessionId` to `req.hwSession` that has a value of the current session id (regardless of whether or not it was an existing id or one that was just generated)
  - don't forget to call `next` when you're done!
  - export the function from the module so that it's availabe when `require`'d

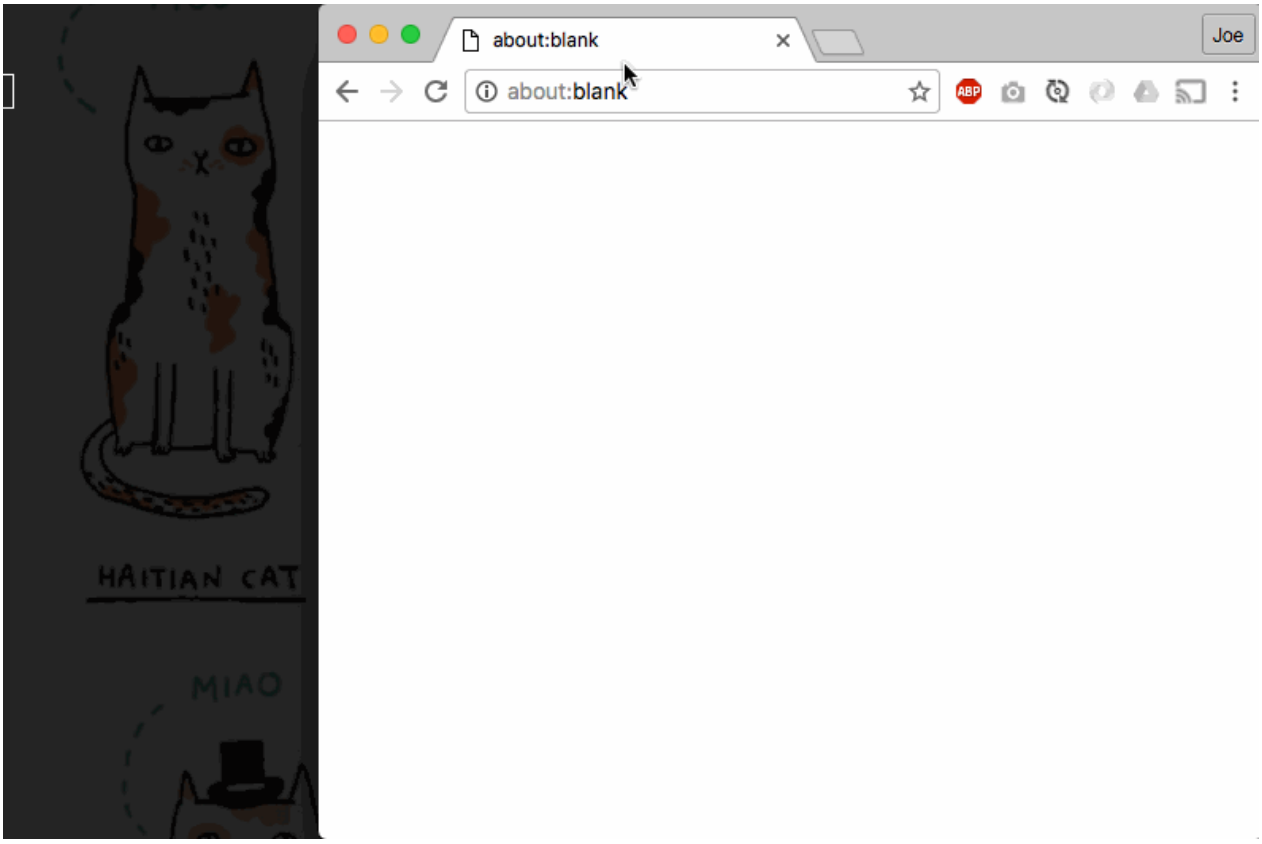
6. when you're done writing your middleware functions, you can test that they all work by running the application (change into the `src/cookied` directory and run `app.mjs`)

7. when you first go to `localhost:3000` , clear your cookies

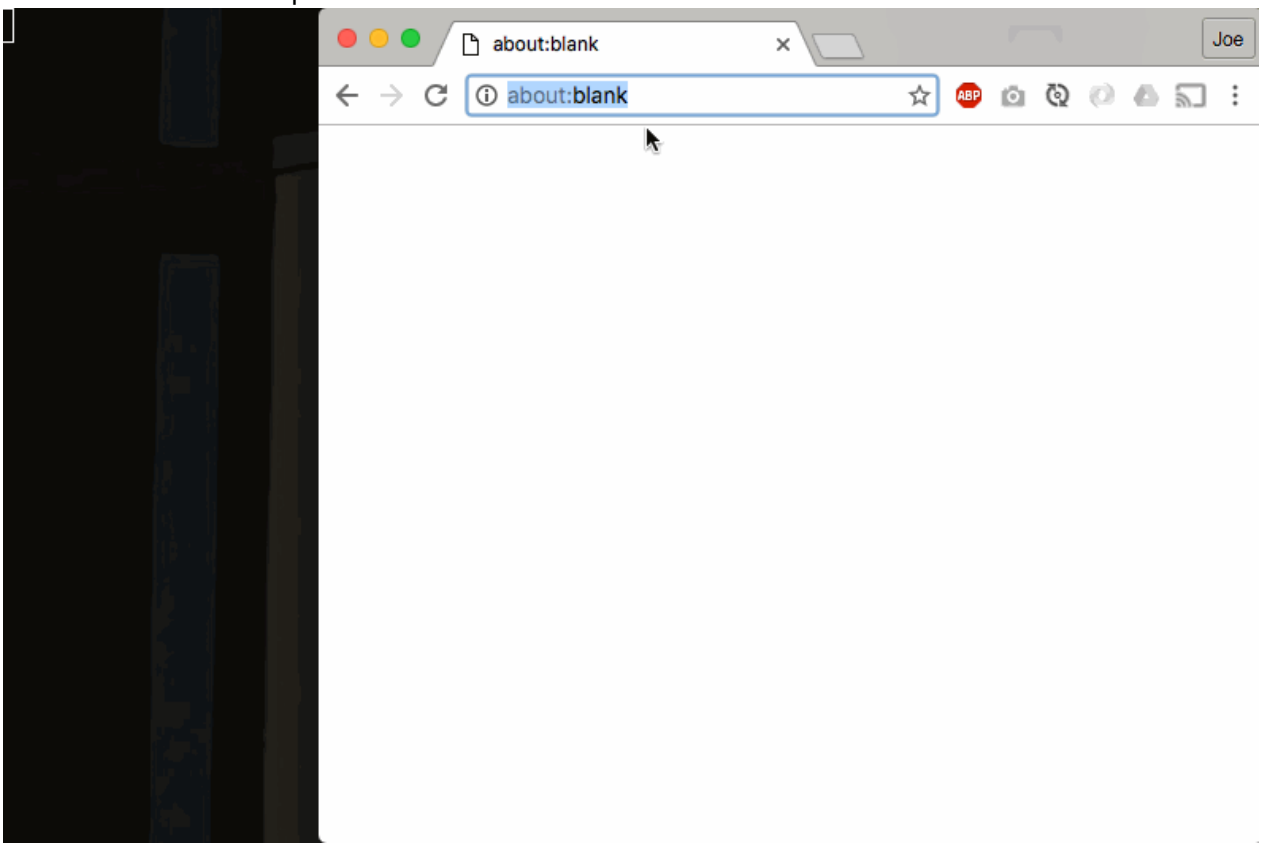


8. refresh your page….
9. check that your application is generating session ids
    - when you refresh, you should see your server logging out `session generated`
    - this should be true for first time visits to both pages: `/` and `/preferences`
    - you should also see a `sessionId` in your browser's cookies for `localhost:3000`
    - `/` should show your current session id as well (again for debugging purposes; the session id shouldn't be so readily viewable!)

- see below for an example:

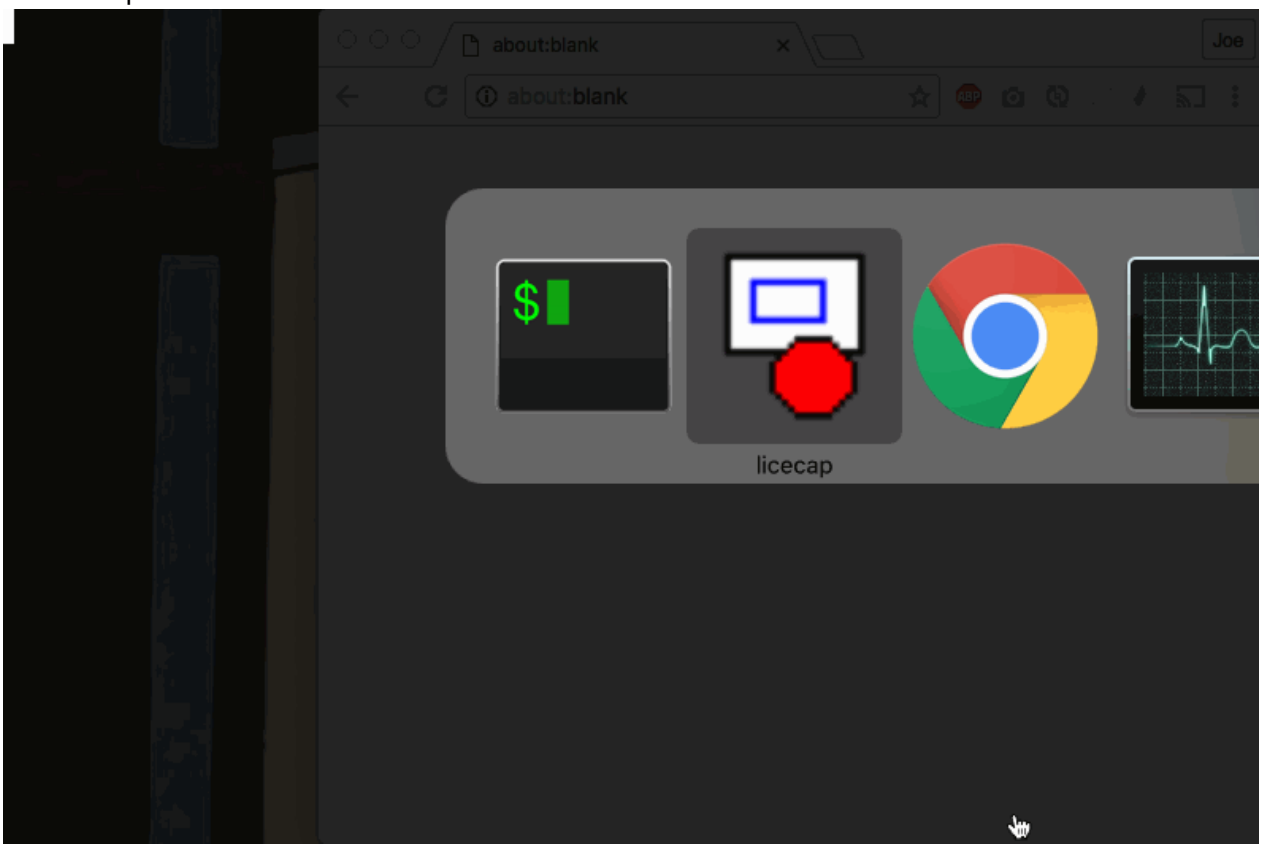


10. note, however, that once a session is generated…
    - if you go to the other page …
    - or refresh again …
    - your application should log out `session already exists`
    - see below for an example:

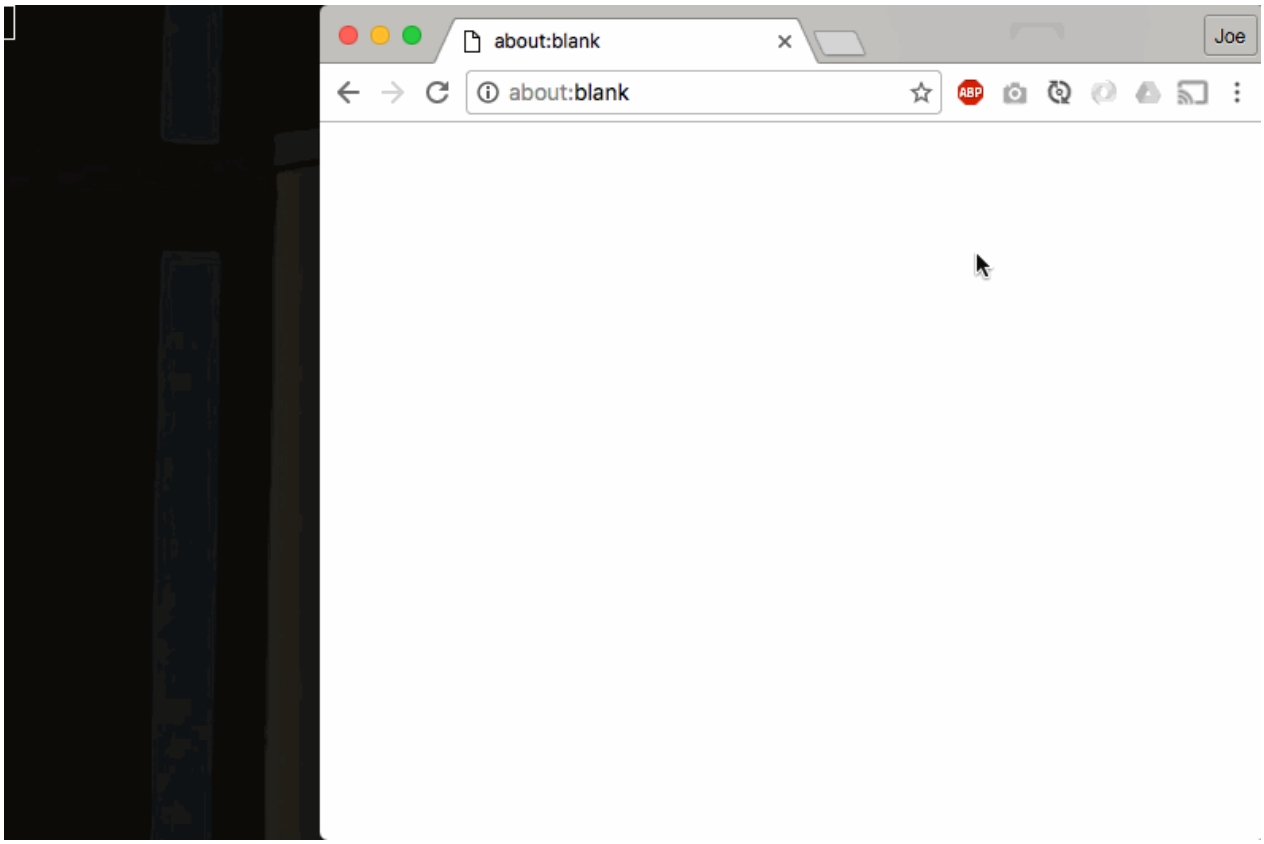


11. finally, if you set a color through the `/preferences` page …

- that color should be persistent for both pages for that session
- which means that opening another browser will have a different color
- `/` will now show that color is part of the session store
- see example below:



12. note that incognito mode will also give you a different session:



Make sure lint your code and fix any warnings.

When finished, commit and push.

Part 2 - Setup for Movies App (Storing Data in a Database)

Installing MongoDB and Preparing Data

- follow the install instructions (https://www.mongodb.com/docs/manual/administration/install-community/) for your operating system
  - make sure to find the **instructions for installing the community version**; DO NOT INSTALL anything else (you might have to scroll down to find the appropriate instructions)
- or use a package manager, like apt or homebrew (brew install mongodb)
- by default, mongodb does not require a username/password to connect (**!?**), but if you'd like to add authentication you can follow this guide (https://docs.mongodb.com/manual/tutorial/enable-authentication/) (optional, but recommended)
- in order for you to connect to your database to work with data, your database server must be running
  - for some installations, mongodb will start when your computer starts
  - for other installations, you'll have to start it manually
  - you can test if your database is running by:
    - attempting to connect to the test database
    - in a terminal window (cmd.exe or windows terminal in windows), type in `mongosh` (in any directory) to start the commandline client
    - you should be given a message with the version number of the Mongo shell
- once you're connected with a commandline client (`mongosh`), start inserting documents into a database called `hw05` and a collection called `movies`:
  - movies will have a title, year and director
  - so to insert, just do this in the commandline client: `db.movies.insertOne({title:'Los Abrazos Rotos', year:2009, director:'Pedro Almodovar'});`
  - (inserting will automatically create the database and collection for you if they don't already exist)
  - insert the following movies (note that you can also use `insertMany` along with an `Array` of objects if you want to run just a single command):

```
Stroszek (1977) by Werner Herzog
Fitzcarraldo (1982) by Werner Herzog
Cave of Forgotten Dreams (2010) by Werner Herzog
Me and You and Everyone We Know (2005) by Miranda July
In the Mood for Love (2000) by Wong Kar-wai
Chungking Express (1994) by Wong Kar-wai
Enough Said (2013) by Nicole Holofcener
Walking and Talking (1996) by Nicole Holofcener
Los Abrazos Rotos (2009) by Pedro Almodovar
```

- use `db.movies.find()` to show all of the movies that you've inserted
  - make sure there's *something* there…
  - so that you know your web app actually has movies to read!
- use `ctrl + d` to exit the commandline client
- (make sure you keep your database server running, though)

Directory Structure and Dependencies

Add dependencies by going back to the root of your project:

- `express` and `hbs` should have already been installed (as well as `uuid`)
- additionally, install **mongoose**: `npm install mongoose` to connect to the database

- lastly, install **dotenv**, a module for loading variables in `.env` into your environment: `npm install dotenv`

Start your usual express app in `src/movies` by:

- creating (or modifying it if it already exists) an `app.mjs` file
- note that once again, we attempt to listen to a port specified by the environment, but default to 3000 if that doesn't exist
  - `app.listen(process.env.PORT ?? 3000);`
- creating the appropriate folders and files for templating (some may already be present)

Setting up `.env` and `config.mjs`

You'll create two files for configuring your application. This will mainly be used for determining the DSN / Data Source Name for your database by using an external file and environment variables.

- create a file called `.env` in the root of your project
- `.env` will contain any configuration parameters for your application
- it may contain sensitive information (such as passwords or api keys), so don't push it to your repository
  - ⚠️ make sure that `.env` is in `.gitignore` so that git doesn't keep track of it!
- edit your `.env` file so that it contains a database connection string, your DSN:
  - to configure without authentication (default), the `DSN` is `mongodb://localhost/hw05` *⚠️ **IF YOU'RE ON WINDOWS** you may have to use the ip address, 127.0.0.1, in place of localhost due to how localhost is mapped to an ipv6 address and mongodb's handling of ipv6 addresses (see https://www.mongodb.com/community/forums/t/econnrefused-27017/131911/5)
  - add the following to your `.env` file:
  - `DSN=mongodb://localhost/hw05`
  - if you've configured authentication: `DSN=mongodb://username:password@localhost/hw04`
  - if you'd like, you can also set `PORT=3000` here (though the express code will default to 3000 regardless)
- create / modify a file called `config.mjs`
  - this file will be used to load the environment variables from your `.env` file
  - in `config.mjs` ...
  - bring in `dotenv`'s `config` function: `import { config } from 'dotenv'`
  - call config to load variables: `config()`
- finally, in `app.mjs`
  - before all other imports, bring in the file that you just created, `config.mjs`
  - `import './config.mjs'` (make sure this is at the top!)

Connect to the Database

Create a file called `db.mjs` within `src/movies`. `db.mjs` will contain:

- the code to connect to our database
- …and our Schema and model (which we'll use to access data in our database)

In `db.mjs`, add the import statement for the `mongoose` module:

```
import mongoose from 'mongoose'
```

Add the code that connects to the database. We'll connect to the local instance of MongoDB, and we'll use a database called `hw05` (this will be created for you once you start inserting documents… which you should have done already above!).

Note that instead of hardcoding a string, you'll be using an environment variable (initialized from your `.env` file)

```
mongoose.connect(process.env.DSN);
```

Leave a placeholder for your schema…

```
// my schema goes here!
```

Or… with authentication

```
mongoose.connect('mongodb://username:password@localhost/hw05');
```

Schema

For larger projects, there is usually one file per schema, all located in a separate folder called models. For now, however, define the following Schema within `db.js`. Check out the slides on:

- the MongoDB Demo (../slides/14/mongo.html)
- and/or the Mongoose API (../slides/14/mongoose.html)
- (or alternatively check out the docs! (http://mongoosejs.com/docs/guide.html))

Since we're storing movies, we'd like each document to have:

- a title (a `String`)
- a director (also a `String`)
- a year (a `Number`)

Create a schema based on the above slides, and insert your code under your `// my schema goes here!` comment.

Then, use your schema to define your model… the model is used as a constructor to create new documents… or as an object with methods that allows the read or update of existing documents.

You can place the following code after your schema and before the connection (assuming that you're schema looks something like this) so that mongoose is aware that your model exists (it *registers* our model so that you can retrieve it later):

```
mongoose.model('Movie', Movie);
```

Part 3 - Displaying All Movies

Overview

We'll be using mongoose to read in all of the movies from the database. Then, we'll be able to display the movies in a table.

Details

There's a bunch of setup that we need in order to integrate our databases access code with our express app:

- in `app.mjs`, import the `db.mjs` file that you created so that the code that you wrote for the Schema and for connecting to the databases is executed (make sure this is after importing `config.mjs`)
- the import statement would look like this: `import './db.mjs';`
- after that, retrieve the model that you registered with mongoose:

```
const mongoose = require('mongoose');
const Movie = mongoose.model('Movie');
```

You can now use `Movie.find` to retrieve all of the movies in your database!

- create a route handler that accepts requests for `/movies`
- in that route handler, the callback should use `Movie.find` to retrieve all movies!
- `find` takes an **query object** (just a regular object) that specifies the criteria for what we're searching for using name/value pairs… for example {year: 1978} would be all movies made in 1978
- if you leave the query object empty, it'll just give back all movies
- `find` will return a promise, so you can:
  - `await` the result
  - but make sure the route handler is specified as `async`!
- you can also wrap your `find` (and `render` if you like) within a `try`, `catch` block to handle errors (if there's an error, you can send back a `500`)
- find works like this:

```
// function that this code is in must be defined as async (`async function` or `async () => {}`)
try {
  const result = await SomeModel.find({search: criteria});
  res.render('myTemplate', {foundData: result});
} catch(e) {
  // send back error status
}
```

- so, once you've retrieved stuff from the database, you'll probably want to render your template… so call `res.render`, rendering whatever template you'd like to display your table
- of course, you'll have to pass in your find results so that you can iterate over them in your template
- in your template, use standard `table` markup, with each row containing a movie
- try opening your page in your browser to show a table of all movies!

Part 4 - Filtering

Overview

In this part of the assignment, you'll add a form to your page that allows you to filter the table by director name via GET and query string parameters.

Details

You already know how to do most of this, but here's a rough sketch of some of the relevant tasks:

- create a form that uses GET, and goes to (submits or makes a request to) `/movies`
  - why are we using GET instead of POST? because we're merely reading data… (pretty common convention for search)
- modify your request handler to try to get the value of query string parameters (`req.query.nameOfFormElement`)
  - for example, submitting your form may result in adding a ?foo=bar to the url
  - to access that name/value pair in the query string on the server side, `req.query.foo`
- use the value passed in from the form (via GET and the query string) to filter the movies by director name

Check out the example interaction at the end of this page

Part 5 - Adding a Movie

Overview

In this part of the assignment, you'll create another page that contains a form to add new movies. The form will POST data… and then redirect back to `/movies`.

Details

You already know how to do most of this, but here's a rough sketch of some of the relevant tasks:

- add a link on the bottom of your /movies page
- set up `express.urlencoded` so that you'll have access to POST data
- create the appropriate route handlers that accepts requests for `/movies/add`
    - GET will handle showing the form
        - create another template file
        - add a form to your template
    - POST will handle the form submission
        - your request handler that deals with POSTs will create a new movie in the database… check out the slides (../slides/14/mongo.html)

Check out the example interaction at the end of this page

Part 6 - Sessions

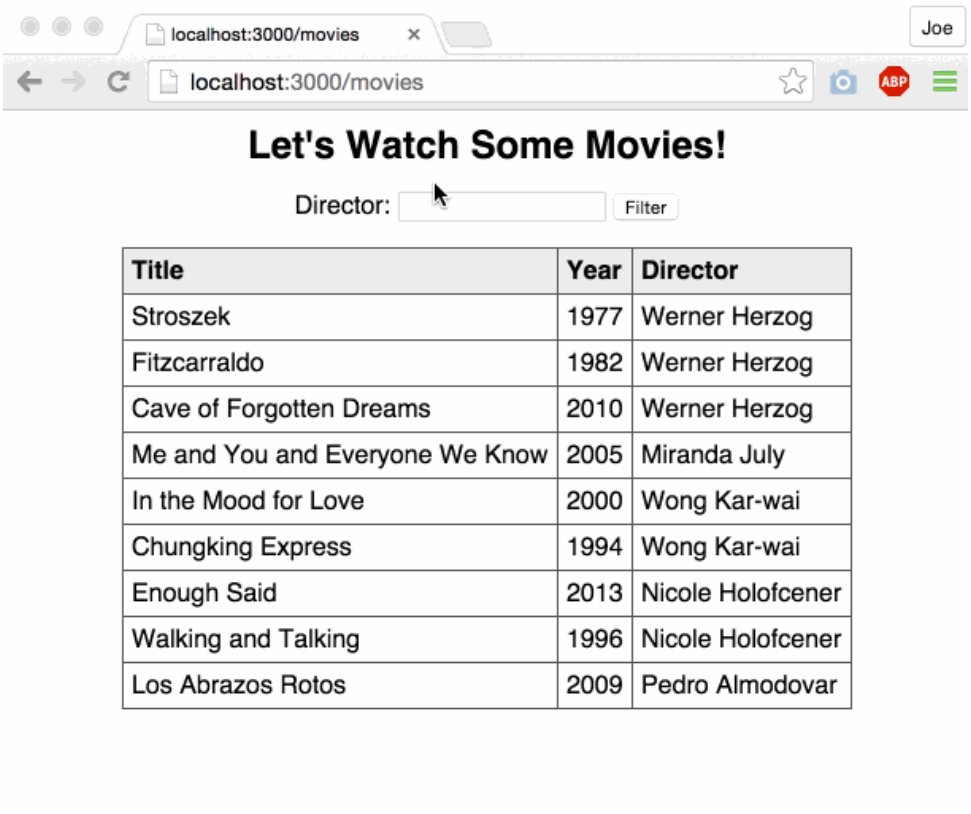
- create another page, `/mymovies`, showing all of the movies that have been added by the user during their session (it can be as simple as an unordered list)
- you must use the `express-session` middleware to do this (see the relevant slides (../slides/10/sessions.html#/17))
- make sure you link to `/mymovies` from both of the existing pages so that the graders can see that you've implemented this feature

Again, make sure lint your code and fix any warnings.

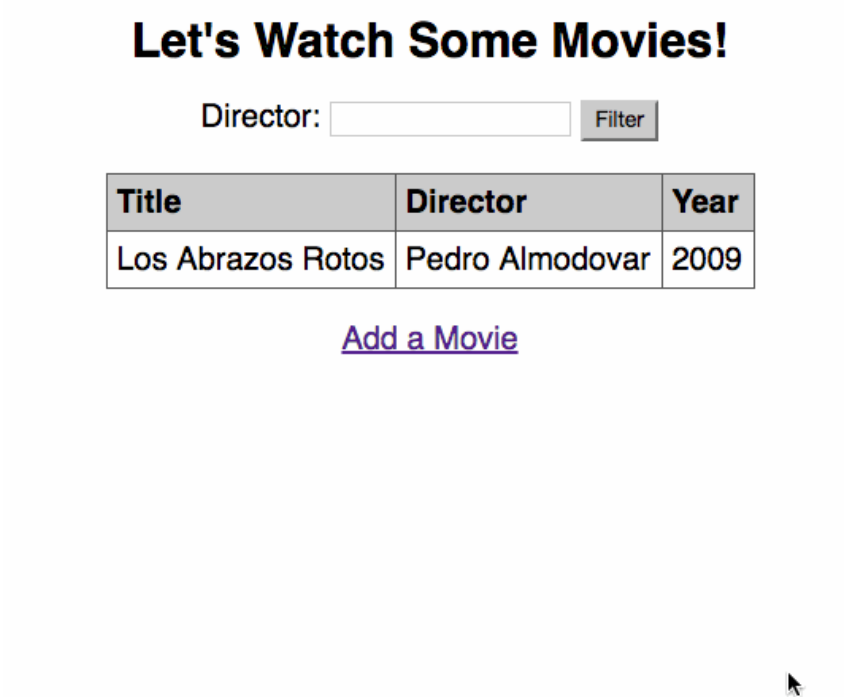
When finished, commit and push to GitHub, then **submit your assignment through gradescope by using your GitHub repo link.**

Example Interaction

By the end of parts 2-4:


localhost:3000/movies
Joe
localhost:3000/movies
Let's Watch Some Movies!
Director:
Filter
Title
Year
Director
Stroszek
1977
Werner Herzog
Fitzcarraldo
1982
Werner Herzog
Cave of Forgotten Dreams
2010
Werner Herzog
Me and You and Everyone We Know
2005
Miranda July
In the Mood for Love
2000
Wong Kar-wai
Chungking Express
1994
Wong Kar-wai
Enough Said
2013
Nicole Holofcener
Walking and Talking
1996
Nicole Holofcener
Los Abrazos Rotos
2009
Pedro Almodovar

Part 5:


Let's Watch Some Movies!
Director:
Filter
Title
Director
Year
Los Abrazos Rotos
Pedro Almodovar
2009
Add a Movie