Online Chess

An exploration into website design and infrastructure

Gaston Aganza and Jacob Marshall

Spring 2019

For Professor Nico

Abstract

Chess has been around as early as the 6th century (Chess Here). While the last major modification to standard chess was added in the 15th century (En Passant) (Chess Here), the modes and platforms in which people play has continued to evolve. Our project was intended to tackle the idea of creating an interactive web application to play one of the world's oldest board games.

Introduction

While it only ranks #439 on boardgamegeeks.com, chess is arguably the most popular board game in existence. Personal interests in the game along with interests in developing full stack applications lead to the idea of this project. The main goal we took into consideration when planning and developing our application was learning and improving our ability to manage and maintain a (relatively) complex software project. Because we focused on learning we had to work with several unfamiliar technologies. To comprehend this process we used documentation and self-learning resources as a guideline to direct our progress. The field of Software Engineering moves at a rapid pace, and our research enabled us to fulfill professional expectations of learn by doing.

Related Works

We are definitely not the first to implement a web based application for playing chess. Using your favorite search engine and looking for "online chess" will result in a list of several different implementations, the most popular being chess.com, which has the most robust features. While our implementation does not provide any new features that have not been previously seen on other applications - we built this application with the sole intention of learning what it takes to create a full stack application.

Audience

While the StockFish chess engine (will go into further detail on StockFish) does account for a wide skill range, the main demographic for online chess are people who are already familiar with playing chess, as this application does not necessarily teach the rules of chess. Since it is a challenge for a computer to mimic the blunders that humans encounter while playing chess, the skill range that StockFish provides does not translate well into a USCF[1] rating. Due to this

[1] United States Chess Federation: the official governing body and nonprofit 501(c)(3) organization for chess players and chess supporters in the United States. Our mission is to empower people, enrich lives, and enhance communities through chess. Our vision is that chess is recognized as an essential tool that is inclusive, benefits education and rehabilitation, and promotes recreation and friendly competition. (US Chess)

conflict we could not give a defined skillset, so online chess is best used a beginner[2] skillset and challenging enough for Grand Master's[3]. While that is all true for playing against the StockFish engine, this application will also attract friends who would like to play against one another - given that our project allows for peer to peer games.

Implementation Goals

As with most large software projects, our first steps in the implementation of our project was breaking the problem down into smaller solvable problems. Now with our project in mind we had to define a realistic scope of our project. We agreed upon the following:

Functional Requirements:

- User can create an account (unique username)
- User can create a game against an AI
- User can create a game account an opponent with an account
- User leave/join a game they have in progress
- Game prevents moves that are not possible (and shows possible moves)

Nonfunctional Requirements:

- Moves should update on all clients seamlessly (within X timeframe)
- UI should properly scale (define properly)
- Should be somewhat secure?
- Game seamlessly displays possible moves given a selected piece (seamlessly? Be more specific)

Given our requirements we then picked out the technologies that would best suite our needs. After our initial research we decided on a server-client architecture with the following technology stack:

[2] Beginner as in somewhat who understands the basics of playing the game (not just someone who understands how to move the pieces.
[3] The most distinguished title in chess, awarded by FIDE. A grandmaster is usually rated between 2500 up to 2851. (US Chess)

Front-End: ReactJs

We decided to use ReactJs as the front-end framework because it is a fairly modern framework that has a lot of community support in terms of documentation of open source modules. In addition to the support that would be available to us, ReactJs has been growing in popularity in the professional environment, so it would greatly benefit us to be familiar with the framework.

Back-End: Django (Python)

When making our decision on a back-end framework we first chose the language we would be most comfortable working with, Python. Choosing Python as our back-end language left us two viable options: Flask and Django. After doing our research, we chose Django because it is a heavier weight framework and we hoped it would have some solutions built in for us. Overall this was a very good decision, as Django seamlessly handled URL endpoints, database management, and several other things for us.

Chess Engine[4]: StockFish

From early on, we decided on using the StockFish engine. We chose StockFish mainly because it is an open sourced project (go open source community!) and due to this support we believed it would be easy to find resources. Another reason why we chose this engine was because of the multi-platform support - it can be run on Windows, MacOs, Linux, Android, and iOS.

Database: Sqlite3

Our choice for a database was fairly clear cut — we understood that we did not intend on storing a vast amount of data. We each have had experience working with a variety of databases, and we had a really good experience with the ease of setting up and maintaining a lightweight database such as Sqlite3. We understood the limitations of using Sqlite3 and decided that it fit our expectations and needs of the project we were building.

[4] Is a program that analyzes the current state of a given chess board and calculates the next best move (Chess.com)

Key Architectural Decisions

When implementing a web application, or any software for that matter, there has to be some initial architectural decisions that will be made that will drive the development of the rest of the project. This section will delve into further detail of our initial structures that shaped our project.

Front-End VS Back-End

As mentioned earlier, we decided to use two different frameworks for our application: ReactJs (front-end) and Django (back-end). This decision enabled us to implement the entities independent of each other — these entities were even run on separate domains. Unfortunately, this division of domain ultimately caused additional issues, which we will elaborate on later.

Back-End Implementation

In order to follow this isolated structure a key plugin that we used on the back-end was the 'djangorestframework'. We used this plugin to treat the back-end as a RESTful[5] application. The main purpose of the back-end in our project was to manage multiple StockFish Engine instances that could be running, managing the database, and to manage the data that is accessible to each user.

URL Patterns

The back-end was split into one main application, `chess_app`, and several sub-apps: `user`, `game`, and `move`. The chess_app application contained several URL patterns that the front end could query, and it provided the main structure of our website, guiding every request to where it needs to go. Django's method of URL redirections also allows us to break up the functionality of our website and spread the logic across the various applications.

[5] also referred to as a RESTful web service -- is based on representational state transfer (REST) technology, an architectural style and approach to communications often used in web services development. (TechTarget)

```
urlpatterns = [
    path("api-auth/", include("rest_framework.urls",
    path("game/", include("game.urls")),
    path("user/", include("user.urls")),
    path("token-auth/", obtain_jwt_token),
]
```

In the above image, you can see the URL for `game/` includes the `game` application's `urls.py` file. Below, you can see what the `game` application's `urls.py` file looked like.

```
urlpatterns = [
    path("", logic.create_chess_game, name="create_new_game"),
    path("create/", logic.create_chess_game, name="create-game"),
    path("get_moves/", logic.get_all_moves, name="get_all_moves"),
    path("make_move/", logic.make_move, name="make_move"),
    path("get_opponent_move/", logic.get_opponent_move, name="get_opponent_move"),
    path("get_current_games/", logic.get_current_games, name="get_current_games"),
    path("get_game_info/", logic.get_game_info, name="get_current_games"),
]
```

In this image, the second argument to `path()` was a specific function. Whenever the front-end queried `/game/create`, Django would call our `create_chess_game()` function. This functional style of programming made it unbelievably easy to set up new endpoints for rapid testing and development. This was by far one of the best things about Django.

Object Relational Mapping

Django has a built in ORM that makes it extremely simple to query a connected database of your choice. For this project, we decided to use a sqlite3 database. To create a new table, you can add a models.py file to any of your applications. Django would then parse any of the classes you created and convert them into a table. In the image below, you can see the model for the Moves table we created.

```
class Moves(models.Model):
    id = models.AutoField(primary_key=True)
    white_user_id = models.IntegerField()
    black_user_id = models.IntegerField()
    move_number = models.IntegerField()
    pre_move_fen = models.CharField(max_length=128)
    post_move_fen = models.CharField(max_length=128)
    move_algebraic = models.CharField(max_length=32)
    game_id = models.ForeignKey("game.Games", on_delete=models.CASCADE)
    turn = models.CharField(max_length=8)
```

This also made it easy to change tables if we wanted more or less information. If you change one of the fields in Moves, Django will automatically migrate the changes and update the table.

To create a user for example, you can just import the User class where you defined the model, and say:

```
User.objects.create(username="pnico", password="please_give_us_an_A")
```

Behind the scenes, Django would handle hashing and storing the password. It would then store the new user in the database. If the username was already taken, the query would fail and the back-end would report an error to the front-end.

Accessing the database just as easy. One of the most common database queries we made was getting the most recent move of a specific game. In the image below, we would first get all the moves belonging to a game, sort them in descending order from newest move to oldest move, and then return the first object out of that list of moves.

```
def _get_most_recent_move(game_id: int) -> Moves:
    moves = Moves.objects.filter(game_id=game_id)
    return moves.order_by("-move_number")[0]
```

Stockfish Engine

Integrating the Stockfish chess engine was relatively simple. We found an online implementation that would activate the Stockfish engine through the shell, and then We modified that to Python. We also added several helper functions for setting the difficulty, initializing the engine with a specific game board, and finding the best move possible. The hardest part about this was learning the UCI chess protocol that Stockfish uses.

Pychess

Perhaps one of the most difficult and time-consuming parts of the back-end was find a reasonable chess implementation in Python. We eventually settled on Pychess. It was actually built as a desktop app, so We had to rip out the front-end and massage the back-end to get it into a more friendly and usable module.

That wasn't the end of it though. Pychess' documentation was nearly non-existent. If we were lucky, an internal function might have one comment describing what it did, but other than that, we had to grep for specific keywords and read the code line by line to figure out what it did. To make matters worse, functions were often hundreds of lines long (the function that initialized a chess board object was 283 lines long). Whoever made Pychess has clearly never heard of function decomposition. Magic numbers were everywhere, and we could hardly tell what some triple nested for-loop was trying to accomplish.

Bad design aside, Pychess actually worked very well. Once we figured out how to initialize a new chess game and move pieces, things went smoothly.

Making Moves

When the front-end would query the back-end to make a move, we would do several things:

1. Load the most recent game board state from the database into a Board object
2. Get the piece at the location the user selected and move it to the location they specified.
3. Record all the pieces that were moved, in the case of castling.
4. Check if the move resulted in a checkmake
5. Return the moved pieces to the front-end.

Playing Against the Computer

When querying the back-end for a computer move, we followed many of the same steps for making a user move:

1. Load the most recent game board state from the database into a Board object
2. Set the stockfish engine to the user-set difficulty
3. Feed the board representation into Stockfish and get the best available move
4. Convert Stockfish's answer (in standard algebraic notation) to a Pychess Move object
5. Apply the move to the board
6. Record all the pieces that were moved, in the case of castling.
7. Return the moved pieces to the front-end.

Multiplayer

Our multiplayer was hacky, to say the least. We struggled with implementing real-time multiplayer, and created our own system of communication. Our multiplayer is a polling system. When the front-end detects that it is not your turn, it queries the back-end for the opponents move. The function will do this:

1. Query the database and see if the opponent made move yet
    a. If not, wait 0.5 seconds and go back to step 1
    b. If so, gather that information and return it to the waiting user

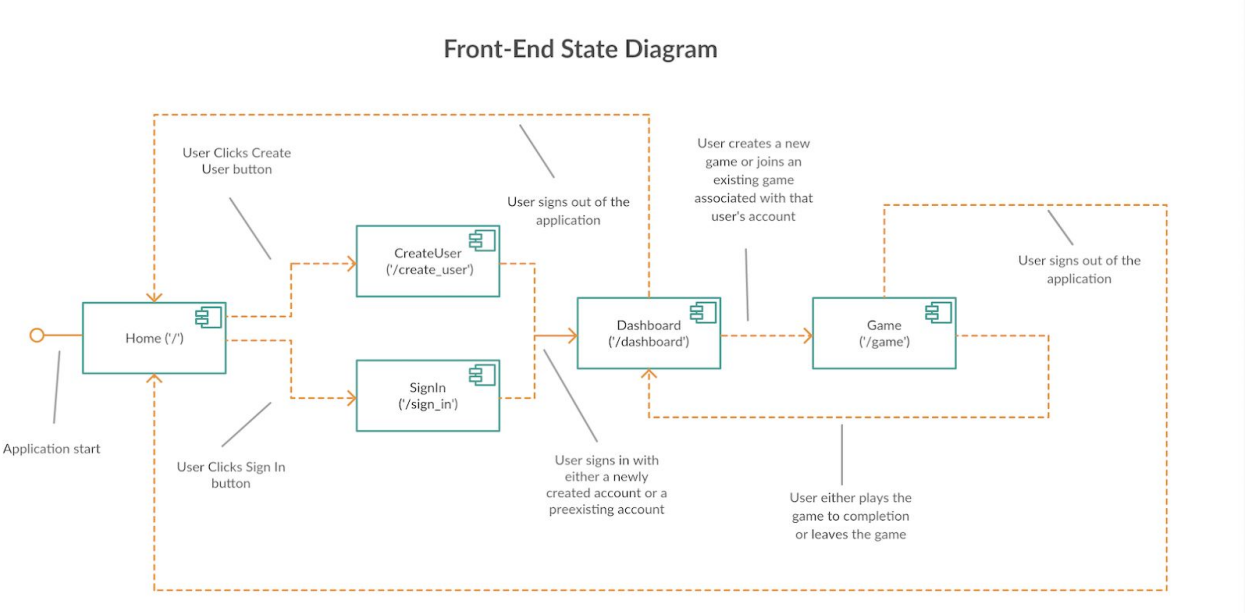
This is actually isn't too much load on the database, as Django caches any repeated requests, so this is really only making two requests. The downside of this approach is that you have to refresh your screen to get the necessary changes. This is an unfortunate user experience, and we wish we could find a better solution.

Front-End Implementation

The front-end of the project is event-driven[6] and is in charge of directing the user to the different views of the project given the user's actions. This can best be summarized with a small chunk of code from 'App.js' file:

```
<BrowserRouter>
  <Switch>
    <Route exact path="/" component={Home} />
    <Route path="/sign_in" component={SignIn} />
    <Route path="/create_user" component={CreateUser} />
    <Route path="/dashboard" component={Dashboard} />
    <Route path="/game" component={Game} />
  </Switch>
</BrowserRouter>
```

The five parent components that make up the front-end of our application are shown above. The user's actions in the front-end delegate which components are rendered. The high level interactions of the parent components shown above could be seen with the following state diagram:



While that state diagram shows a high level overview of the application, the meat of the logic is built into the Game parent component. The Game component has seven subcomponents defined

[6] *Event-driven programming* is a programming paradigm in which the flow of program execution is determined by *events* - for example a user action such as a mouse click, key press, or a message from the operating system or another program. (Technology UK)

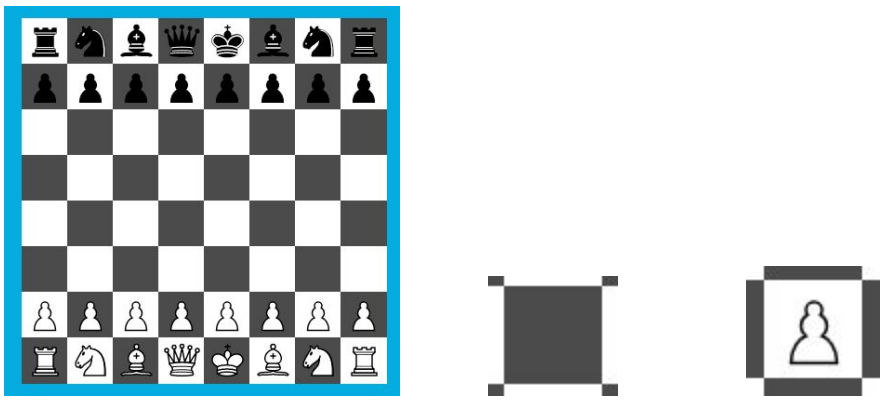
within itself: RightSidebar, LeftSidebar, Board, Square, Piece, PiecePromotion, and GameOver. Each of those subcomponents are used to render the data that is stored in the Game component's state.

The Game component's state is represented as a Hash Table:

```
this.state = {
    squares: this.initBoard(),
    selected: NOT_SELECTED,
    capturedWhitePieces: [],
    capturedBlackPieces: [],
    turn: WHITE,
    count: 0,
    error: "",
    movableSquares: [],
    showPiecePromotion: false,
    promotedPiece: null,
    yourColor: WHITE,
    gameOver: false,
    winner: null,
    gameId: this.getGameIdFromUrl()
  };
```

Each of the entries defined in the Game component's state hash is used to render and modify the data using the subcomponents defined within the Game component class.

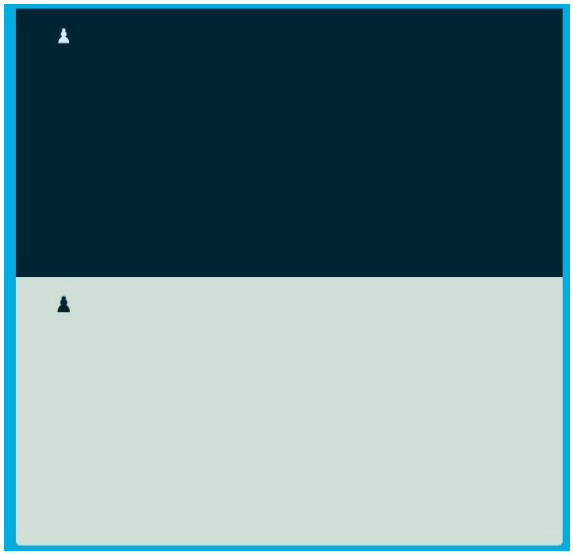
Board Subcomponent



The Board subcomponent uses the following entries from the Game component's state hash: yourColor, squares, handleClick, turn, and move. This subcomponent is used to represent the actual game board in the project. Within this subcomponent Square and Piece are defined. For each element in the square array that is passed in from the Game component, the Board component instantiates a Square which is used as a wrapper for the Piece class. Once Board

renders, it has created an array of size sixty four in which each element in the array represents a square on the board that wraps a Piece object. This process would define a “Square” instance that is visually represented on the board. While none of the logic for moving the pieces was defined in the Square class, the Game component would “pass” it’s handleClick function down to each initialization of the square to handle the logic of the game.

LeftSidebar Subcomponent

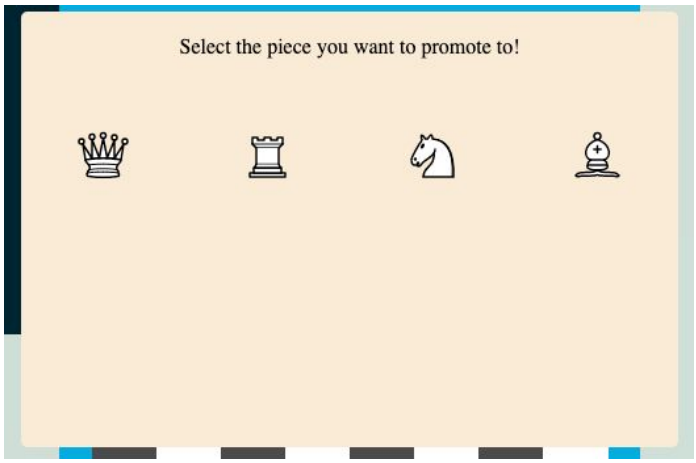


The LeftSidebar subcomponent uses the following entries from the Game component’s state hash: capturedWhitePieces and capturedBlackPieces. The LeftSidebar component has a very simple job of just displaying the data that is passed down to it. Anytime the Game component’s state updates the capturedWhitePiece or capturedBlackPieces, the subcomponent is forced to update with the new array.

RightSidebar Subcomponent

The RightSidebar subcomponent uses the following entries from the Game component's state hash: turn, count, and error. Similar to the LeftSidebar, the RightSidebar component has a simple job of just displaying the data that is passed down to it. On each move, or attempt of a move made by the user, causes an update to the Game component's state entries that are displayed by this subcomponent. On a successful move, the turn switches to the opponent's color and the (move) count increases by one. On an incorrect move, the error string is populated with the suspected error to inform the user what went wrong.

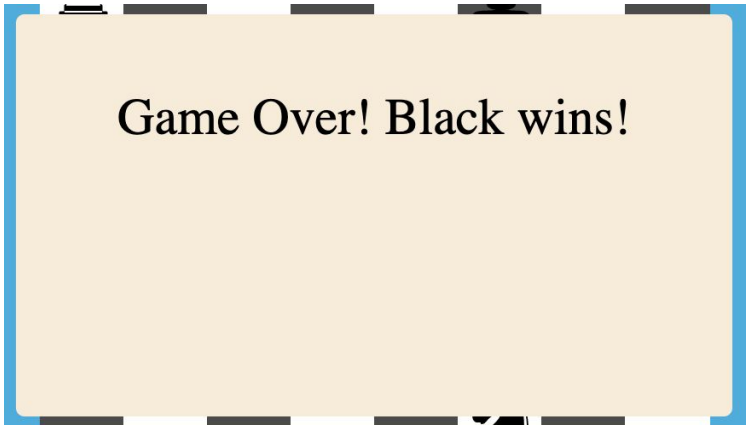
PiecePromotion Subcomponent



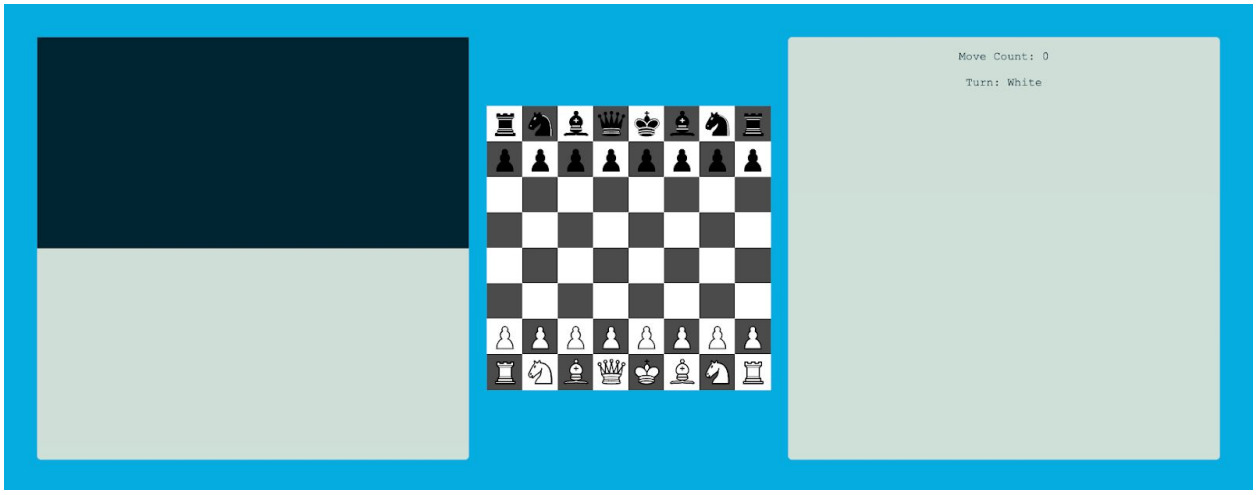
The PiecePromotion subcomponent uses the following entries from the Game component's state hash: show, color, and handleClick. The PiecePromotion subcomponent is

only visible when the show variable is set to true. This will only occur when a user is promoting a piece. Once the condition is met, an overlay displays to the user the possible pieces they are able to promote to and allows them to select one, the overlay disappears after their selection finishes.

GameOver Subcomponent

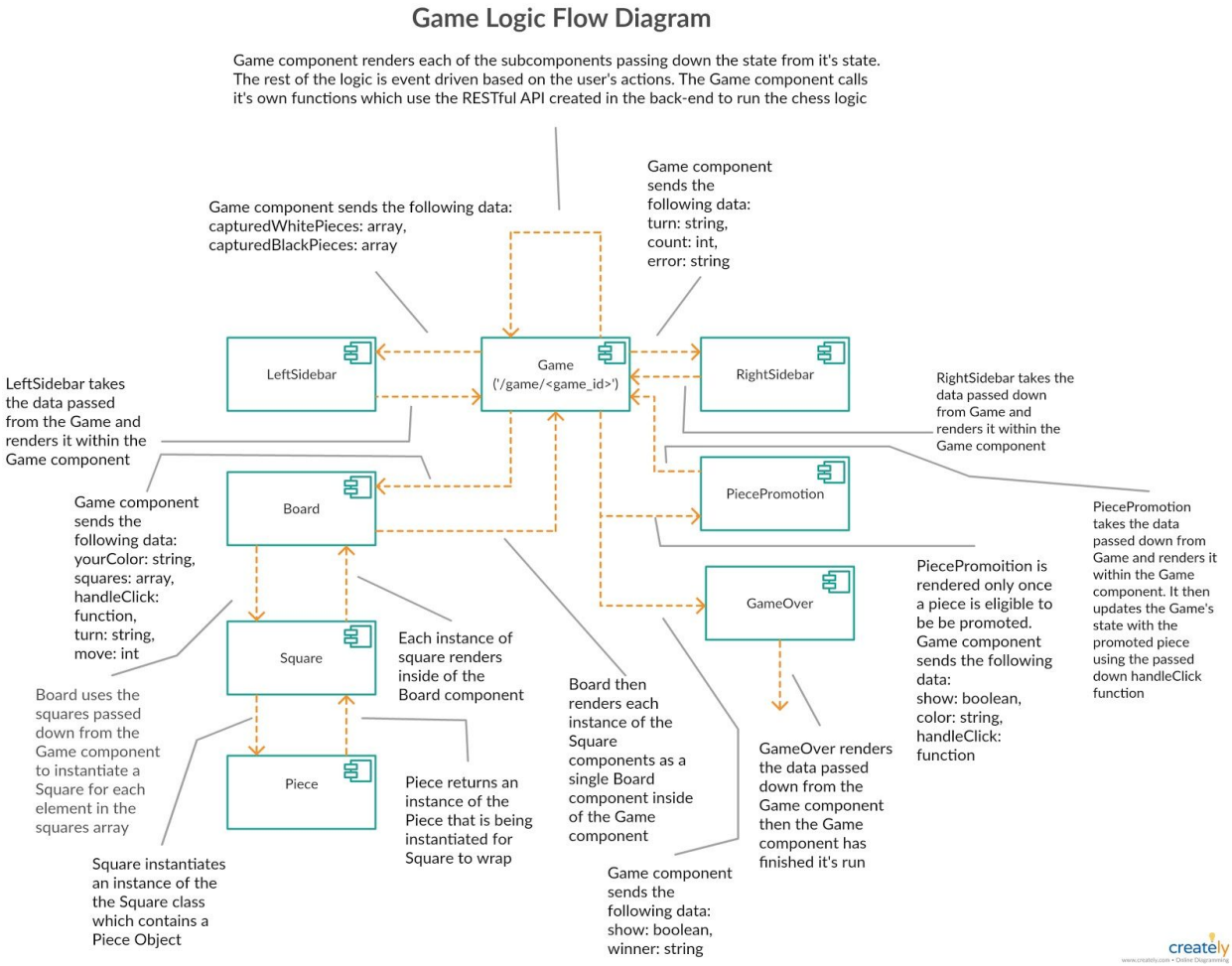


The GameOver subcomponent uses the following entries from the Game component's State hash: show and winner. The GameOver subcomponent is only visible when the show variable is set to true. This will only occur when a user has made a move that would satisfy an end condition. The user is then given the option to return to the Dashboard.



Given that the project is event driven, the chess logic is mostly defined in the 'handleClick' function inside the Game component. The handleClick function is used to ping the back-end to

guide the user's actions to follow the rules of chess. An overview of all these components working together can be seen in the diagram below:



JWT Authentication

While not in our initial scope of the project, we had to implement our own authentication system used to authenticate user accounts on login in and on sign up. Initially when planning our project we intended on using Django's User implementation to handle the User authentication. We encountered issues when attempting to use Django's built in features so we researched alternatives we could use and we agreed on JWT. [7]

[7] JSON Web Token (JWT) is a means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is digitally signed using JSON Web Signature (JWS) and/or encrypted using JSON Web Encryption (JWE). (Open ID)

Back-end JWT Implementation

JWT is essentially a username and salted password encoded with a private key. Whenever a user would login, we would verify they exist in the database, and then return a JWT token to the front-end. The front-end would store this for any subsequent requests. Upon receiving a token, the back-end could decode the token, check the username, and do user-specific actions. Each JWT token has a time-to-live of 48 hours.

Front-end JWT Implementation

JWTs are used throughout the entire front-end side of the project. When a user logs in or signs up, the front-end stores the JWT returned by the back-end using the following code:

```
localStorage.setItem('token', response.token);
```

Given that the JWT is now stored on in the localStorage, any subsequent request to the back-end will send the JWT in the header of the request in order to let the back-end know who is the user. The JWT is sent in the header of a request using the following code:

```
headers: {
  'Content-Type': 'application/json',
  'Authorization': `JWT ${localStorage.getItem('token')}`
}
```

If the user does not have a JWT stored (due to the JWT expiring, or logout) then ‘null’ would be sent to the back-end on the request header which would result in ‘Bad Request’ which is then handled in the front-end by redirecting the user to a screen in which they could request a JWT (login or sign up).

Testing

Software is not complete without tests.

Back-end Testing

The back-end was a lot of organization, mostly reliant on several packages and modules, such as Django and Pychess. I also kept function decomposition in my head at all times. Because of this, my tests were quite easy to write. I was able to mock out external function calls and just ensure that the correct functions were called at the correct times.

Front-end Testing

The testing on the front-end was tested with an Rspec[8] style (also known as 'spec-style') approach. This style of approach helped write very quick tests for the front-end given that most of the game logic is handled in the back-end. This Rspec style of testing was almost enforced by React because it's own built-in testing library, Enzyme, follows that style of testing. Enzyme has an amazing feature (one that we had never heard of before React) called snapshot testing that made the testing of rendered components a breeze.

The way snapshot testing would work is in tests, we would mount a component we would want to test and store a 'snapshot' of the mounted component. That snapshot would be used as a reference for that test anytime it needs to be run again. The next time that the same test is run, the component is mounted again with the same inputs and another snapshot is created. If the newly created snapshot does not match the reference snapshot (that was stored the first time the test was run) then there was an error and a test would fail. Anytime there is an update to the front-end components we would be able to update the snapshots that are stored to ensure our tests would then pass. Shown below is a short example of how a snapshot test is written.

```
describe("modules/Home/Home.jsx", () => {
 it('should render the Home component', () => {
   let home = shallow(<Home />);
   expect(home).toMatchSnapshot();
 });
});
```

[8] writing human readable specifications that direct and validate the development of your application (Better Specs)

In the above example, the Home component is being ‘shallow’ mounted in the testing environment. On the first run (ever) the ‘toMatchSnapshot’ function checks whether there exists a snapshot stored locally that will be used as a reference - if there is no snapshot it will then create a snapshot if itself and store that. Any subsequent run of this test will compare the mounted component with that snapshot. The entire front-end of the project is tested using this manner.

Problems Encountered

“A clever person solves a problem, a wise person avoids it.” - Albert Einstein. We are neither, so we encountered problems and had to live with some of them.

CORS[9] Headers

Possibly the most annoying and time consuming problem we had encountered throughout the entire project were CORS Header issues. As mentioned early on in this paper, we ran our front-end and back-end servers on different domains, in reality they were both run on localhost, but using different ports: port 3000 for the front-end and port 8000 for the back-end. Because the two ends were on different ports they were considered different domains causing an issue when sending data from one to the other because the browser would prevent the data being sent if it did not contain the expected CORS headers. While we did follow every tutorial on how to fix this issue exactly as told - the issue would persist. We still don’t have a full understanding of why this issue arose - or how it was solved since it would sometimes come and go. In the end we have the project working by using Django’s ‘corsheader’ plugin used to add the required headers on each request.

User Authentication

Let’s go a little bit into the issues of original approach of our user model. Django has its own built in authentication system that is rather simple. The front-end can query a simple login function that takes a HttpRequest object containing a username and password. Once you have

[9] Cross-Origin Resource Sharing (CORS) is a mechanism that uses additional HTTP headers to tell a browser to let a web application running at one origin (domain) have permission to access selected resources from a server at a different origin (Mozilla)

that, authenticating the user credentials is easy. In the figure below, you can see the login function that the official Django documentation recommends.

```
def login_user(request)
    user = authenticate(request.username, request.password)
    if user is not None:
        login(request, user)
    else:
        raise UserError("User authentication failed")
```

The `authenticate()` function looks up the username in the user database and makes sure that the hashed password matches what the database has on recond. Assuming that succeeds, it returns a `User` object. You can then call `login()` on the user, and Django will add a `sessionid` cookie to its session database, and will forever associate any incoming request that has the aforementioned sessionid with the authenticated user.

At first we were ecstatic that the User model would be so simple, but alas, we were wrong. For some reason or another, this simply did not work. I wish I could tell you why, but I have no answer. We spent around fifteen hours debugging this, watched countless online tutorials, and even copy and pasted the exact code from other Django projects that used this method successfully. For whatever reason, we could not get it to work.

The JWT method we ended up using was simple and worked instead, so we were glad to find a replacement.

Chess Clock

Implementing the chess clock in the front-end of the project introduced more issues than we would like to admit. Initially, we implemented our own ‘in-house’ solution by creating a simple timer to mimic a chess clock. Problems arose because Javascript is always run on a single thread and this caused some issues such as the timer not counting down correctly (sometimes it would skip numbers or count fairly quickly).

We fixed that by using a React module and adjusting it to fit the needs of the chess game, but the new clock had issues in which the White and Black clocks would not count down individually - they would each seem to be using the same count down time even when explicitly given two different values. This issue was never fixed, because the solution seemed impossible (I still don’t know what is wrong, they were given their own values to count down from) and because we realized that the counting would either have to be implemented in the back-end, or we would have to send the count down time everytime it was updated to the back-end in order to store the time in the database. We did not want to make a request to the back-end every time the

clock would update - so we decided to opt out of using the chess clock, and because of that any instance of the chess clock is omitted from this report.

Conclusion

Wow, so much goes into building a website. What we thought would be a simple database with a UI, we quickly found out that simple configuration items, like user models and cookies, could cause a world of trouble. We never had any idea that so much went into something We previously thought was simple, like logging into a website. The whole world takes the Internet for granted, and the fact that any website works is a miracle.

Thanks

Thank you for your guidance and understanding of our struggles throughout this project! - Gaston

I wanted to say thanks for being a great senior project advisor! - Jacob

Dependency List

Back-end Dependencies

Django==2.2
django-cors-headers==2.5.2
djangorestframework==3.9.2
djangorestframework-jwt==1.11.0
PyJWT==1.7.1
pytz==2019.1
sqlparse==0.3.0

Front-end Dependencies

babel/core: 7.4.5

material-ui/core: 4.0.1

material-ui/icons: 3.0.2

material-ui/lab: 4.0.0-alpha.14

chess.js: 0.10.2

enzyme: 3.10.0

enzyme-adapter-react-16: 1.14.0

jest-enzyme: 7.0.2

jquery: 3.4.1

material-ui-slider: 3.0.3

react: 16.8.6

react-autobind: 1.0.6

react-countdown-now: 2.1.0"

react-dom: 16.7.0

react-router-dom: 4.4.0-beta.6

react-scripts: 2.1.3

react-test-renderer: 16.8.6

Resources

https://www.chesshere.com/resources/chess_history.php

https://new.uschess.org/about/

https://searchmicroservices.techtarget.com/definition/RESTful-API

https://www.chess.com/blog/zaifrun/creating-a-chess-engine-from-scratch-part-1

http://www.technologyuk.net/software-development/designing-software/event-driven-programming.shtml

http://www.betterspecs.org/

https://openid.net/specs/draft-jones-json-web-token-07.html

https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

http://wbec-ridderkerk.nl/html/UCIProtocol.html