Clark Bains
101149052
Connect 4
Testing Requirements
I will no longer provide local build instructions beyond npm i, then npm run build, then npm run deploy:local.

For cloud builds, I have two options:
My openstack instance lives at thebest2406projectinthe.cloud (134.117.131.57). To connect to it, you can use the command `ssh -L 3000:localhost:3000 student@thebest2406projectinthe.cloud` The password is `crackThisJohn` . This will open a port tunnel to your own computer on port 3000. From there you can visit http://local.thebest2406projectinthe.cloud:3000/ (Dns record is just configured to go to localhost).
On openstack, I run it through a docker image. It should already be running for you, but to check, run `~/status.` If it is running you will see some output. If its not running, `cd connect4` and then you can run the commands to start (`~/start`), stop(`~/stop`) and view logs (`~/logs`).
To reset the database, follow the instructions to stop it, run this command to restore the database (`~/reset`) and then run the command to start it.
These are all just shell scripts to run the correct docker commands for you.
My Self Hosted Instance
Visit https://connect4.clarkbains.com , and it will show the most current version of my code. Note that it is public so things might not be exactly the same as in the openstack instance, but I will try and keep them as close as possible.

Seeded Data
I have a few users by default, with friends set up, and a game or two.

Username	Password	Privacy Setting	Friends With
Username1	Password1	Private	Username2,Username3
Username2	Password2	Public	Username1
Username3	Password3	Private	Username1,Username5 (pending)
Username4	Password4	Private	
Username5	Password5	Public	Username3(pending)

Testing Instructions
I did my best to make my site fairly intuitive and error proof, but here's a quick walk through to test functionality.

Account Creation:
Start by visiting the server, going to the log in tab and click on the create account button. Enter a Name, username, and email. Click create account. You will be taken back to the login page. Click on reset password. Enter the username and click on Get JWT(Dev). This prefills information that would have otherwise been sent over email, were I to use email validation. Enter your password and press "Set Password."

Logging In:
To test you can use the account you just created, and another one from the seeded data section down below. You have to click login after filling out your password.

Friends:
Ensure you have two accounts signed in. Use an incognito browser tab for one user, and a regular tab for the other. Click on the edit button in each users profile page, and click on the globe/lock to change visibility. Ensure one account is set public (globe), and one is set private (lock). Click save again to save your changes.
From the private users account, go to the Search for user tab, and type their username (You can test case insensitivity as well). When they show up, you can click on them, and click on request friend. If you wish, you can try it the other way, searching from the public user for the private user. You should find no results.
In the public users window, you should see a slight flicker as it updates and add the new pending friend request. This is websockets in play. Click on the private user under incoming requests, and then click on accept request. You should see them both update and list as each others online friend. If you wish, you can log out of one, and refresh the others profile page. You should see them marked as offline. Log back in. If you click on a friend, and then unfriend them, you should see that they are no longer friends.

Starting A Game:
You have two options to test this, both are different interfaces but run nearly the exact same code.
- You can, from the private user, go to the public users profile, and click on start game.
- You can, from one users account, create a request game with a random user, and with the others, join that as a "random user"
If you chose the **first option** you will first have to select a privacy setting. Make sure you do this on the left side. While private works, for testing purposes, chose friends only or public. Click Send Request. Then the public user will have the chance to accept or deny the request. From the public users window, first click deny, you should see it disappear from pending, and instead show in the declined section of requests. The private users profile page will show it as declined as well. The public user will still have an option to accept, so click on that, and it should go back into pending. Click on Initialize game from either public/private window

If you chose the **second option**, in one users account, click on select game, and try to join a friends only or public game with a random user. It should not succeed as there are no current games for this user to join. When it fails, it should show a second menu. This allows you to create a game request rather than trying to join an existing one. You can set the game name, and then click on Create $privacy game with random user. When it takes you to your profile, you should see the second player for the match you created is "No Player Yet". In your other user account, go to select game, and try to join a game with the same public/friends only setting that you used in the previous step. You should see if has now matched the two users for this game. Click on Initialize game from either window

Playing a game
Both players should now have an option to View the game or resign. Click on play game for both users. You should now be on the play game page. Note that this page makes use of websockets, and does not need to be reloaded for you to see changes. It should say who's turn it is at the top of the page. Send a message as each user, and play a piece or two by clicking on the grid. If it is your turn, and you go back to your own profile, you should see the game name is now red, bold, and says "Your Turn" at the end. You can optionally go back and come back to the game to verify that messages are viewable.

Spectating a game
You will need to login with a third session. If you set the game to friends only, you will need to add one of the two users as friends and then you will be able to see the game. Navigate to the profiles of one of the two people playing the game, and then you can click on spectate game. You should see that users name show up in the spectator list on all clients. You can click the button at the top to change modes and view the historical moves in the game.

Viewing Scores
In one of the accounts that way playing in the Playing A Game section, click resign. It should update the scores of both users. You Should also see it show under Finished Games, with the correct winner, and you should be able to revisit the game and look at all the moves.

Core Requires Summary (If you don't want to read about the implementations)

- New Accounts/Password Resets
  - Uniqueness Constraints on usernames
- Sessions in front end
- - Search For Users
- Friends
  - --Search for users to friend
  - See incoming/outgoing friend requests
  - Accept/Deny requests
  - Online/Offline friends
  - Navigate to friend profiles
- My Profile
  - Edit your profile, including public/private status
  - View games you played/are playing
  - --View Summary Statistics
- Others Profiles
  - Hide if not friends/public
  - View games you should be able to see
- Playing Games
  - Can resume games
  - Can forfeit games
  - In Game Chat
    - Message History
- Creating Games
  - Can create with friends
  - Can join with a random user also looking to join

Core Requirements – Implemented

New Accounts/Password Resets
New account creation is handled much like it would be in a proper system. While I don't enforce having a valid email, the account creation page takes your email and name and username. No Password. In order to set your password, you would have to go back to the login page and click on reset password. Enter your username and click get JWT. A JWT will be created for you that allows you to reset your password. This could easily be extended to emailing a message to the account with a link that includes this JWT, allowing a user to reset their password. Uniqueness constraints are handled by the database.

Sessions In Front End
I did my entire project using a front end framework. This means that I had to create an extensive REST API for every action in my system, and deal with authentication for that. Since REST is supposed to be stateless, instead of using express-session, I implemented my own system using JWTs. I have private routes separate from public routes, and the middleware for private routes checks the validity of the JWT, and then will look up the user and put it in res.locals.user. It then regenerates a new JWT and sends it back in the Set-Cookie header, so that as long as there is an API request every hour, you stay logged in.

My Profiles
I have several API requests that get run so that I can display the my profile page. Like every other user, I have the basic info, but then I also have one for friends, friend requests, and all the games. Online and offline functionality was added into the authentication middleware I wrote, it updates an in memory map between userids and last seen times, and then there are also methods to delete this info when a user logs out, and a method to get the time since last seen for a user. This allows me to attach a lastseen to every user, and lets the front end filter users into online/offline.

Other Profiles
Other users profiles use the same front end page as your profile, but the server includes flags that hide elements of the UI, and change if they are even requested at all. IE, friends of a user aren't requested when you aren't viewing your own profile. All the filtering of games you can see is done by the server, you ask it for a users games, and it will get you ones that you should be able to see, and add flags and information that allows the front end to categorize them into ones pending acceptances, played with you, played with others, and denied.

Playing Games
Games work fine, and use socket.io to make the experience a bit better for the user. Resign functionality is implemented on your own profile page. In game chat operates over web sockets to emit messages from the server, but works via ajax post to send messages into the system. This will be explained under the design decision section.

Creating Games
Creating games can be done two ways. Initially I make a match, which is basically a pending game. From there both users have an option to accept the request. If a match is made with one user (ie creating a $privacy Game with Random User) then one of the userid is null. When you request to join a public game, I run a fancy SQL query that cross joins and groups in order to efficiently find a match for them. It then add them.

Core Requirements – Not Implemented
My API documentation is probably slightly lacking based on what is expected. My public API cannot return if a game is resigned or not, since I did not store that information. I simply set a user as a winner.

Extensions to the Project
I made use of **websockets,** which allow for communication from the server to the client (and vice versa). This helped with my overall goal of not making the site annoying to use, since it let me dynamically update the page whenever I need. Socket.io also helps make my program more idiot-proof,

since unlike plain web sockets, it will fall back to long polling, which means it will work in a variety of network environments and browsers, even if they don't natively support sockets.

Also contributing to my "don't annoy the user" philosophy is my **front end framework**. I used Aurelia, which isn't too popular, but lets me use templates, npm packages, and dynamically update the page by just changing variables in models. This way I can run some AJAX in the front end and then have the page update immediately. This also means it is fairly **developer friendly.** Even if I don't document every internal API route does, its fairly clear what most things do.

The site is also **responsive**. Most things will work on smaller screens. I achieved this by using bootstrap, which is extensively tested and used, so it works on most devices.

I've also configured the entire project as a **docker project**. This lets anybody interested in running it run it quickly and easily. If I were to publish on docker hub, running my entire project would be as simple as running one docker command. This also provides anyone hosting it with a better sense of security, since even if they manage to exploit the program, they would have to break out of the docker sandbox. My openstack instance is currently running a docker build of the project.

I've also set up a fairly primitive **CD system**. Every time I push my project to github, I have my server receive the webhook, pull the code, run a new build and auto deploy it. The deployed version of this is at https://connect4.clarkbains.com. I've also set up **HTTPS** on it, so any data sent to it is secure, at least in transit. It sits in public view in my house with my roommates so I can't promise anything about security of data at rest :).

I wrote my API using **TypeScript(TS)**, which in theory makes it less prone to bugs, since I know what type every variable is. Typescript is just javascript with type annotations.

I've created a rather interesting **database system** for it. I opted to use SQL since I am already familiar with SQL and thought it would be easier for me to do. Instead of using an ODM for SQLite (mongoose is an ODM for mongo), I ended up **creating my own ODM**. The design of it is questionable, however it allows my to instantiate a model, and then call the insert() method on it, which will asynchronously generate the SQL for that specific operation (using parameterized queries, of course) and then run it. I also made it support delete, and select methods, which will take a semi – fully complete model, and either return all of the full models that match it in the database, or delete them. It also supports update, which works a bit differently, by finding the primary keys for the specific model, and then using those in the where clause of the query. That way so long as you pass the objects id, and the fields you want to update, it should work. I also **wrote a utility to convert TypeScript models into SQL (/tools/generateSql.js)**. (Even though SQL is statically typed, I can do this since TS classes include both instance variables and their types). I do this for the SQL CREATE TABLE commands. This way I can rapidly update my schemas, and then get the new SQL to create it.

Game Design

As stupid as it may seem, 99% of the code is set up so you can do >=2 player connect4 matches. There is even code to set the size of the board game. I never ended up implementing a front end for this, but it should work with minor tweaks to the code. The original plan was to have like 10 players on a 20x20 board, and have a random player appear the same colour as the background for every person, so you would never really know if you were playing on a blank spot or not. I realized how dumb of an idea this was so its why you may see weird code specifying sizes of the board and using a whole table of game players to I could do a one-to-many relationship between games and players.

Good Design Decisions

Route Security
Since I used a front end framework, I had a lot of API routes to do privileged things, like modify your own account, making moves, and managing friends and resignations. To the best of my ability, I hardened all of these routes against the attacks I could think of while writing. I'm sure there are still flaws, but say user1 were to manually PATCH the /user route, using user2's id, they wouldn't be able to since my server would check to make the the JWT from the user making the request has the permissions (ie, they are the same user) to modify the user in the request. I also check for things like sending undefined fields in hopes of getting my sql update commands to accidentally modify the whole DB. There is still work to do, but this step required a lot of thought, and accounts for a large amount of code.

Route Treeing
I'm not sure what to actually call this, but I made my server dynamically load all the files containing routes. Every route file has a reference to its parent route. So user.ts("/user") can be a child of private.ts("/private") which can be a child of API.ts("/api"). This allows me to quickly rearrange my API if needed, and makes the location of code very clear. If I wanted to change the route GET /api/private/user/me to be /api/priv/user/me, I would only have to change one string in private.ts, and all the child routes would update. I also took advantage of this to make sure that the API structure matches the folder structure on disk. This can be found in connect4/src/api/routes.

Event Bus
This is becoming a common theme at this point, but I wrote my websocket code before I knew about socket.io rooms. I ended up coming up with a really neat implementation, which basically allows anyone to connect to websockets, but stores the connection in a separate area. My implementation requires the client to send a regular POST request with their ID to whatever route I've set up for that type of event. The handler function is then able to check to make sure the client has sufficient permissions to subscribe to that event. If not, then it won't set the event bus to transmit events to it. If the user sending the API request with their socket id has adequate permissions, then it will call a method in the event bus to subscribe the user to receive those events. This approach lets me move all my permission checking logic into the specific file that handles those events, which makes it cleaner. In terms of handling the actual sockets, I just need to instantiate the socket.io class and pass my event bus any new sockets that come into the system.

Front End Framework
Using my front end framework, Aurelia, puts my project in a position where I could make it much more scaleable. Currently I have my express server serving the static files for the entire website, but it could easily be moved to AWS S3 or something. If I were to do that, then my express server would just need to handle doing database queries and returning the relatively small API responses.

Bootstrap
I'm going to preface this by saying I am very, very bad at css. I know how it all works, but I can't get things to do what I want. By using bootstrap i saved your eyes from the atrocity you would have seen otherwise, and made my site much more responsive than it would have been otherwise.

Asynchronous Code
Most of my code is written using asynchronous code with promises, which makes it pretty nice to work around with. I use an asynchronous wrapper in all my API routes which handles errors fairly well, and then my client has a gateway file which is just a class that returns promises for a bunch of various api calls. This makes the client a lot cleaner than having a bunch of XMLHttpRequests, and makes the server code readable, since I use await everywhere.

Bad Design Decisions
I'm going to preface this saying that most of the things I include in this section were written between 11pm and about 6 am, which is exactly when I tend to not be thinking super clearly.

The Custom ODM – Really Bad
The worst offender in this project is probably my ODM. It's a very neat chunk of code, but it's design is awful. Any time you want to do any database operation, it requires the database to operate on passed as a parameter, despite all of my code using the same SQLite connection. In terms of the actual code in it, the code for update is rather bad, with it asking for a description of the table *every* time I call update. It does this so it can decide which parts of the partial model its given go in the where clause, and which bits get updated. For those that don't know, a SQL update command might look like UPDATE Users set name="foo" WHERE userid=5. If I just call new DatabaseUser({userid:5, name:"foo"}).update(), it needs to figure out which side of the "WHERE" it goes on. This should have been cached, but thankfully the update method isn't used a bunch. Doing raw SQL queries if you need to do something nonstandard, like counting, or cross table joins, is also functional but really ugly in code.

Some of my code surrounding the ODM is really bad as well. To log in a user that has played about 10 games, about 50 Database queries get run. This could lead to slowdowns as the user base scaled up.

If I had to redo it, I would use a real ODM, or do the entire thing with raw SQL queries. For the latter, I could write it more efficiently, at the expense of being a bit harder to add changes to the code.

Data Usage – Fairly Bad
Particularly for GET /API/private/user/:userid/games, I currently don't have it paginated. I should, since it requires a lot of queries, and the returned data could rapidly grow for an avid user of the site. On my API used by my site (not the public one), I have very little filtering built in, so in some cases the server returns a bunch of data that might not even by used by the client, like IDs of friend requests. I could optimize this a bunch to save on data usage and server bandwidth usage. My implementation of Socket.io doesn't allow for great receiving of events from the client to the server, and as a result of that, my sockets are used mostly to signify that there is new data, and the onus is on the client to make an AJAX request to get the new data. By just sending the data I could have used a lot less bandwidth, at the expense of requiring more code. This also causes my page to flicker sometimes, if it is getting new data, and it clears the whole page so it can redraw the new data.

Internal API Responses - Meh
As time went on, I got progressively more lazy, and stopped creating custom models for specific sets of data. Most of my code written near the end for my internal API sends a GenericError class, which might have a string with a bit of detail about the error, or a GenericResourceSuccess, which just has a property with whatever data I wanted to send. At the beginning I used specific classes for specific types of error, like UserCreationSuccessResponse. This would likely make it a bit harder to use the internal API, but it's not technically a public API, so developers shouldn't be trying to use it and debug what

the responses mean.

UI – Meh
I'm not great at UI design work, and so some things in my UI don't make too much sense if they aren't explained. The Select game page is a good example, the difference between Joining and Creating a game with a random user isn't really clear.

Another thing that isn't great is my game play view. On some monitors you have to scroll to see anything, which isn't great. It also doesn't currently show which colour is which when in spectator view with neither of the players focused on the game

Modules Used
For my front end framework, I'm going to bundle a bunch of them under one name, **Aurelia**. It involves over a dozen different modules, but they mostly are just used to add all the functionality for aurelia, the framework I used. It let me show things to the user without having to reload, or do a bunch of ugly DOM element creation. It also removed the burden of server side rendering off of the API, as discussed in the Good Design decisions section

I used **Aurelia-Notifier** purely for sending nice notifications to the user that can be clicked on to be dismissed. I could have written this myself, but it was beyond the scope of the project enough that it would have little grade benefit over using the standard alert() method, so I used this instead.

On my server, I used **bcrypt** for hashing and password security. Without this I would have to write my own cryptographic hash, which I am not qualified to do

I used **SQLite** for database interactions with my SQLite db. This was needed to interact with my database for what I hope are fairly obvious reason. My ODM took care of promisifying everything, so I didn't have to worry about callback hell with it.

I used **Socket.io** for sending events from the server to the clients. Without it I would have to write my own implementation, which would probably lack the long-polling fallback, or just be implemented as short-polling.

My Favourite Feature(s)
I have two systems that I really like.
My CD system, which will build and deploy my program using docker every time I commit to master on my github repository. While its not really a part of the program, I like getting a discord notification every time I commit, where it shows the build progress, and then I can instantly share my updated version with friends.

My all-ajax design. Through testing, I might have called for you to refresh the page by switching to one tab and back to another. But the site itself never needs to be refreshed, it does it automatically where it counts, through websockets (which is most of the place). It was a lot of work to do, having to make and secure all the internal API routes

API Docs
GET /* (Express Static files)
GET /api/private/logout (Gets a blank cookie to overwrite jwt cookie from login)

GET /api/private/loginstatus (Checks validity of JWT)
GET /api/private/user/me (Get your own account)
GET /api/private/user/:userid (Get a specific user)
PATCH /api/private/user/me (Update my account)
DELETE /api/private/user/me (Delete my account)
GET /api/private/user/me/friends (Get my friends)
DELETE /api/private/user/me/friends/:userid (Remove a friend)
GET /api/private/user/me/friendrequests (Get my friend requests)
POST /api/private/user/:userid/friendrequests (Send friend request to user)
POST /api/private/user/me/friendrequests/:friendrequest/accept (Accept a friend request)
POST /api/private/user/me/friendrequests/:friendrequest/deny (Reject a friend request)
GET /api/private/user/search?limit=10&term= (Search for a username)
GET /api/private/user/:userid/games (Get all the games the current user has perms to view)
GET /api/private/games/:gameid/match (Get a list of the match used to start this game)
GET /api/private/games/:gameid/players (Get a list of all the players)
POST /api/private/games/:gameid/moves/authorize (Authorize for socket for getting game move updates)
POST /api/private/games/:gameid/participants/authorize (Authorize for socket for getting participant updates)
POST /api/private/games/:gameid/messages/authorize (Authorize for socket for getting message updates)
GET /api/private/games/:gameid/participants (Get a list of people viewing the game)
GET /api/private/games/:gameid/board (Get a 2d array of the game board)
POST /api/private/games/:gameid/resign (Add your resignation to an event)
GET /api/private/games/:gameid/state (Get some state info about turn, and winner)
GET /api/private/games/:gameid/messages (Get all the messages for the game)
POST /api/private/games/:gameid/move (Add a move)
GET /api/private/games/:gameid/moves (Get all the move object)
POST /api/private/games/requests (Create a match)
POST /api/private/games/requests/join (Join a match with no other user)
POST /api/private/games/requests/:matchid/response (Accept or Reject a match request)

(The following three are from the requirements, The are kinda strange since all my data is stored with ids and they all work by usernames).
GET /api/public/games?player=&detail=&active= (Search Games)
GET /api/public/user/?name= (Search Users)
GET /api/public/user/:username (Get by Username)

POST /api/public/user/ (Create Account)
POST /api/public/user/login (Returns JWT)
POST /api/public/user/changepassword (Takes old password or password reset JWT and changes passworsd)
POST /api/public/user/sendresetemail (Returns JWT for the login currently. Real system wouldn’t do this)