Homework #6

Authentication and One Hand of Blackjack (Client Side JavaScript)

Overview

Repository Creation

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

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

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: Authentication

Description

Login, Registration, and Multiple Models

Finish a link aggregator site (*like* reddit, hacker news, etc.) that supports user registration and login… along with the ability to post articles (links).

Registering or logging in will create an authenticated session that contains all of the logged in user's information. Some elements on pages will only appear when a user is logged in. Some pages will redirect to login if a user arrives unauthenticated.

Most of the code for this is already written; only three functions must be implemented

Goals

1. use `bcrypt.js` (argon2 (https://www.npmjs.com/package/argon2) is suggested by owasp (https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html), but argon2 may not install on all systems) to salt and hash a password and to compare a hash to a plain text password
2. use the slides on authentication (../slides/16/auth.html) to implement login and registration
3. use `req.params` to capture a part of the url path

The starter code also contains:

- express-session to store user data / an authenticated session
- referenced documents to model users and posted articles

⚠️⚠️⚠️ WARNING ⚠️⚠️⚠️

This homework is for learning purposes only; **do not use it** for authentication on a deployed site

- our application will only be served locally, and consequently, it will not be served over an encrypted connection … and - related - cookies aren't set to secure
- it will allow case sensitive usernames
- not all errors are accounted for or handled gracefully
- system level errors and user errors may not be distinguishable
- some error messaging may reveal **too much** information
- user interaction and error messaging will be minimal (for example, successful login should redirect to page that required login)
- some error messages reveal info about the existence of a user
- our secret for session options is kept in our repository

Features

The following features are partially implemented

1. register a new account
2. login using an existing account
3. show a single article's details

These features are fully implemented

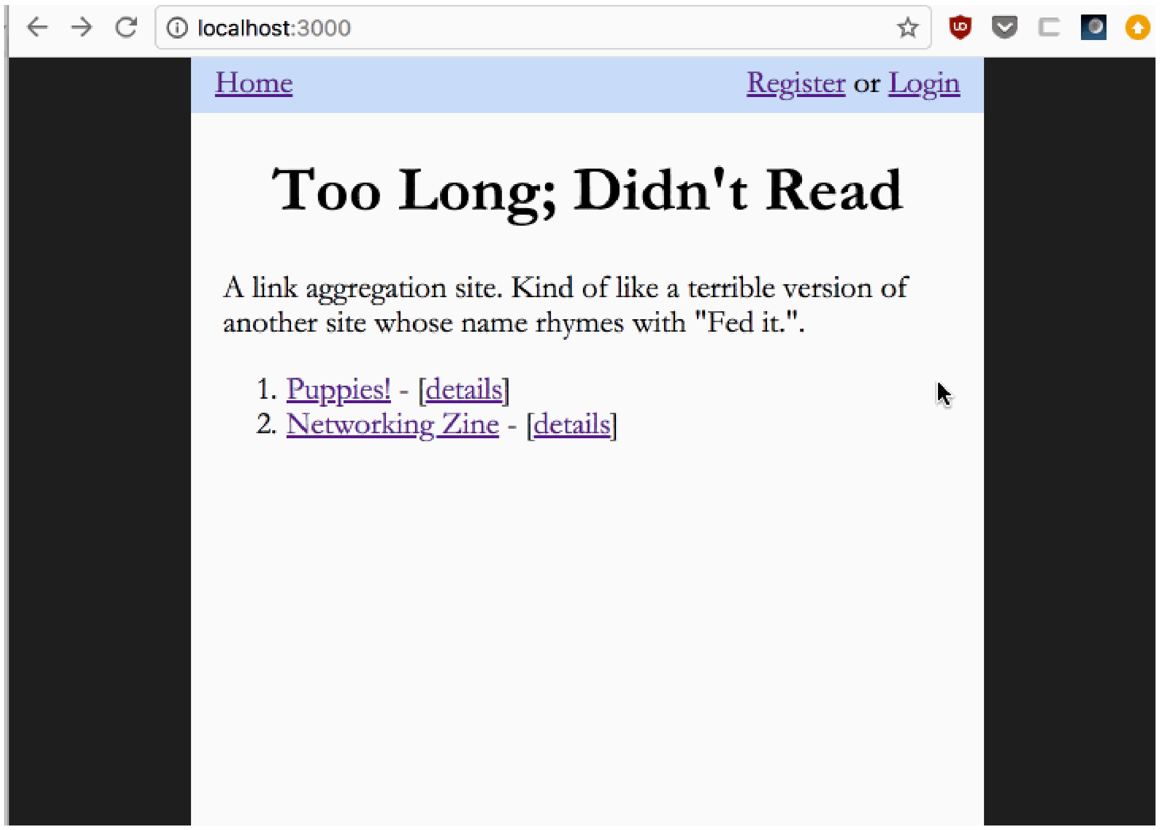
1. view all posted articles
2. add a new article (only when logged in)
3. prevent / allow access to certain ui elements or pages based on authenticated status

The app has 5 pages and 3 forms:

- **/** - lists all articles
- **/register** - register form
- **/login** - login form
- **/article/add** - add new article form
- **/article/:slug** - detail page for a specific article

Example Interaction

Here's what it looks like to login, add a new article, and view the link and details page:.



Too Long; Didn't Read

A link aggregation site. Kind of like a terrible version of another site whose name rhymes with "Fed it.".

1. Puppies! - [details]
2. Networking Zine - [details]

Setup

- within the project root, install the dependencies with `npm install`
  - it should install all of our typical express modules: `express`, `hbs`, `express-session`
  - `mongoose` and `mongo-sanitize` - for database access and cleaning user input
  - `bcryptjs` - module for salting, hashing and comparing passwords
  - `mongoose-slug-plugin` - plugin for auto-generating slugs (human readable, unique identifiers for documents)
- `cd` into the the `src` directory
- create a `.env` file with a DSN: `DSN=mongodb://localhost/hw06` (use `DSN=mongodb://127.0.0.1/hw06` if on windows)
- use `node` or `nodemon` to run your application

A Note About bcrypt.js

`bcrypt.js` is a JavaScript implementation of a password hashing function called (you guessed it!) bcrypt. We'll use it for login and registration. The result of using bcrypt contains both the hash and the salt! Check out the details in the first section of the wikipedia article (https://en.wikipedia.org/wiki/Bcrypt) and the diagram below illustrating the output of bcrypt:

```
$2a$10$N9qo8uLOickgx2ZMRZoMyeIjZAgcfl7p92ldGxad68LJZdL17lhWy
|_____||_____________________||______________________________|
   |             |                          |
   |             |                          +-- hash
   |             |
   |             +-- salt
   |
   +-- algorithm and cost factor
```

Check out how to use bcrypt.js (https://www.npmjs.com/package/bcryptjs).

Fix Template Code Errors / Config

- in `auth.mjs`
  - modify `startAuthenticatedSession` so that the line that calls reject so that the value passed in is `err` rather than `error`
    - `reject(err);`
  - the messaging for registration error should be swapped (`USERNAME ALREADY EXISTS` for registering with duplicate username, and `USERNAME PASSWORD TOO SHORT` for password that does not meet the minimum length requirements)
- modify `.eslint.js` to handle starter code issues regarding the `Promise` constructor executor function:
  - within the `rules` key, add: `"no-async-promise-executor": "off"` (see the docs for more details (https://eslint.org/docs/latest/rules/no-async-promise-executor))
  - OR use the "alternate" implementations for `login` and `register` as described in the specs below
- modify `.eslint.js` so that the linter options include the following within `parserOptions`:

```
    "parserOptions": {
        "ecmaVersion": "latest",
        "sourceType": "module",
        "ecmaFeatures": {
            "experimentalObjectRestSpread": true
        }
    },
```

Complete the Mongoose Article Schema

In `src/db.mjs`, a `UserSchema` containing `username`, `email`, and `password` (the combined salt and hash) is already present.

A schema representing an article, `ArticleSchema`, is partially implemented; to finish it:

- add the following fields:
  - `title`
  - `url`
  - `description`
- additionally, add a field that references a user document
  - every article will "link" back to a user
  - you must use referenced documents with population (https://mongoosejs.com/docs/populate.html) – rather than embedded documents (https://mongoosejs.com/docs/subdocs.html) to implement this
    - when relating documents, add a `User`'s `_id` as a value for the `user` field in `ArticleSchema`
    - see the mongoose docs (https://mongoosejs.com/docs/populate.html) on references and population
- note that the module, `mongoose-slug-plugin`, is present in the implementation to add a slug - a human readable unique identifier for each article:
  - this plugin will autogenerate a `slug` field (no need to explicitly add it to your schema)
  - a slug is a string that serves as a short, human readable name
  - it usually contains dashes to separate words, along with a number as a suffix
  - for example, `this-is-a-slug`
  - the code to create a slug using the configuration, `{tmpl: '<%=title%>'}`, is already present

Complete Registration

Finish the implementation of the `register(username, email, password)` function in `src/auth.mjs`. This function will:

- check if the given username and password are valid (meets minimum length, username doesn't already exist)
- saves a new user given the username and the salted and hashed version of the password

This function will return a promise, so success and error is expressed by calling fulfill and reject (see description below).

Parameters:

- `username`: username
- `email`: email
- `password`: plain text password

Return:

This `registration` function returns a `Promise`:

- when the promise is fulfilled, it is fulfilled using the newly saved user object
- if the promise is rejected, it is rejected using an object containing a message property with a value describing the error
  - `USERNAME ALREADY EXISTS`
  - `USERNAME PASSWORD TOO SHORT`

Description:

- check if the incoming username and password are both greater than 8
  - if not, reject with `{ message: 'USERNAME PASSWORD TOO SHORT' }`
- check if user with same username already exists
  - implement this using mongoose: `User.find` or `User.findOne`
  - if not, reject with `{ message: 'USERNAME ALREADY EXISTS' }`
- salt and hash using bcrypt's sync functions (https://www.npmjs.com/package/bcryptjs#usage---sync) (note that the result will contain both the salt and hash; this value can be used directly as the value for the `password` field)
- using the incoming username and email… along with the salt and hash:
  - instantiate a new `User` object
  - set the `username` and `email` to whatever was passed in as arguments, and the `password` should be set to the salt and hash generated
  - call `save`
- if registration is successful, fulfill with the newly created user

Alternative

The starter code can be cleaned up a little bit by letting the `register` function implicitly return a `Promise`. To do this:

- modify the code so that `register` is `async` (see template below)
- rather than reject, throw an object with the appropriate message: `throw({message: yourMessage})`
- rather than fulfill, return the new user

The returned value will be wrapped in a promise, and consequently can be awaited. The thrown errors will end up as exceptions, and can be caught in other parts of the code.

```
const register = async (username, email, password) => {

  // TODO: implement registration
  // * check if username and password are both greater than 8
  //   * if not, throw { message: 'USERNAME PASSWORD TOO SHORT' }
  // * check if user with same username already exists
  //   * if not, throw { message: 'USERNAME ALREADY EXISTS' }
  // * salt and hash using bcrypt's sync functions
  //   * https://www.npmjs.com/package/bcryptjs#usage---sync
  // * if registration is successfull, return the newly created user
  // return user;
};
```

Testing:

To test the database setup and registration:

1. start your server by going into the `src` directory and running `app.mjs` with `node` or `nodemon`
2. go to `http://localhost:3000/`
3. click on registration
4. fill out the registration form
5. after submitting, check the following:
    - the resulting page that the browser is redirected to contains `Logged in as: username` in the header
    - use `mongosh` to connect to the `hw06` database
    - check if the user that you just registered with is present using `db.users.find()`
    - verify that the password field contains a hash rather than the original pain text password
6. note that the starter code automatically starts a new authenticated session on successful registration
7. you should be able to add articles (links)
8. try adding one through the interface
9. again, using `mongosh`, query for articles and verify that the created article references your user

Complete Login

Finish the implementation of `login(username, password)` in `src/auth.mjs`: This function will:

- check if the given username and password match with the stored values in the database

This function will return a promise, so success and error is expressed by calling fulfill and reject (see description below).

Parameters:

- `username`: username
- `password`: plain text password

Return:

This `login` function returns a `Promise`:

- when the promise is fulfilled, it is fulfilled using the object that matches the incoming username and password hash
- if the promise is rejected, it is rejected using an object containing a message property with a value describing the error
    - `USER NOT FOUND`
    - `PASSWORDS DO NOT MATCH`

Description:

- find a user with a matching username using `User.find` or `User.findOne`
- if username isn't found, call `reject` with `{ message: "USER NOT FOUND" }`
- use bcrypt's sync functions to check if passwords match (https://www.npmjs.com/package/bcryptjs#usage---sync)
- if passwords (that is, the hash and the hashed incoming password) don't match, `reject` with `{ message: "PASSWORDS DO NOT MATCH" }`
- however, if there is a match, call `fulfill` with found user

Alternative

The starter code can be cleaned up a little bit by letting the `login` function implicitly return a `Promise`. To do this:

- modify the code so that `login` is `async` (see template below)
- rather than reject, throw an object with the appropriate message: `throw({message: yourMessage})`
- rather than fulfill, return a value (the found user)

The returned value will be wrapped in a promise, and consequently can be awaited. The thrown errors will end up as exceptions, and can be caught in other parts of the code.

```
const login = async (username, password) => {

  // TODO: implement login
  // * find a user with a matching username
  // * if username isn't found, throw { message: "USER NOT FOUND" }
  // * use bcrypt's sync functions to check if passwords match
  // * https://www.npmjs.com/package/bcryptjs#usage---sync
  // * if passwords mismatch, throw{ message: "PASSWORDS DO NOT MATCH" }
  // * however, if passwords match, return found user

  // return user;
};
```

Testing

To test login:

1. stop and start your server to clear any active sessions
   - make sure you start your from within the `src` directory
2. use your browser to go to `http://localhost:3000/`
3. click on login
4. fill out the login form using the information from the user that you created for the previous registration test
5. after submitting, check the following:
   - the resulting page that the browser is redirected to contains `Logged in as: username` in the header
   - open an incognito window and go to `http://localhost:3000` to verify that the header is not present (instead, links to register and login are shown)

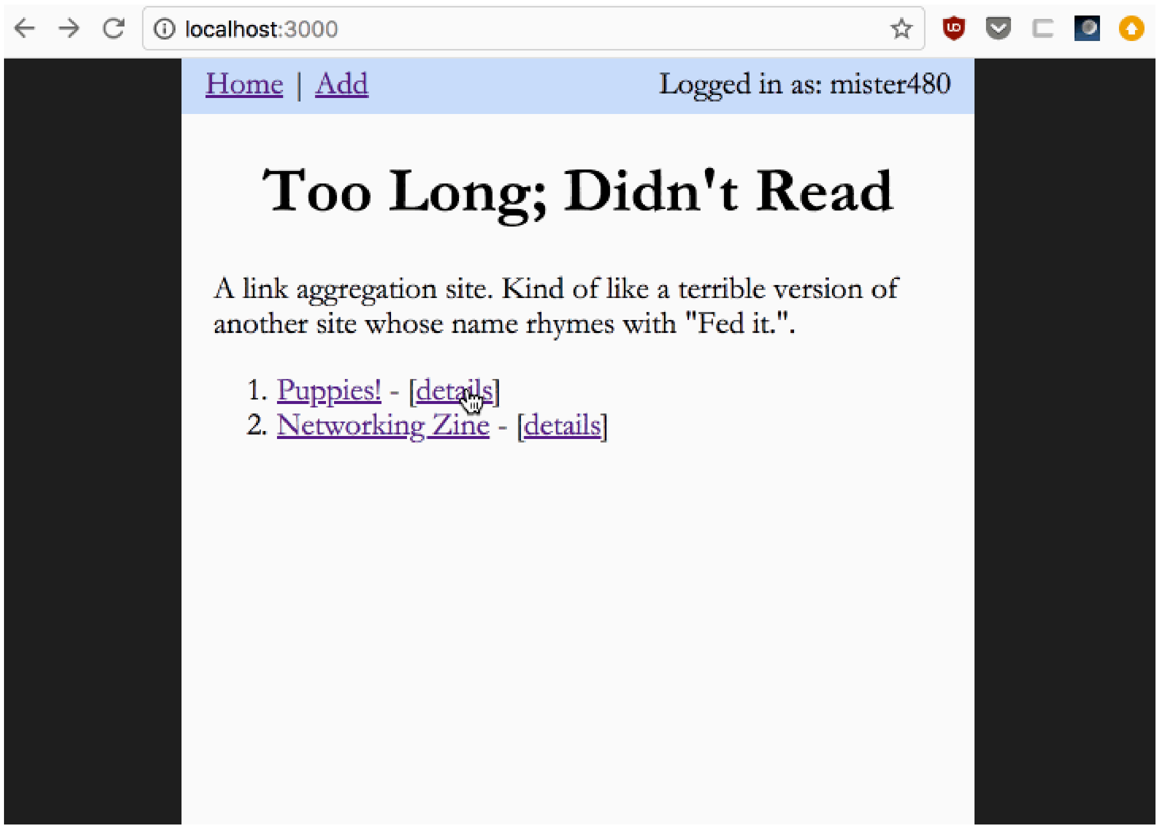
Implement Article Detail Page

Finally, implement the article detail page, `/article/the-article-slug`. It should show all of the fields of an article, including:

- title
- url (this should actually link to the url displayed)
- username of the user that added the article

- description

You can see an example of this in the image below:



localhost:3000

Home | Add   Logged in as: mister480

Too Long; Didn't Read

A link aggregation site. Kind of like a terrible version of another site whose name rhymes with "Fed it.".

1. Puppies! - [details]
2. Networking Zine - [details]

To implement:

- create a route handler in `src/app.mjs` that will respond to requests to `/article/name-of-article-as-slug`
    - `name-of-article-as-slug` will vary based on the *actual* path (
    - remember to use `:paramName` in the variable part of url, which can be accessed via `req.params.paramName`
- find the article using the `findOne` using the param (specifically, the slug) extracted from the url
- use `populate` to get the related user info
- when found, render the article-detail handlebars file
- set the template variable to the found user when rendering

Part 2: Blackjack

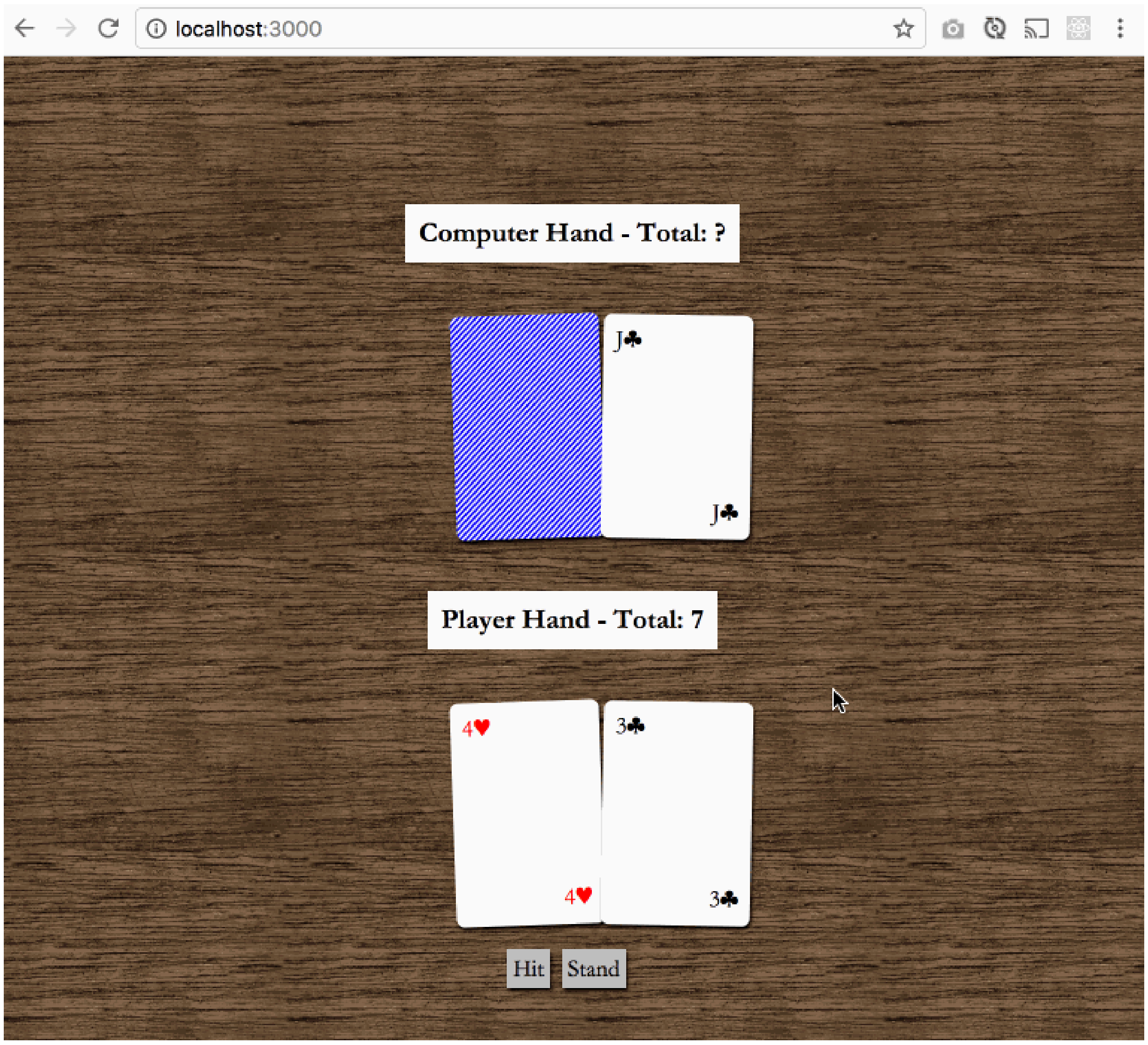
Blackjack

Create a two-player (user vs computer) client-side card game. You can use the existing express application from the previous part to serve the html, css, and client side JavaScript. The card game will be **a single hand** of blackjack. If you're unfamiliar with the rules:

- each player will try to construct a hand of cards that's equal to 21 or as close to 21 as possible, without going over
    - the sum of the numeric values of the cards determine the value of a hand
    - face cards are worth 10
    - aces are worth 1 or 11
    - the player with the hand closest to (or equal to) 21 wins
    - ties are possible

- each player is dealt 2 cards from a 52 card deck, with each card representing some numeric value
- in our version, the initial cards are dealt to the computer and user in an alternating fashion, with the computer being dealt the 1st card:
    - the computer is dealt one card, and the user is dealt another card
    - this repeats one more time so that both the user and computer have 2 cards each
- once the initial two cards are dealt, the user can choose to be dealt more cards ("hit") or stop being dealt cards ("stand")
    - if the user's hand ends up exceeding 21, then the user automatically loses
    - if the user chooses to "stand" (to stop being dealt cards), then the computer can choose to continually "hit" or "stand"
- once both players have either chosen to stand or have a hand with a total that's more than 21 ("bust"), the hands are compared
    - the player with the hand that's closest to 21 without going over wins
    - again ties are possible (either same total, or both player "bust")

Here's an example of what your game may look like:



Blackjack Game Requirements

There are three main features required for the game:

1. An initial screen where the user can enter a series of comma separated face values (2 - 10, J, Q, K, A) that will set the cards of the top of the deck to those values
    - Consequently, if a user puts in 2,3,4,5 …. 2 will be on the top of the deck, 3 next, etc.
    - So when hands are dealt, the computer will be dealt a 2 and a 4, and the player will be dealt a 3 and a 5
2. A client-side JavaScript implementation of one hand of blackjack
3. A small amount of styling:
    - the cards in a hand should appear adjacent to each other
    - cards should have a height and a width
    - *some effort* should be made to make the game appear visually polished

Setup and Initial Screen

In the existing express application in `src` …

1. In your express app's `public` folder verify that `css/game.css` exists
2. In your express app's `public` folder verify that `js/game.js` exists
3. from the `public` folder, open `game.html`
4. **add a script element to bring in the JavaScript file that you created in the `js` folder (remember to `defer`, use `type="module"`**
5. note that there is already a link element referencing `css/game.css`

Aside from the `script` element above, ⚠️ **you are not allowed to use any additional markup other than what's already contained, and you cannot modify the markup above**;

You must generate any additional elements you'll need with JavaScript!

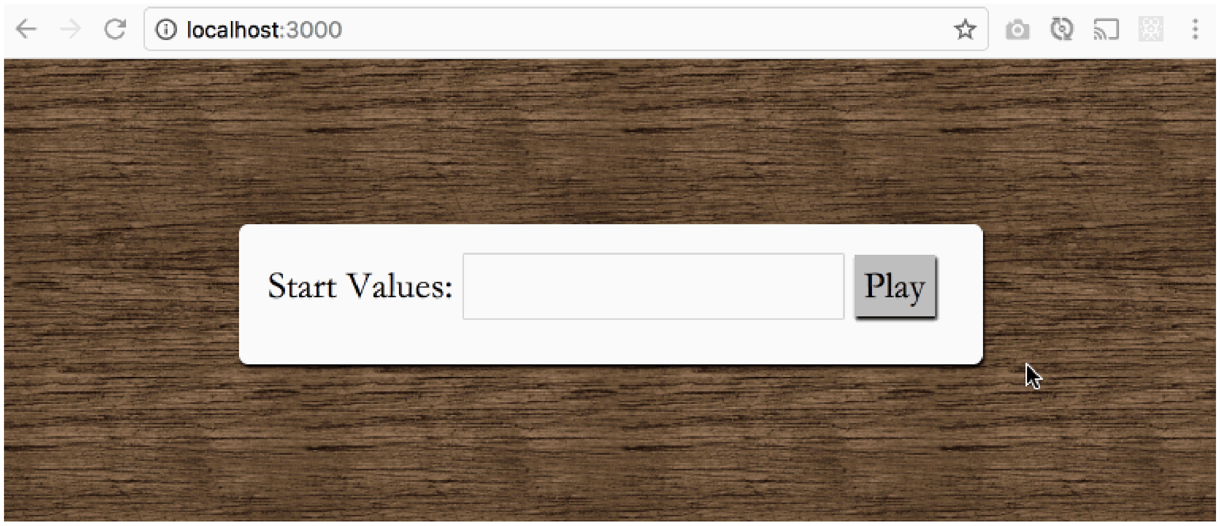
Handle the Form Submission and Modify the User Interface

Submitting the form hides the original form and distributes (deals) cards to the computer and user. If the user entered a list of card faces in the form before submitting, then those cards will be dealt first from the top of the deck.

Start off by handling the submit button on the form and making the form disappear. Use a combination of `document.querySelector` and `addEventListener` to allow the submit button on the form to be pressed:

- see the slides on `document.querySelector` (../slides/19/js-css.html#/6) or the mdn documentation (https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelector)
- check out the slides on events (../slides/19/events.html#/)
- …along with mdn's documentation on addEventListener (https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener), click (https://developer.mozilla.org/en-US/docs/Web/Events/click), and DOMContentLoaded (https://developer.mozilla.org/en-US/docs/Web/Events/DOMContentLoaded)
- because the button in the form is a submit button, it'll *actually* make a GET or POST request when pressed; we don't want this to happen, so we'll use preventDefault (../slides/19/events.html#/15) to stop the default action from occurring on a click event
- and lastly, the slides (../slides/19/css.html#/56) on classList (../slides/19/css.html#/7) will show you how to add, remove and toggle css classes on elements
- Remember, you'll need to put all of your DOM dependant JavaScript in a `DOMContentLoaded` listener (which you already did above)
- and, of course, you'll need to add a `click` event listener for your submit button
- within the callback to your `click` event listener:

    1. make sure to call `preventDefault` (see the slides on events (slides/19/events.html#/15)) to stop the form from submitting
    2. create and apply the appropriate CSS classes to get rid of the *form* (do this with styles, there's no need to remove the element) to make room for displaying cards (in addition to the slides above, here's the mdn docs on classList (https://developer.mozilla.org/en-US/docs/Web/API/Element/classList))
- Note that `this` within the callback passed to `addEventListener` will refer to the element that was clicked on, as long as the function you use is not an arrow function
    - For example….
    - Using this markup: `<div id="clicker">Click Me</div>`
    - Using this JavaScript: `document.querySelector('#clicker').addEventListener('click', myCallback)`
    - `this` within `function myCallback` will refer to the original div element clicked on!



Generate a Deck of Cards and Deal Cards

Once you've gotten rid of the form, generate a deck of cards and distribute a hand to the user and the computer (optionally setting the top cards to the values set in the form above).

- Note that there was a text field in the original form - this can be used to set the cards at the top of the deck
    - If the player enters a value in this field, then the cards on top of the deck are set to the sequence inputted
    - The input should be a comma separated list of characters and/or numbers (for example `K,2,3,7`), the suit of the cards can be any suit (in the example below, all of the specified cards were given a default suit of diamonds)
    - No validation is required (assume that the user puts in valid input or no input)
    - You can retrieve the user input from the text field by using the `value` property on the form element that contains the user input - see the mdn docs on value under HTML Input Element (https://developer.mozilla.org/en-US/docs/Web/API/HTMLInputElement)
- **The following should done programmatically via client side JavaScript** (the graphical part will be specified in the next section)
- Generate a deck of 52 shuffled cards (perhaps an Array of objects?)
- If there were card faces that were entered in the form above…
    - add those cards to the top of the deck (again, you can just pick a suit; it's the faces that matter)
    - these cards will be dealt first, alternating between the computer and the user

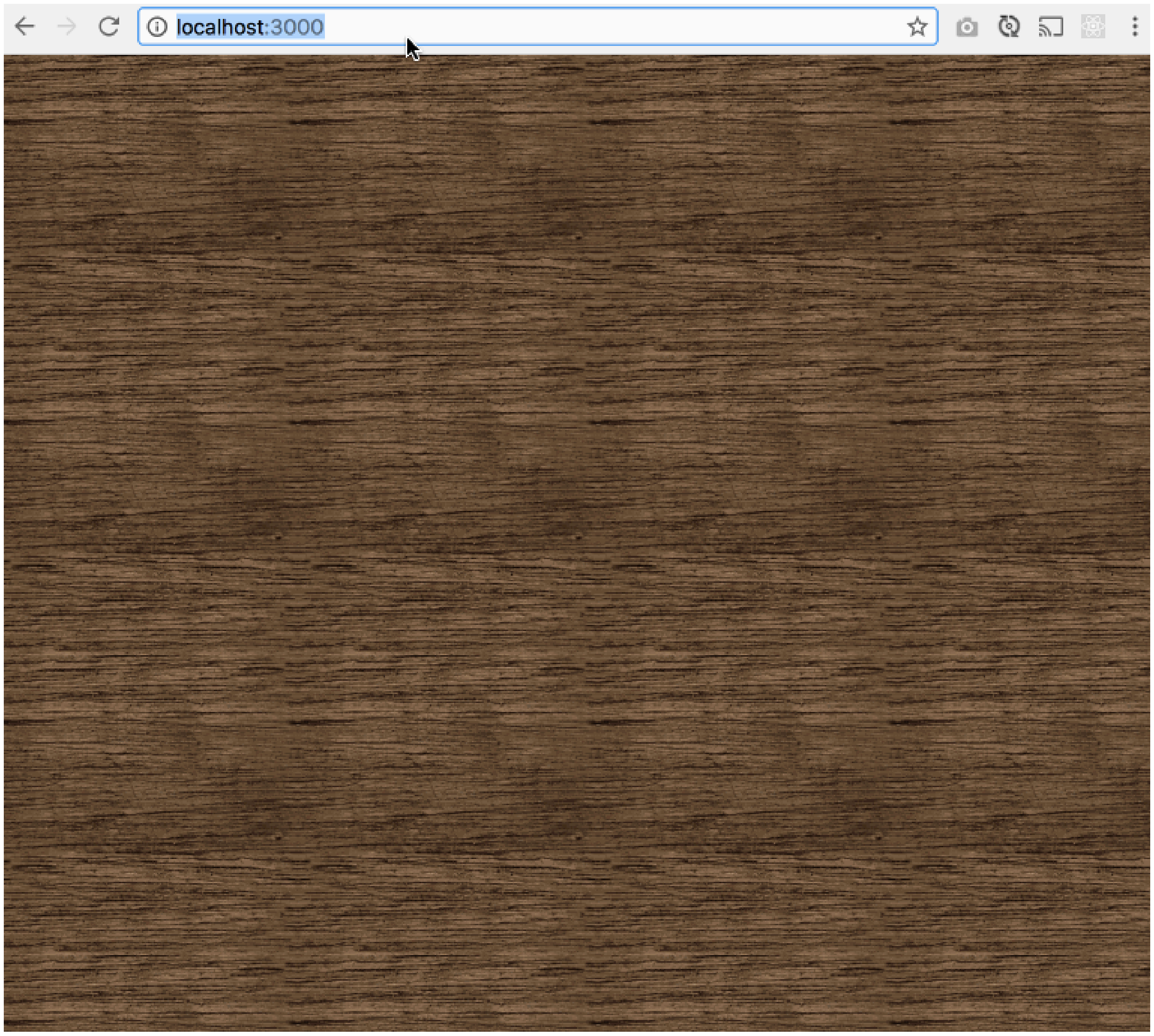
    - for example, if the cards 2, 5, 2, 5, J, Q, K, then:
        - the computer would get a 2 and 2
        - the user would get a 5 and 5
        - the next cards dealt (from pressing the "Hit" button) would be J, Q, and K
- Deal the cards - alternate between the computer and player
    - the computer gets the first and third card
    - the player gets the second card and fourth card

Display the Cards and User Interface

Based on the cards dealt, add elements to the user interface to represent both the user's and computer's hands (set of cards).

- Create DOM elements to represent cards
    - Check out the slides and mdn documentation on modifying and creating elements
        - slides on adding and removing elements (../slides/18/modifying-creating.html#/1)
        - slides on changing an elements text (../slides/18/modifying-creating.html#/5)
        - slides on creating elements (../slides/18/modifying-creating.html#/7)
        - mdn docs on `createElement` (https://developer.mozilla.org/en-US/docs/Web/API/Document/createElement)
        - mdn docs on `createTextNode` (https://developer.mozilla.org/en-US/docs/Web/API/Document/createTextNode)
        - mdn docs on `appendChild` (https://developer.mozilla.org/en-US/docs/Web/API/Node/appendChild)
    - You should consider writing some helper functions for creating and adding elements
    - For example, one of our books introduces a function to create an element and add children immediately (http://eloquentjavascript.net/13_dom.html#c_Mnkp5ioh9C)
    - Programmatically create elements to represent cards
    - All of the computer's cards should appear next to each other horizontally - one of the cards should not be revealed
    - All of the user's cards should appear next to each other horizontally
    - All of the cards should have a width and height
    - Hint: to lay out elements adjacent to each other and still maintain a width and height, you can use …
        - `display: inline-block`
        - `display: flex`
        - a `table`
        - or `float` your elements
- Calculate the hand total for both the computer and the user
    - Remember that aces can be 1 or 11
    - Make sure that your algorithm optimizes the values of aces so that the hand total is as close to 21 as possible without going over
- Create two elements to display the computer and user total
    - the computer total should be displayed as ?
    - the user total should reflect the total in the user's hand
- Create two buttons, `Hit` and `Stand`
- See an example interaction below:
- THE STYLING IN THE ANIMATED GIFS IS JUST FOR EMBELLISHMENT; YOUR STYLING DOES NOT HAVE TO MATCH

- (but you should put in some effort to make the cards *card-like*)



Hit and Stand

Once user's hand and the computer's hand are displayed after the initial deal, the user can now decide whether they want more cards or stay with the cards that they have. The previous section added buttons (Hit and Stand), but now you'll have to add event listeners to those buttons.

1. pressing `Hit` should deal the next card from the deck
    - the next card should be moved from the deck to the user's hand
    - an element should be created to add the card to the user's hand in the user interface
    - if hitting makes a user's total go over 21:
        - then the user's turn ends immediately
        - … and they lose the hand
        - (the computer does not even need to decided to hit or stand)
2. pressing `Stand` should end the user's turn and allow the computer to Hit or Stand
    - the computer's strategy is your discretion
    - though the computer must hit in some situations or stand in others
    - an easy strategy is to always hit if a hand total is underneath a threshold, but stand if it's eqaul to or above that threshold