CMP3035M: Cross-Platform Development - Assignment 2

Angelo Hague

University of Lincoln, School of Computer Science
16631843@students.lincoln.ac.uk

Table of Contents

1

Overview

IOU Poker is a hypothetical gambling application (app) which allows users to play games of Texas Holdem with custom wagers. This concept was drafted from the COVID-19 pandemic causing individuals who would typically play at-home games to seek online alternatives. As a result, these players had to choose between bet-free games, managing debts themselves, or online live tables, such as *Sky Vegas*, in which others could join. Subsequently, I chose to create an app which would allow for the desired gameplay whilst also tracking the user's gambles. The app was created using React Native, and is thus accessible on both Android and iOS devices alike.

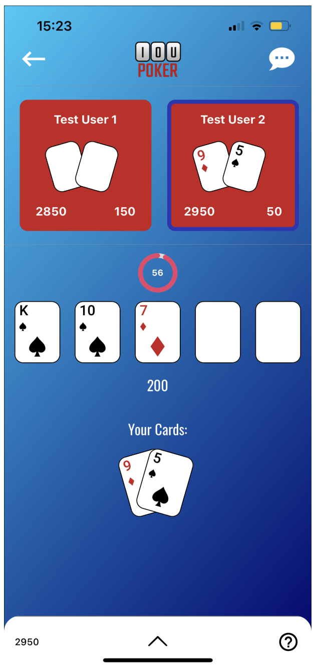


Fig. 1. Screenshot of IOU Poker running on an iPhone 13 Pro

The app utilises *Firebase Authentication* which allows for users to create an account to utilise the service. While logged in, users can create a game with custom stakes, or none at all, and then share the join code with their friends so that they can join and play. Some factors can also be changed, such as the amount of chips each player starts will and the big and small blinds. Upon joining a lobby, players will be able to see the game lobby's stakes and rules as well as being able to chat freely and toggle their ready state. When all players are ready, a countdown is initiated and then the game begins. Upon completing a game, if a stake and amount was set, then the respective debt will be added to each player's account.

The lobby and the game itself is handled by a custom backend server developed using Multiplayer Node.js Framework *Colyseus*. This is to ensure synchronous and timed gameplay between clients. Another option was to use *Firebase Realtime Databases*, but the main advantage *Colyseus* had over it was that it allowed for server side authentication, preventing clients from cheating by checking the room's state within code. *Colyseus* also allowed for information to be communicated to clients on demand, and for player turns to be timed out if they took too long, providing a better user experience.

2

Final Application

1 Proof of Deployment

– A video showcasing the app can be found on YouTube here: `https://youtu.be/6aV0bMh1ys0`

– Images can also be found here: `https://imgur.com/a/kT9KKit`

2 Design and Development

2.1 Debt System

User Experience Upon completing a game with a stake and amount, a debt to the winning player is registered for each of the losing players. This information is then displayed in the Debts section of the app. The Debts screen loads the information from *Firebase's Firestore* and displays it accordingly. Debts are separated into 3 sections; outgoing, incoming, and settled *(see Fig. 4 below)*. Spreading this information across 3 different sections reduces the amount of clutter on the screen, making it much more readable and allowing for easier navigation. Navigation between the sections is easy due to large "finger-friendly tap-targets" Babich (2018), which makes them very clear to the user *(see Fig. 2)*.

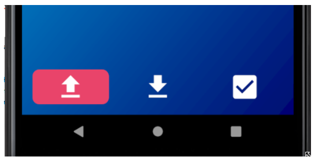
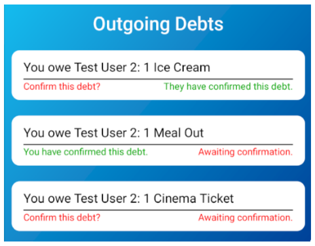


Fig. 2. Large navigation buttons are used to easily switch between the different debt screens



Outgoing Debts

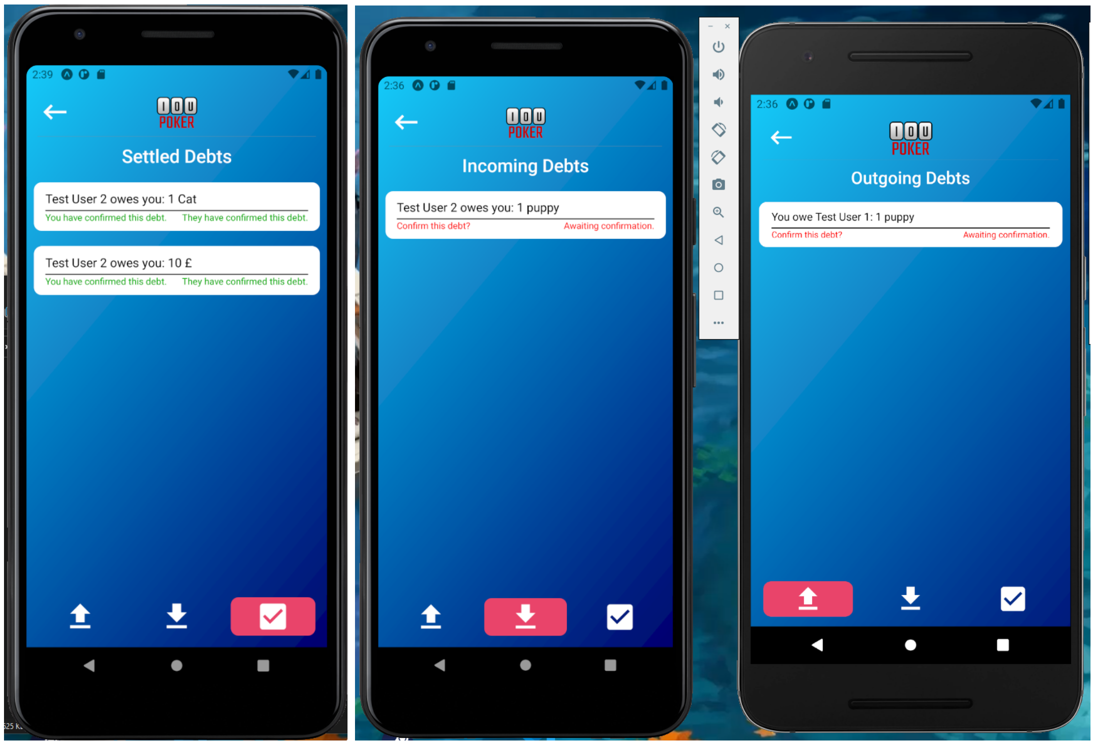
You owe Test User 2: 1 Ice Cream
Confirm this debt? They have confirmed this debt.

You owe Test User 2: 1 Meal Out
You have confirmed this debt. Awaiting confirmation.

You owe Test User 2: 1 Cinema Ticket
Confirm this debt? Awaiting confirmation.

Fig. 3. Large navigation buttons are used to easily switch between the different debt screens

Each section lists its respective debts, each of which clearly indicates whether or not the winner and loser have yet to mark it as complete *(see Fig. 3)*. If the player has not marked it as complete, then the component reads "Confirm this debt?", encouraging engagement with the use which invites them to tap the component. Upon doing so, their device will display an alert, telling them that once both players have marked it as complete it will be moved to their Settled Debts section. As this is primarily a text component, "making text legible is a mandatory requirement" Babich (2018). This, along with all other sizing of components and fonts throughout the app, is done through normalisation based on screen-size. This way, the app's dimensions will remain consistent across all devices.



Test User 2 owes you: 1 Cat
You have confirmed this debt. They have confirmed this debt.

Test User 2 owes you: 10 £
You have confirmed this debt. They have confirmed this debt.

Test User 2 owes you: 1 puppy
Confirm this debt? Awaiting confirmation.

You owe Test User 1: 1 puppy
Confirm this debt? Awaiting confirmation.

Fig. 4. Debts listed in the outgoing section.

Functionality Sections are shown and hidden accordingly using their respective version of the following function. By using setState(), the component will refresh as a result. Each section is then only displayed conditionally if their respective "show" variable is set to true. The renderDebts() function then iteratively loads and renders a component for each debt, displaying them in a list.

```
showOutgoing = () => {
    this.setState({show_outgoing_screen: true, show_incoming_screen: false, show_settled_screen: false})
}
```

```
renderDebts(debts) {
    const components = []
    {[...debts.values()].map(debt=> {
        let [doc, out] = debt
        components.push(<Debt key={doc.id} id={doc.id} document={doc} outgoing={out} />)
    })}
    return components
}
```

```
                    <ScrollView style={{ flex: 1, flexGrow: 1 }}>
                        {this.state.show_outgoing_screen &&
                        ↪   this.renderDebts(this.state.outgoing)}
```

When the screen itself is mounted, the fetchData() function is called. This asynchronously connects to Firebase's Firestore, and gets any records containing the user's Authentication UID (user id). This data is then stored in a respective map - depending on whether its outgoing, incoming, or settled - with each key being its document ID. The boolean value which is added alongside the document object ([doc, true]) indicates whether it is outgoing or incoming, as settled debts could be either.

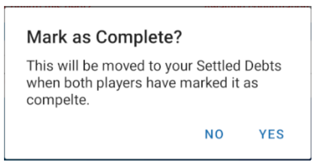
```
    async fetchData() {
        let outgoing = new Map()
        let incoming = new Map()
        let settled = new Map()

        await db.collection("debts").where("sender", "==", auth.currentUser.uid)
        .get()
        .then((querySnapshot) => {
            console.log('loading outgoing debts') // debug purposes
            querySnapshot.forEach((doc) => {
                // doc.data() is never undefined for query doc snapshots
                // console.log(doc.data())
                if (doc.data().completed == true || (doc.data().approved_by_recipient
                ↪   && doc.data().approved_by_sender)) {
                    settled.set(doc.id, [doc, true])
                } else outgoing.set(doc.id, [doc, true])
            });
        })
        .catch((error) => {
            console.log("Error getting documents: ", error);
        });
```

...

```
        this.setState({outgoing: outgoing, incoming: incoming, settled: settled})
        console.log('Done loading')
    }
```

Upon tapping a debt, a user is prompted with a device alert which asks them to confirm their decision *(see Fig 5.)*, as well as conveying that the debt will be moved to the Settled Debts section upon **both** the user's confirmation. This is doing using React Native's Alert module, which prompts the device, regardless of its Operating System (OS), to display a system alert.


Mark as Complete?

This will be moved to your Settled Debts when both players have marked it as compelte.

Fig. 5. The Alert which shows upon tapping an unconfirmed debt component on an Android device.

```
        Alert.alert(
        "Mark as Complete?",
        "This will be moved to your Settled Debts when both players have marked it as
        ↪  complete.",
        [
            {
            text: "No",
            onPress: () => {},
            //onPress: () => console.log("Cancel Pressed"),
            style: "Cancel"
            },
            { text: "Yes", onPress: () => {this.markAsComplete(this.props.outgoing)} }
        ])
```

Upon confirmation, the component will determine if the user is the sender or receiver, and then connect to Firebase and update the corresponding record to be marked as approved by them. If it is already confirmed by the other player, is is also marked as completed. The following if-else expression checks to see if the debt is the user is the sender or receiver of this debt, and the contents then update the respective records.

```
    markAsComplete(outgoing) {
        let ref = db.collection('debts').doc(this.props.id)
        if (outgoing) {
            if (this.isMarkedAsCompleteByUser(!outgoing))
            ref.update({
                approved_by_sender: true,
                completed: true,
            }).then((promise) => {
                console.log('Updated. ', promise)
                this.setState({approvedBySender: true, completed: true})
            })
            else ref.update({
                approved_by_sender: true,
            }).then((promise) => {
                console.log('Updated. ', promise)
                this.setState({approvedBySender: true})
            })
        } else {
```

As aforementioned, to ensure these components, along with all others across the app, are displayed consistently across devices, normalisation functions are used. To do this, I determined a scale factor based off the screen's dimensions in relation to the emulator screen I primarily tested on - the Pixel 3a.

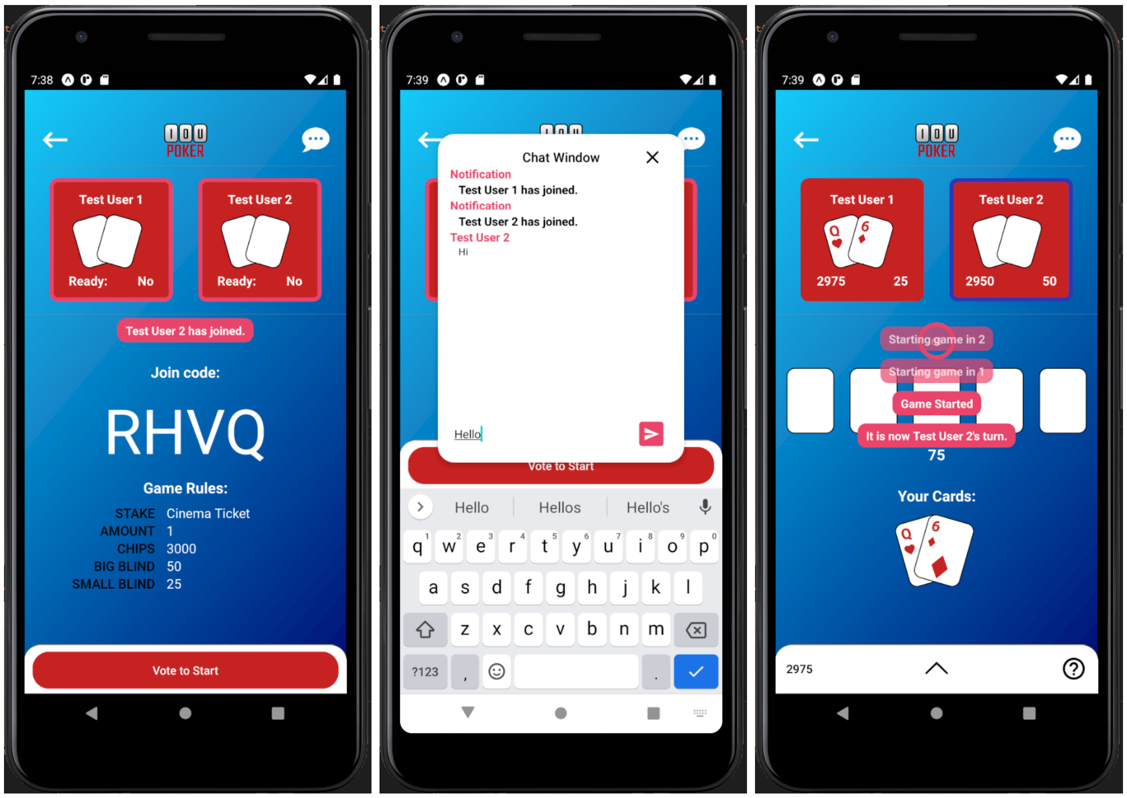
```
  const scale_width = SCREEN_WIDTH / 393;
  const scale_height = SCREEN_HEIGHT / 760;
```

Through these determined scales, a normalising function, as follows, can then alter the size of any component or font that I use throughout my application.

```
    if (Platform.OS === 'ios') {
      return Math.round(PixelRatio.roundToNearestPixel(newSize))
    } else {
      return Math.round(PixelRatio.roundToNearestPixel(newSize)) - 2
    }
```

2.2 Game Lobby and Gameplay

User Experience Designing a user interface for a mobile game proved to be rather difficult; it required a lot of information being communicated to the user while ensuring the screen is not overly cluttered, which could draw the user's attention away from the main aspects. An important technique here was simple sizing differences. Each player's information can be seen within their Player Card, which are contained within a ScrollView which accounts for perhaps 30% of the screen. That makes the information easily accessible whilst also containing it as to not be too greedy with screen space *(see Fig. 6)*. The Room Code then takes up a similar amount of space in a rather large font, while the rules take up far less space, but are also still clear enough to read. This technique is then used once the game has begun, having only the players cards at the top, a turn timer, the community cards (the five in the middle), and the player's hand on screen.



Test User 2 has joined.

STAKE	Cinema Ticket
AMOUNT	1
CHIPS	3000
BIG BLIND	50
SMALL BLIND	25

Notification
Test User 1 has joined.
Notification
Test User 2 has joined.
Test User 2
Hi

Starting game in 2
Starting game in 1
Game Started
It is now Test User 2's turn.

Fig. 6. The Lobby Screen, the Chat Window, and Notifcations, respectively.

Any important information about the game, whether from the user or another player's actions, is displayed through notifications over the game screen, as "it's essential to provide some sort of feedback in response to every user action" Babich (2018). Vibration is also used to communicate information, alerting the player when it is their turn. This is a good use of a mobile-exclusive feature and, by using it sparsely, it alerts and engages the user; "[harnessing] tactile feedback to deliver timely and useful information through the experience of touch" Baker (2019).

The notifications are also listed within the Chat Window, which using a Keyboard Avoiding View, will move and remain on-screen along with the rest of the game screen when the chat is open and the keyboard is active, which is also considered good design *Improving User Experience · React Native* (n.d.).

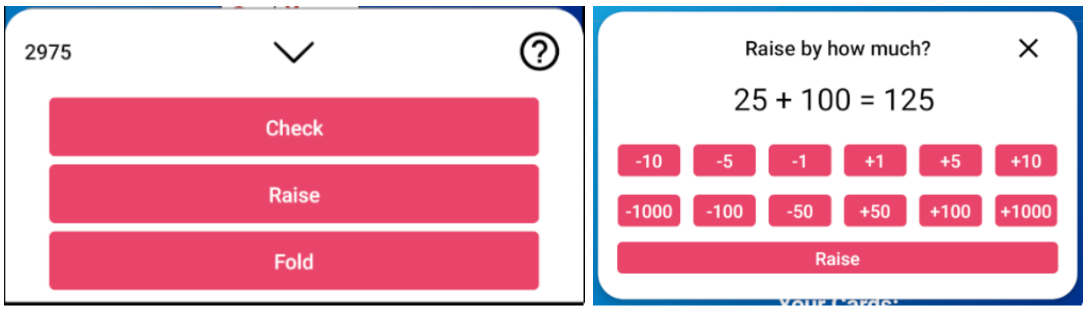


Fig. 7. The Expandable Options Menu (left) and then Modal Raise Window (right).

The user's available actions are all neatly compacted into the expandable options menu at the bottom of the game panel *(see Fig. 6 and Fig. 7)*. Contrary to the Keyboard Avoiding View aforementioned, an effort has also been made to "minimise the need for typing" Babich (2018). The modal window which allows the player to raise their bet consists of buttons allowing for various increases and decreases, meaning the player does not have to use the keyboard and also making for simpler control over the window.

Functionality The Gameplay screens are effectively the visualisation of the Game State that is managed and communicated by the Colyseus server. Therefore, both the front and backend of this component will be discussed here.

When creating a lobby, the client (application) connects to the server and calls the onCreate function. In doing so, a room is created with the user's specified settings and a 4-long alphanumeric room code is generated and assigned to the room. Upon doing so, the server also creates 3 onMessage() listeners which will be used to handle the client's actions; *message*, *changeReadyState* and *playerTurn*.

```
  async onCreate (options: any) {
    let settings = options.settings

    this.setState(new GameState());
    this.state.stake = settings.stake
    this.state.amount = parseInt(settings.amount)
    this.state.chips = parseInt(settings.chips)
    this.state.b_blind = parseInt(settings.b_blind)
    this.state.s_blind = parseInt(settings.s_blind)

    this.roomId = await this.generateRoomId(); // get unique room code
    this.onMessage("message", (client, message) => {
      console.log(client.sessionId, " said: ", message)
      sendMessage(this, this.state, message, getPlayerName(this.state,
      ↪  client.sessionId), false)
    });

    this.onMessage("changeReadyState", (client, message) => { ...
    });

    this.onMessage("playerTurn", (client, message) => {
      playerTurn(this, this.state, client, message)
    });
```

Upon joining a lobby, players have one clear option - the ”Vote to Start” button. The only other buttons on screen are the*Back* and *Chat* buttons *(refer back to Fig. 6)*. The player’s *ready status* is handled by toggling a boolean variable and communicating it with the server. In doing so, the server updates the player’s ready status which, along with lots of other information, is then listened for in the application and rendered appropriately. The border of the respective player card is changed from red to green to indicate their ready status.

```
        if (!room.state.game_started) {
            borderColor = this.props.player.ready ? '#b4f56c' : '#E9446A'
        } else {
            borderColor = (this.props.current_player === this.props.player.sid) ?
            ↪  '#2b37b5' : poker_red
        }
```

The Player Listeners are defined as such, and can manage the addition, change or removal or items within the structure. A similar listener is also used for handling the community cards. Each player structure contains many different variables which are used for managing the game state, though only some are important to the user; *name, ready, chips, current_bet,* and *cards* (though the values are hidden from other players until revealed).

```
export function playerListener(component){
    global.room.state.players.onAdd = (player, key) => {
        global.room.state.players.set(key, player)
        addPlayers(component, player) // render players in current state

        player.onChange = (changes) => {
            changes.forEach(change => {
                if (change.field == 'ready') {
                    let player = global.room.state.players.get(key)
                player.ready = change.value
                global.room.state.players.set(key, player)
                updatePlayers(component, global.room.state.players) // render players in current state
            } else if (change.field == 'chips') { ...
```

The aforementioned Chat, is handled by sending a message to the server, which is then broadcasts to every client in the respective room, which is then parsed and rendered as a message. As the chat window is also used to log notifications, this same system is used by the server to communicate notifications. The client parses the message, and checks for the isNotification value, adding it to either just the messages array or both that and the notifications array.

```
    room.onMessage("message", (message) => {
        component.setState({ chat_messages: [...component.state.chat_messages, message] })
        if (message.isNotification == true) component.setState({ notifications: [...component.state.notifications, message.message] })
    })
```

The Chat window then, similarly to the debts screen, extracts and renders individual messages into Message components. The component checks if the message is marked as a notification, and if so renders it differently to a chat message, so that the user can differentiate between the information *(refer back to Fig. 6)*.

```
    render() {
        let message = this.props.message
        if (message) {
            return (message.isNotification) ? (
                <View style={{width: '100%'}}>
                    <Text style={{color: '#E9446A', fontWeight: 'bold', fontSize: normaliseFont(14)}}>{'Notification'}</Text>
                    <Text style={{marginLeft: normaliseWidth(10),fontWeight: 'bold', fontSize: normaliseFont(14)}}>{this.props.message.message}</Text>
                </View>
            ) : (
                <View style={{width: '100%'}}>
                    <Text style={{color: '#E9446A', fontWeight: 'bold', fontSize: normaliseFont(14)}}>{this.props.message.sender}</Text>
                    <Text style={{marginLeft: normaliseWidth(10), fontSize: normaliseFont(12)}}>{this.props.message.message}</Text>
                </View>
            )
        }
        else return(<></>)
    }
```

Notifications are also rendered in a Flash Displayer component as they are added to the array. The Notification component itself uses React Native's Animated module to fade-out and hide the displayed view when it is no longer visible. This is done by reducing a value from 1 to 0 over a specified duration, and applying it to the components opacity in its style property.

```
    fadeOut = () => {
        let duration = (this.props.duration) ? this.props.duration : 1000
        Animated.timing(this.state.fadeAnimation, {
          toValue: 0,
          duration: duration,
          useNativeDriver: true
        }).start()
      }; ...

            return this.state.visible ? (
                <Animated.View style={[styles.notification, {opacity:
                ↪  this.state.fadeAnimation}]}>
                    <Text style={{color: '#fff', fontWeight: 'bold', marginHorizontal:
                    ↪  normaliseWidth(10)}}>{notification}</Text>
                </Animated.View>
```

When each player marks themselves as ready, the server commences the game, creating and shuffling a deck of cards - an Array Schema of the custom Card object defined on the server. A Card Object has 3 values; *owner, value,* and *revealed*. The value of the card is only visible to clients if their Session ID matches the owner variable, or the card itself has been set to be revealed by the server. This prevents users from being able to see the cards in the source code. These cards are then dealt to each player and the community cards pile.

Then, the dealer, big blind and small blind players are all set by indexing the Map of players which players are added to upon joining the lobby. The current player is set to the big blind, and the game then awaits their action. If they do not act within 60 seconds, the game will automatically fold the player, removing them from the round. This is done by using the Clock module's setTimeout() function, to time out after 6000 milliseconds. If the function times out, it calls the playerFold() function. Using the setInterval() function, we can check every second to see if the turn_idx value has changed, meaning the player has acted, and, if so, cancel the Timeout() function.

```
export function announceWhoseTurn(room: Room, state: GameState) {
  state.turn_idx+=1
  let player = state.current_player
  console.log("Player Turn: ", player)
  let message = 'It is now ' + getPlayerName(state, player) + '\'s turn.'
  sendMessage(room, state, (message), 'server', true)

  let turnIdx = state.turn_idx
  console.log('Starting %s\'s turn timer.', player)
  let turnTimer = room.clock.setTimeout(() => {
    playerFold(room, state, state.players.get(player))
  }, 60_000); // 60 seconds
  let interval = room.clock.setInterval(() => {
    //console.log('Checking %s\'s turn timer.', player)
    if (turnIdx != state.turn_idx) {
      console.log('Clearing %s\'s turn timer.', player)
      turnTimer.clear()
      interval.clear()
    }
  }, 1000);
}
```

Should the player perform an action on the app, that action is communicated to the server and parsed here. Their action then calls the corresponding function and executes their desired action.

```
export function playerTurn (room: Room, state: GameState, client: Client, turn: any) {
    // ensure its the correct players turn
  if(client.sessionId === state.current_player && state.round_over == false) {
      console.log(client.sessionId, ' is trying to ', turn.action, ' on their turn.')
    // player turn
    if (turn.action == 'fold') {
      // FOLD
      console.log(client.sessionId, " chose to fold")
      playerFold(room, state, state.players.get(client.sessionId))
    } else if (turn.action == 'check') {
      console.log(client.sessionId, " chose to check")
      playerCheck(room, state, state.players.get(client.sessionId))
    } else if (turn.action == 'raise') {
      console.log(client.sessionId, " chose to raise: ", turn.amount)
      //bet code
      playerRaise(room, state, state.players.get(client.sessionId), turn.amount)
    }
```

At the end of the *playerCheck, playerRaise* and *playerFold* functions, the server checks if the stage is over and progresses the game if so. Then, the nextPlayer() function is called which determines the next player and sets the current_player variable to that player's Session ID. In doing so, the next player is able to act. The application listens to this variable and if it matches the client's Session ID, it causes the device to vibrate. This is the only use of vibration in the application as this increases the value of its effect and draws more attention its use.

```
    global.room.state.listen('current_player', (value, previous) => {
        component.setState({current_player: value})
        if (global.room.sessionId === value) {
            // vibrate phone
            Vibration.vibrate()
        }
    })
```

If the round comes to an end, a winner, or several in case of ties or side pots, is determined and the winnings are allocated. This was a challenging feature to incorporate, as it meant devising a method of ranking each hand by numerous properties. As a Poker Hand consists of five cards, it could have up to 5 different deciding factors depending on the combination of cards. Therefore, functions to determine what the ranking of the hand was, as well as a function to then rank hands by secondary parameters had to be created.

To do this a Hand class had to be defined, which took had several variables; *owner, rank, primary, secondary, tertiary, quaternary* and *quinary*. The owner variable consists of a client's Session ID, the rank variable is a number from 1-10, referring to which hand it is(e.g. a Royal Flush is 10, with higher rank meaning better), whereas the next five were used to store any additional parameters that were needed in case of a tie break.

For example, a Full House consisting of 3 Kings and 2 Jacks would have a rank of 6, a primary value of 13 (King) and a secondary value of (Jack). As there are only 2 differing parameters in a Full House. This would win out over a Full House of 3 Jacks and 2 Kings.

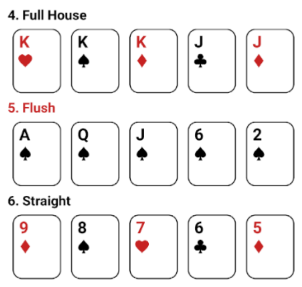


Fig. 8. An example of various hands as shown in the Hand Rankings modal window of the app.

A Flush, however, would have a ranking of 5 and would utilise all five parameters as there are 5 differing values. In Flush in Figure 8, we would have a primary value of 14 (Ace), a secondary of 12 (Queen), a tertiary of 11 (Jack), a quaternary of 6, and a quinary value of 2. Therefore, if, in descending order, the nth value of another Flush hand is matching, then the $(n+1)$th value is compared instead. If the second hand's value is higher, then the second hand wins. If they are the same, the next is checked. This is done as following.

```
function sortHandsByRank(a: Hand, b: Hand) {
  if (a.rank === b.rank) {
    // if ranks are the same, sort by tie break params:
    if (a.primary == b.primary) {
      // etc.
      if (a.secondary == b.secondary) {
        if (a.tertiary == b.tertiary) {
          if (a.quaternary == b.quaternary) {
            // if (a.quinary == b.quinary) {
            //   // tied
            // }
            return a.quinary < b.quinary ? 1 : -1;
          }
          return a.quaternary < b.quaternary ? 1 : -1;
        }
        return a.tertiary < b.tertiary ? 1 : -1;
      }
      return a.secondary < b.secondary ? 1 : -1;
    }
    return a.primary < b.primary ? 1 : -1;
  }
  return a.rank < b.rank ? 1 : -1;
}
```

Each player's hand can be determined by the function findBestHand(), which will return a Hand object containing the respective values. When stored in an array alongside all other playing hands, we can use the aforementioned sortHandsByRank function to list them in order. The findBestHand() function uses several other helper functions to determine the best possible hand, as shown below.

```
export function findBestHand(communityCards: Card[], player: Player) {
  let bestHand = new Hand(player.sessionId)
  let cards: Card[] = []
  let values: String[] = [] // DEBUG PURPOSES
  communityCards.forEach(card => {
    cards.push(card)
    values.push(card.value)
  });
  player.cards.forEach(card => {
    cards.push(card)
    values.push(card.value)
  });
  console.log('Finding best hand: ', values) // DEBUG PURPOSES
  let checkHand = isFlush(player.sessionId, cards)
  if (checkHand.rank > bestHand.rank) {
    bestHand = checkHand
    // check for straight flush
    checkHand = isStraight(player.sessionId, cards)
    // If straight flush, check for royal and assign rank
    if (checkHand.rank === 5) {
      if (isRoyal(checkHand)) {
        console.log('Hand is a Royal Flush')
        checkHand.rank = 10
      } else {
        console.log('Hand is a Straight Flush')
        checkHand.rank = 9
      }
      // Update best hand
      if (checkHand.rank > bestHand.rank) bestHand = checkHand
    }
  }
  // Only continue checking if the hand is not a Royal, Straight or a normal Flush:
  if (bestHand.rank > 8 || bestHand.rank != 5) {
    let checkHand = checkForMultiples(player.sessionId, cards)
    if (checkHand.rank > bestHand.rank) bestHand = checkHand
  }
  return bestHand
}
```

3 Reflection

Developing for a cross-platform nature had several impacts on the project. Though most of this was made easily enabled by the React Native libraries. A notable advantage was the ability to incorporate vibration into my app, allowing for an extra level of user engagement. This wouldn't be possible if developing for a standard React website which simply scales down for mobile devices, thus providing a more specialised experience for the user. Unfortunately, this feature presents some inconsistencies across platforms; on iOS devices the React Native Vibrate module can only be called with a fixed 400ms duration, whereas this is more customisable on Android. Because of this, the default duration was always used to ensure consistency across the application.

A notable disadvantage was that by default, React Native neither supported Linear Gradients in its Styling nor did it support the use of SVGs in its Image components. As a result, additional libraries had to be used to utilise an SVG to provide the Linear Gradient background that was desired. Through *react-native-svg* and *react-native-svg-transformer* I was able to load and use the SVG background. While working seamlessly on an iOS device, it seemed that to incur a small (perhaps around a millisecond) but visible load time on the Android emulator, showing a slight white background extremely briefly before loading into each screen.

As a whole, however the cross-platform nature is massively useful as it allows for the development of an artefact across multiple platforms without having to create entirely different iterations. Some features may have to defined on a per-OS basis, but this is much more desirable than than developing entire applications separately.

4 Supplemental Material

4.1 Project Files

Project files can be found on in the Supporting Documentation section.

4.2 Video Demonstration

Video demonstration can be found on YouTube here: `https://youtu.be/6aV0bMh1ys0`

4.3 Pictures of Deployment

Pictures of Deployment can be found here: `https://imgur.com/a/kT9KKit`

Bibliography

Babich, N. (2018), '10 do's and don'ts of mobile ux design — adobe xd ideas', *Adobe XL* . Accessed: 23/05/2022.
URL: *https://xd.adobe.com/ideas/principles/app-design/10-dos-donts-mobile-app-design/*

Baker, J. (2019), 'Haptic ux — the design guide for building touch experiences', *Medium* . Accessed: 23/05/2022.
URL: *https://medium.muz.li/haptic-ux-the-design-guide-for-building-touch-experiences-84639aa4a1b8*

Improving User Experience · React Native (n.d.), *Adobe XL* . Accessed: 23/05/2022.
URL: *https://reactnative.dev/docs/improvingux*