1 Introduction
As a child, I loved all kinds of games. Whether it was sports, video games, Monopoly with my family, or just some random game at the park, I could play for hours on end. Even today, I find myself attracted to the competitiveness and rule sets of different games, particularly board games. Most recently I picked up Settlers of Catan and initially thought I could create an AI to play against. However, after learning the nuances of the game and realizing the vast space of potential states, that dream quickly died. With my history of games and Professor Glenn’s background in the same field, we discussed potential project ideas and decided on implementing Kismet, a variant of Yahtzee. Yahtzee is one of the more vintage board games in the US, created in the early 1940s. Its survival as a popular family board game staple attests to the attractive simplicity of its rule set. Initially, the goal was to create a simple, playable web app along with an intelligent computer player that could optimally play 2 player Kismet. The optimal strategy for solitaire Yahtzee has already been solved but little research has been done on an optimal 2 player strategy as the presence of another variable (your opponent’s score) vastly increases the difficulty and computation cost. For example, in the optimal solitaire version you are always attempting to maximize your probability of points (go for the highest expected value) but with Kismet you may want to go for a safer option that garners less points but still puts you ahead of the opponent. Additionally, I was enrolled in Professor Glenn’s Computational Intelligence for Games course this semester. One of our assignments was creating an intelligent player for solitaire Yahtzee. As my main goal for this project was to expose myself to new ideas and frameworks, I thought then that implementing an intelligent player for Kismet was less of a novel task. With these two things in mind, along with my desire to focus on more of a UI/front-end approach to the project, I pivoted halfway through the semester to focus only on creating a simple, sleek web app and then integrating my Kismet app with fellow senior Dirk Bell’s server architecture to create a playable Kismet game through the command line against a player on another server.

2 Goals
I wanted to create the web app GUI with a front-end framework that would be useful both for my personal use and future career experience in software engineering. As such, I chose to use React and Redux, of which only the former I had slight experience in. Both frameworks utilize JavaScript. React has been exploding in popularity recently: it utilizes a virtual DOM as opposed to the regular DOM which allows for extremely efficient rerendering while also encouraging a declarative, intuitive approach to building out components that make it relatively pain free to learn. Redux,
an interface for managing application state, nicely couples with the design architecture of React, making them a powerful duo in building and scaling out front-end GUIs.

One of the main goals was also to employ good object oriented programming skills while also creating a simple, yet attractive user experience. Doing so would allow for easy porting to other interfaces and applications, which I eventually did by integrating with Dirk Bell's project. Additionally, by nature of being a turn based board game, Kismet required keeping track of various scores, states, and board traits. These all had to correctly monitored and updated in order to allow for smooth play, further enforcing a paradigm of good object and class design. The usage of Redux greatly helped with this.

3 Kismet

Kismet, while practically identical to Yahtzee, holds some key characteristics. Before explaining the differences, the general rules of Kismet are as follows:

1. There are a total of 15 rounds and 15 scorable categories. 5 6-sided dice are used, with certain sides having matching colors: 1s and 6s share the same color, as do 2s and 5s, and 3s and 4s.

2. At the beginning of every round, for his turn, a player rolls 5 dice. The player then has 2 options: he can score into a category using the values of the dice, or he can select from 1 to 5 dice to reroll. He has a limit of 2 rerolls (for a grand total of 3 rolls per round).

3. Each category holds different rules to score based on the both the value and color of the die rolls. As this is meant to be just a quick overview of Kismet rules, I won’t list each of them out. The most important points are:
   a. Once a category is scored into, it cannot be scored into again.
   b. A player must choose a category at the end of each round. If no categories will give a score for the current dice, a score of 0 is inserted into the chosen category.

4. There are additional rules for bonus points earned based off on the amounts of points in certain sections. At the end of the 15 rounds, the player with the highest score wins.

While initially daunting due to the large number of categories, the rule set becomes quite simple after playing just a few times. The combination of strategy and luck make it exciting but also quite quick (as opposed to the hour long rounds of modern games). It is different from Yahtzee in that it has 15 rounds as opposed to 13 and colored die faces where Yahtzee uses a standard die.

4 Text Based Implementation

While the rules are rather simple in comparison to other board games, coming up with the logic for handling each category based on the current roll of the dice was less trivial. As a starting step, I created a Python script to play solitaire Kismet through the command line. My Python objects were as followed: a Board, 15 Categories that all inherited from the same base class, and Dice. Splitting up Kismet into these 3 components made it straightforward for each object to communicate each other and update accordingly. Creating the logic for category scoring made it easy to port to my JavaScript web app code and then over to Dirk’s Python server.
5 React and Redux

Rather than explaining the nuances of React and Redux (which one can do simply with a quick google search), I thought it would be more informative to explain some of my code in the context of my goals. Here is a snippet of code from one of my React components, Category:

```javascript
import React, { Component } from 'react';
import { connect } from 'react-redux';
import { updateScore } from '../actions/scoreActions';
import { updateTurn } from '../actions/boardActions';

class Category extends Component {
  constructor(props) {
    super(props);
    this.onClick = this.onClick.bind(this);
    this.blank = ' __';
  }

  onClick() {
    var sorted_dice = this.props.dice.slice().sort();
    var score = this.props.rule(sorted_dice);
    this.props.updateScore(this.props.player, score, this.props.cat);
    this.props.updateTurn();
  }

  render() {
    var which_cat =
      this.props.player == 1 ?
        (this.props.p1_cats[this.props.cat] == null ? this.blank :
          this.props.p1_cats[this.props.cat]) :
        (this.props.p2_cats[this.props.cat] == null ? this.blank :
          this.props.p2_cats[this.props.cat]);

    var already_scored =
      this.props.player == 1 ?
        (this.props.p1_cats[this.props.cat] == null ? false : true) :
        (this.props.p2_cats[this.props.cat] == null ? false : true);

    var style_p1 = { 'backgroundColor': '#0000EE' };
    var style_p2 = { 'backgroundColor': '#FF00FF' };

    return (
      <tr>
        <td>{this.props.name}</td>
        <td>{which_cat}</td>
        <td>
          {(already_scored || this.props.player != this.props.turn ||
            this.props.rolls_left == 3) ?
            null :
            <button
              className='cat-button'
              onClick={this.onClick}
              style={this.props.player == 1 ? style_p1 : style_p2}
            >
          />
        }
      </tr>
    );
  }
}
```
const mapStateToProps = state => {
  turn: state.board.turn,
  dice: state.board.dice,
  rolls_left: state.board.rolls_left,
  p1_cats: state.score.p1_categories,
  p2_cats: state.score.p2_categories
}

const mapDispatchToProps = {
  updateScore,
  updateTurn
};

export default connect(mapStateToProps, mapDispatchToProps)(Category)

It would be remiss to not discuss the few most important features of React. The major object of React is its Component class. The beauty of such a class is that it comes with essentially 3 major functions: life-cycle methods that trigger upon virtual DOM related events, a render function, and a constructor. The neat thing about the constructor is that the user can specify a component's state variables, that, when changed, pass a signal to the component and allow the component to rerender in the app. This is essentially Ajax and allows for rerendering HTML without having to reload the entire page and also makes it incredibly easy to debug. Components can pass variables and functions to their child components, which can easily be accessed through a property variable called props. This creates an intuitive downward flow of information.

Redux solves the issue of having to continuously pass down props from parents to children to grandchildren components and so on. For example, my Kismet objects, the Dice, Board, and Categories need to all be on the same page and all have access to the same variables: the value of the dice rolls, which categories have been scored into, the current player turn, the round number, etc. In a simple text-based program, where everything shares a global scope, this is an easy process. However, for React, where there is a hierarchical structure of components (HTML), these variables cannot as easily pass around freely, especially when passing from a child to a parent. That’s where Redux comes in to save the day. For Redux, there are essentially 3 major components as well: actions, reducers, and the store. Most fundamental is the store, which becomes the single source of data for all components. A component simply needs to connect to the store and subscribe to certain variables. In effect, the component will be notified when these variables change in the store (perhaps through changes made at other areas of the web app) and can rerender accordingly. For example, this component Category both displays and updates the score of the specific category (which is computed in a different component class) and can similarly update the store with the score so that other components are aware that 1) a move has been made and 2) a new score has been entered.

One of the biggest issues I ran across while developing the front-end were the dice. I wanted to create animated dice that would create a 3-D spinning effect when rolled. There were some existing open source React components (another one of the beauties of the popularity of React) that required only a little bit a tweaking with my code. The major problem came however with the potential bugs, which I had to learn the hard way. JavaScript programming is inherently
asynchronous. Take the following code as example:

```javascript
var res = someFunction();
makeAsynchCall(res);
```

If `someFunction()` must make some sort of HTTP request, by the time it receives its response `makeAsynchCall()` will have already executed, but the value of `res` would be undefined, leading to a run time error. This was a similar issue to what was happening with the dice: while the dice were in animation, a user could theoretically spam the roll button or complete other actions before the dice values were actually calculated (this did not happen until after the dice finished spinning, a design flaw by the original creator). To mitigate this, I simply set the animation time to 0, sacrificing some UI capabilities for more fail proof code.

6 Command Line Server App

In the final stages of the project, I integrated my Kismet implementation with Dirk Bell’s lightweight server architecture designed to allow for 2 player turn based games. Ultimately, this was not a difficult process as it involved mainly just moving my text-based Python classes into his server code, creating some new logic to deal with playing the game on the command line and prompting the user for certain kinds of information (scoring vs rerolling). We both attribute this to the fact that we initially designed good object oriented classes. This allowed for easy crossover, as many of the class methods for playing the game were either pre-existing or easily implemented using other existing code. Dirk utilized a TCP protocol server to communicate between the two clients playing the game.

7 Final Thoughts and Room for Future Exploration

I had a ton of fun implementing this project. While I have created web apps before, I have never created a turn based board game. Doing so required a much different approach than one I would take to creating a static, single page photography portfolio for example.

Learning Redux was also a treat. It is nice to be able to equip such a versatile and relevant framework into my software engineering tool belt. The concepts of actions and reducers took a little bit of time to understand, but it was a great way to modularize my code and make it clear to follow.

Integrating projects with Dirk was a great experience. In many ways, it felt similar to the conjunction of ideas in a real world setting, a scenario one may see at a start up or something similar. It reinforced the importance of making ones code readable and intuitive (with comments in the case that it is not) so that someone completely alien to it could follow along.

Lastly, there were plenty of areas for improvement and further exploration. Most obvious is the path of intelligent computer players. There is much work to be done in the field of optimal strategies for 2 player board games such as Kismet. There are already major advancements being made with games such as Chess and Go. Additionally, with more time I would love to fully flesh out the UI and make it even more interactive, as well as push it up onto a server so that one does not need to clone the repo and run a server locally themselves.