CITS4401
System Design Document

Badminton Automated Game Scheduling System (BAGSS)

Sarah Connor 21707394
Nerces Kahwajian 21592645
Damon van der Linde 21506136

1st June 2018

Table of Contents

List of Figures

List of Tables

1. Functional Modelling

1.1. Use Case Diagrams

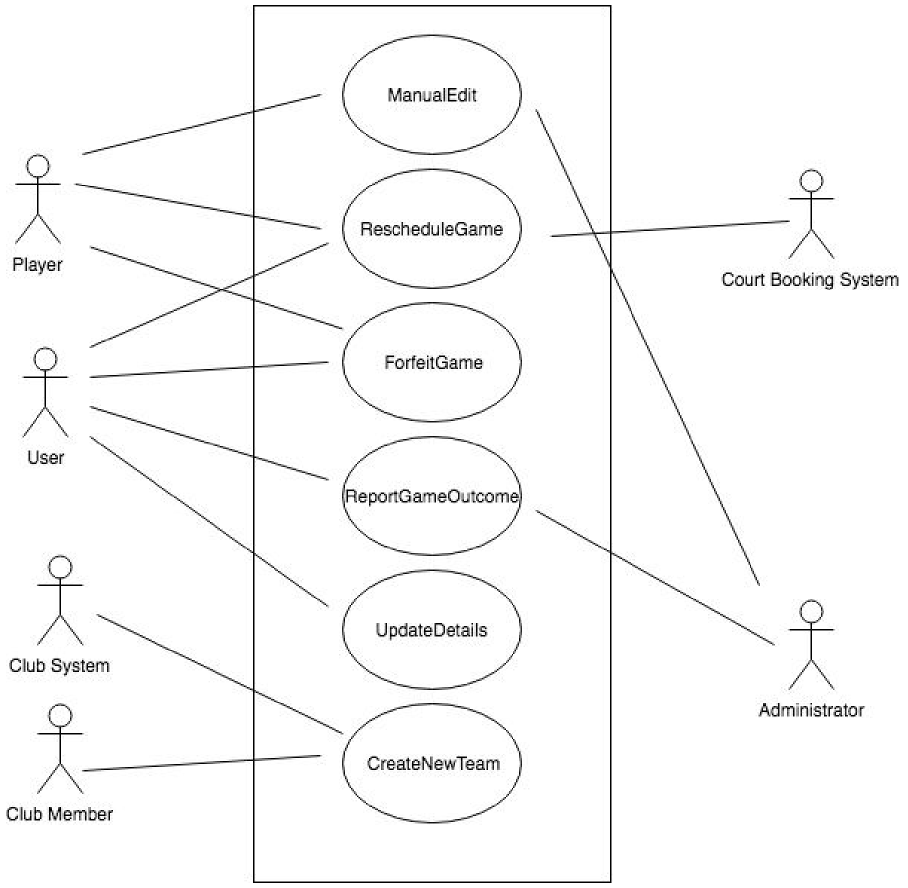


Figure 1 - Overall Use Case Diagram

1.1.1. Reschedule Game

Table 1 - Reschedule Game Use Case

Use case name	RescheduleGame
Participating actors	Initiated by **`User`** Communicates with **`Player1`**, **`Player2`**, **`Player3`**, and **`Court Booking System`**.
Flow of events	1. **`User`** activates the “Reschedule Game” function of the system, and selects the game they intend to reschedule and the time slot

	they would like it to be rescheduled to. 2. **BAGSS** then contacts the other players involved in the game, **Player1**, **Player2**, and **Player3** with the new game time, requesting permission to reschedule. 3. Either **Player1**, **Player2**, and **Player3** will then all respond that they do give permission for the reschedule, or one or more will respond that they do not give permission. 4. If all players give permission, **BAGSS** updates the time of the game with the new time and communicates with the external **Court Booking System** in order to book a court for the new time. It then sends a message to **User**, **Player1**, **Player2** and **Player3** informing them that the reschedule has been successful. If one or more of the players declines the reschedule, **BAGSS** just sends a message to **User**, **Player1**, **Player2** and **Player3** informing them that the reschedule has been unsuccessful.
Entry condition	● **User** is logged into **BAGSS**. ● The use case is initiated before the time of the game and before Tuesday, 17:00.
Exit condition	● **User**, **Player1**, **Player2** and **Player3** all receive the notification that the reschedule has been successful.
Quality Requirements	● The reschedule request arrives to each player no later than 30 seconds after it is sent by **BAGSS.** ● The selected response from each player arrives no later than 30 seconds after it is sent.

1.1.2. Update Details

Table 2 - Update Details Use Case

Use case name	UpdateDetails
Participating actors	Initiated by **User**.
Flow of events	1. The **User** activates the “Update Details” function of the system, selects the detail they wish to update, and inputs the new and correct information. Within this, they also have the option to deactivate their

	account, should they wish to quit the tournament. 2. **BAGSS** then sends a confirmation message to User, with the updated details to inform them the process was successful.
Entry condition	● **User** is logged into **BAGSS**.
Exit condition	● **User** receives the confirmation message.
Quality Requirements	● The confirmation message arrives to **User** no later than 30 seconds after it is sent by **BAGSS.**

1.1.3. Forfeit Game

Table 3 - Forfeit Game Use Case

Use case name	ForfeitGame
Participating actors	Initiated by **User** Communicates with **Player1**, **Player2** and **Player3**
Flow of events	1. The **User** activates the "Forfeit Game" function of the system, and selects the game they intend to forfeit. 2. **BAGSS** then contacts the other players involved in the game, **Player1**, **Player2** and **Player3** and notifies them of the forfeit. It then records the outcome of the game as a loss for **User**'s team and a win for the opposing team.
Entry condition	● **User** is logged into **BAGSS**.
Exit condition	● **User**, **Player1**, **Player2** and **Player3** all receive the notification that the game has been successfully forfeited.
Quality Requirements	● The forfeit notice arrives to each player no later than 30 seconds after it is sent by **BAGSS.**

1.1.4. Report Game Outcome

Table 4 - Report Game Outcome Use Case

Use case name	ReportGameOutcome
Participating actors	Initiated by **User**

	Communicates with **`Player1`**, **`Player2`** and **`Player3`**
Flow of events	1. The **`User`** activates the "Report Game Outcome" function of the system, and selects the game they wish to report the outcome for, input the name of the winning team and the game's score. 2. **`BAGSS`** then sends a message to **`Player1`**, **`Player2`** and **`Player3`** with the name of the winning team and the game's score. 3. If **`Player1`**, **`Player2`** and **`Player3`** wish to dispute the game outcome they can contact the **`Administrator`**, who would communicate with all four players to resolve the issue and update the score on the system if necessary.
Entry condition	● **`User`** is logged into **`BAGSS`**.
Exit condition	● **`Player1`**, **`Player2`** and **`Player3`** all receive the game outcome message.
Quality Requirements	● The game outcome message arrives to each player no later than 30 seconds after it is sent by **`BAGSS.`**

1.1.5. Create New Team

Table 5 - Create New Team Use Case

Use case name	CreateNewTeam
Participating actors	Initiated by **`ClubMember1`** Communicates with **`Club System`** and **`ClubMember2`**
Flow of events	1. **`ClubMember1`** activates the "Sign Up" function of the system and enters the name of their intended doubles partner, **`ClubMember2`**. 2. **`BAGSS`** prompts **`ClubMember1`** to enter in their email address and member number and that of **`ClubMember2`**. 3. **`ClubMember1`** enters the requested details. 4. **`BAGSS`** then communicates with the **`Club System`**, requesting it to verify **`ClubMember1`** and **`ClubMember2`**'s details. 5. **`Club System`** reviews the details and responds affirmative. 6. **`BAGSS`** then emails **`ClubMember1`** and

	ClubMember2 with a username and password which they are free to update later. **BAGSS** then adds this new team to the bottom of the ladder, and incorporates it into the game schedule.
Entry condition	● **ClubMember1** and **ClubMember2** are both members of the badminton club and are both not already apart of the badminton tournament.
Exit condition	● **ClubMember1** and **ClubMember2** both receive the message from **BAGSS** with their username and password.
Quality Requirements	● The message from **BAGSS** with the username and passwords arrives to each club member no later than 30 seconds after it is sent by **BAGSS.**

1.1.6. Manual Edit

Table 6 - Manual Edit Use Case

Use case name	ManualEdit
Participating actors	Initiated by **Administrator** Communicates with **Player**
Flow of events	1. **Administrator** has access to change such aspects of the system such as the leaderboard, players details, games, and the schedule. 2. **BAGSS** however, will send notifications to any **Players** involved in the change. For example, if the **Administrator** changes the time of a game, the players involved in that game will be notified of the change via an automatic message sent by **BAGSS.** The purpose of this is not only to keep the **Players** informed but also to keep the actions of the **Administrator** transparent, in case they make an error or purposefully tamper with the system.
Entry condition	● **Administrator** is logged into **BAGSS.**
Exit condition	● The **Players** receive the notification message from **BAGSS.**
Quality Requirements	● The notification message arrives to each **Player** no later than 30 seconds after it is sent by **BAGSS.**

2. Object Modelling

2.1. Class Diagrams

2.1.1. Main Class Diagram

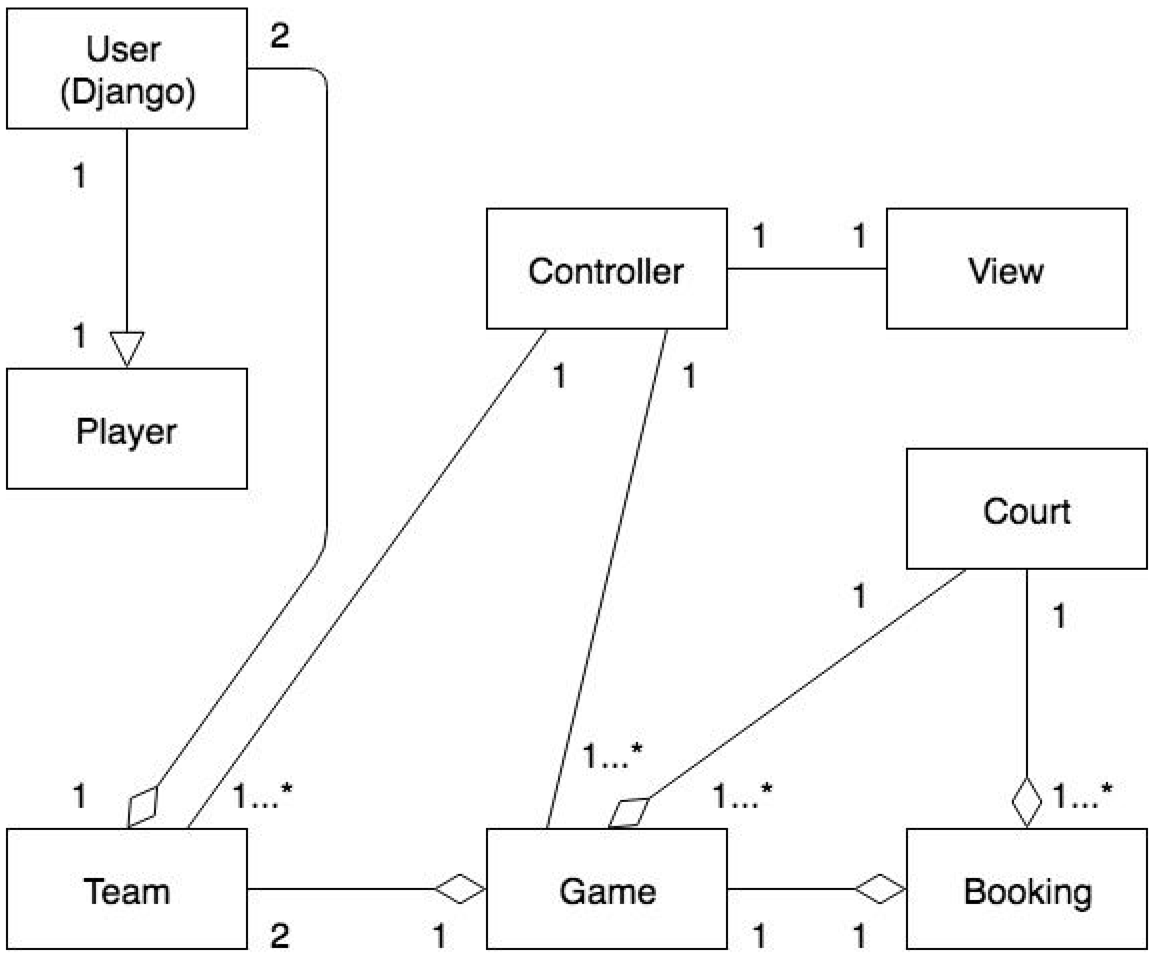


Figure 2 - Main Class Diagram

We have eight classes in our system, Controller, View, Court, Booking, Game, Team, Player and User (a Django class). The relationships between these classes are listed and explained below.

Table 7 - Class Relationships

Classes	**Relationship**	**Explanation**
Player and User	One-to-one inheritance	Player inherits attributes of User and for every User there is one Player class and vice versa.
User and Team	Two-to-one shared aggregate	Team is always composed of exactly two User, however a User can exist without a team so it is not a composition aggregation relationship (e.g. before the User is assigned a team, the User object can still exist independently).
Team and Game	Two-to-one shared aggregate	A game can only ever be composed of two Teams, but the Team object can also exist without a Game object (e.g. not every Team plays every week).
Game and Booking	One-to-one shared aggregate	Each Booking is composed of exactly one Game. It is not a composite relationship as Game objects can exist independently of Booking objects (e.g. once a game has finished the Game object still exists even though the Booking object is destroyed).
Court and Booking	One-to-many shared aggregate	One Court can have multiple Bookings and each Booking is composed of one Court. Court objects can also exist independently of Booking objects (e.g. before a game booking is made and after the game is complete, the court still exists).
Court and Game	One-to-many shared aggregate	Courts can have multiple Games played on them per week, but each Game can only be played on one Court. Each Game is composed of a Court but as mentioned above, Court objects can exist independently of Game objects.
Controller and Team	One-to-many link	Controller is able to access multiple Teams at a time, as it can access the entire Team's table in the database at once. This relationship is only a link as neither class shares attributes.
Controller and Game	One-to-many link	Controller is able to access multiple Games at a time, as it can access the entire Games table in the database at once. This relationship is only a link as neither class shares attributes.
Controller and View	One-to-one link	There is only ever one View object in the system. Likewise with the Controller object, thus they have a one-to-one relationship. Their relationship is also only a

		link as they do not share any attributes.

2.1.2. View Class Diagram

Table 8 - View Class (views.py)

View

index(request)
profile(request)
manageGames(request)
account(request)

The view class is used in our system to present the different pages of the system to the user. This class makes use of Django's rendering object to showcase the different HTML files. The functions contained within this method refer to the different pages such that the index(request) method returns the user's home page, the profile(request) method returns the user's profile page, the manageGames(request) method returns the user's manage game page and account(request) method returns the user's account page.

2.1.3. Controller Class Diagram

Table 9 - Controller Class (db.py)

Controller

get_teamTable(request)
get_gameTable(request)
get_not_played_games(request)
get_games_where_player_involved_have_been_played(request
)
get_games_where_player_involved_not_been_played(request)
get_common_availability_player(player1, player2)
get_common_availability_team(team1, team2)
get_common_availability_with_court(game, court)
schedule_match(game, court)
update_availability(request)
delete_account(request)
deactivate_account(request)

The controller class features a series of methods to interact with the various game's objects of the systems. The method name get_teamTable(request) returns the team object of the current player through utilizing the request object within Django that contains the user's ID. The next method named get_gameTable(request) returns all the game objects. The get_not_played_games(request) method requests all the games for the user using the request object from Django where the game has no winning team as well as having not been marked as played yet while the next method does a similar thing however, it returns the amount of matches that have not been played. The method get_games_where_player_involved_have_been-_played(request) returns all the matches which the player was involved it that has been played (finished). The method get_games_where_player_involved_not_been_played(request) returns all the matches where the player has been involved but the matches have not been played. The class then features a series of methods starting with the prefix get_common_availability_ followed by the subject relating to it. These methods will simply find a common set of availability between the specified objects and return them. The method named schedule_match takes in a game instance as well as a court instance and schedules the match. The time which the court should be booked is contained within the game instance passed to this function. The method named update_availbility, updates the current set of available times for the specified object which is contained within the request parameter. The method named delete_account(request) flags the user account for deletion within a certain time period unless the user contacts the administrator to undo this event. The method named deactivate_account() simply makes the player unavailable to the system, meaning the player will not be scheduled into any future matches as well as being removed from the current matches the player is involved in.

2.1.4. Team Class

Table 10 - Team Class (models.py)

Team
player1: User (Foreign Key) player2: User (Foreign Key) total_wins: Integer (default = 0) leaderboard_position: Integer (default = teamID)

Teams within the system are represented as an object of the Team Class. The attributes of this class involved a references to the object of the first player from the team (team1: Team [Foreign Key]), as well as a reference to the object of the second player from the team (team2: Team [Foreign Key]), the number of overall wins which the team has achieved (total_wins: Integer) and finally the position of the team on the current leaderboard (leaderboard_position: Integer).

2.1.5. Court Class

Table 11 - Court Class (models.py)

Court
name: CharField(max_length = 100) address: CharField(max_length = 100)

Each available court will be represented in the system as an object of the Court Class whose attributes consist of the name of the court (name: CharField[max_length = 100]) as well as the address of the court (address: CharField[max_length = 100]).

2.1.6. Game Class

Table 12 - Game Class (models.py)

Game
date: DateTimeField week: Integer(default = 1) has_been_played: BooleanField(default = False) team1: Team (Foreign Key) team2: Team (Foreign Key) winning_team: Team (Foreign Key) court: Court (Foreign Key)

Each game will be an object in this class. The attributes of this class are the date of the game (date: DateTime), the week in which the game takes place (week: Integer), indicator of if the game has been played (has_been_played: BooleanField), team1 which references the team object of the first team apart of the match (team1: Team [Foreign Key]), team2 which references the team object of the second team involved in the match (team2: Team [Foreign Key]), winning team which references the object of the team that won the match (winning_team: Team [Foreign Key]) and a reference to the court object of the court which the game is being conducted at (court: Court [Foreign Key]).

2.1.7. Player Class

Table 13 - Player Class (models.py)

Player (Extension of Django's User Class)
team: Team (Foreign Key)

availability: String

Since we are utilising the Django framework within our design the player class is simply an extension of Django's user class with and extra couple of attributes. The basic class features attributes such as first_name which refers to the name of the person, last_name which refers to the family name of the person, username which refers to chosen display name, the individuals email address and then from the extension, the object of the team which the user belongs to (team: Team [Foreign Key]) and a string representing the current month and the following month's availability through a 0 (not available) and a 1 (available) separated by a comma.

2.1.8. Booking Class

Table 14 - Booking class (models.py)

Booking
court : Court (Foreign Key) game: Game (Foreign Key) start: DateTimeField end: DateTimeField

Each court's availability will be represented by a booking object which correspond to the times where the court is not available. Attributes of this class include a reference to the court object which the booking applies to (court: Court [Foreign Key]), a reference to the game object which the booking belongs to (game: Game [Foreign Key]), the starting time of the match (start: DateTimeField) and the ending time time of the match which is 90 minutes after the start time (end: DateTimeField).

3. Dynamic modelling Part A

3.1. Sequence Diagram 1

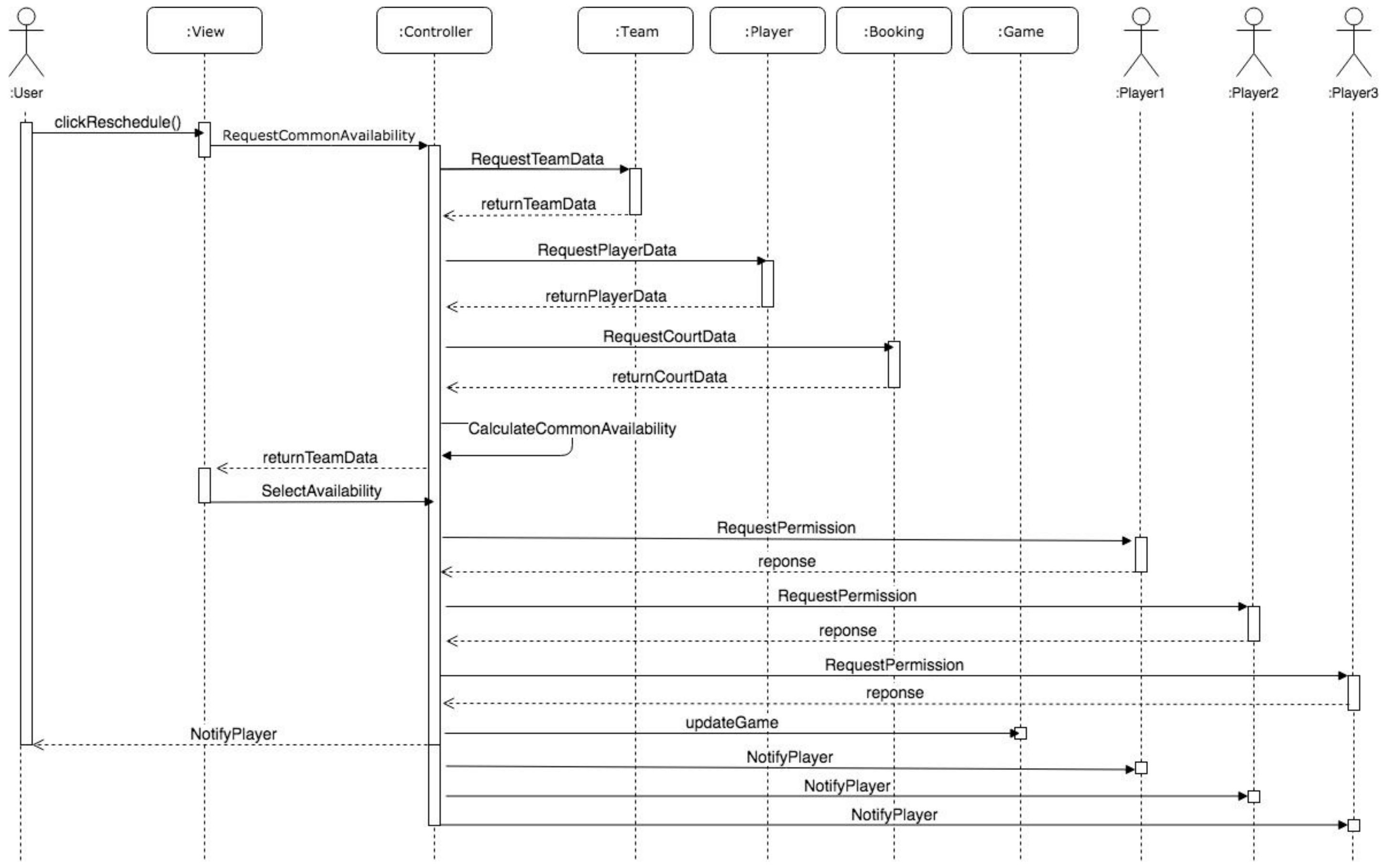


Figure 3 - Reschedule Game Sequence Diagram (Sarah)

This sequence diagram details the process that occurs within the system during the Reschedule Game Use Case. For this particular case, we are assuming that the reschedule was successful. There are six classes, View, Controller, Team, Player, Booking and Game, and four actors, User, Player1, Player2 and Player 3. involved in this use case. First, the User will clickReschedule(), meaning they will activate the reschedule option on the interface. Therefore this communicates with the View class, as this is the visual interface of the system and thus is a boundary object. Next, we essentially want the interface to display to the user the possible dates for the reschedule.

However, in order to do this some information needs to be retrieved and a calculation needs to be performed. Therefore, View will then send a RequestCommonAvailability request to Controller, with the Game instance it wishes to reschedule. Essentially it wants to receive back a list of all the available dates for the reschedule. The controller class, which is a control object, then breaks this down into the smaller internal steps needed to get this list. It gets the Team IDs from the Game instance it was given and uses this to request information from the Team class, which is an entity object, in order to get the names of the player on both teams. Once it has this it uses these player name to request the availabilities of the players from the Player class, which is an entity object. Then it requests the court availabilities from the Court class, which is also an entity object. It performs CalculateCommonAvailablility where it essentially cross-references the availabilities of the four players and the courts to determine a list of game times when everyone is available and when there will also be a court available to play on.

It then returns this list back to the View class, where it is displayed to the user who selects which time they wish to reschedule the game to (SelectAvailability). This is then returned to the Controller class which sends notifications to the three players involved in the game to which they send a response, giving their permission for the change. Once Controller has received all of these replies, it then sends the new time and date to the Game class (updateGame), which is an entity object. After this is just send a NotifyPlayer message to the User and the three Players to inform them that the reschedule was successful.

3.2. Sequence Diagram 2

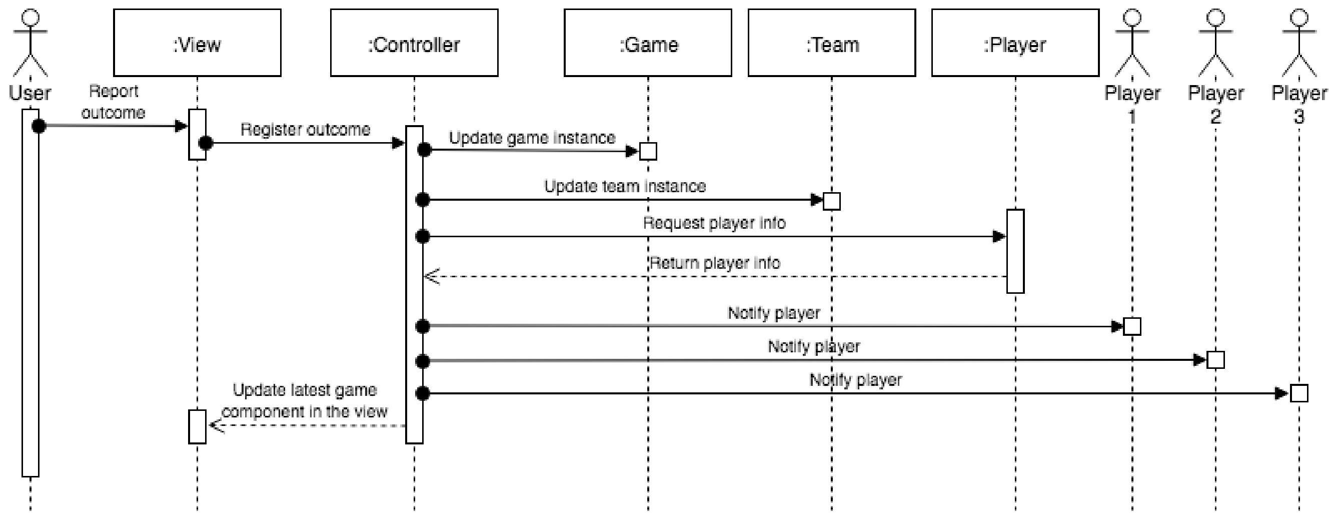


Figure 4 - Reporting Match Outcome Sequence Diagram (Damon)

Actor in the sequence diagram above is the user using the system. The actor interfaces with the boundary object which in this instance is the view (webpage). The control object in the system is the controller class. The user in this instance interacts with the view pressing the win or lose button for the latest match which then sends the register outcome command to the controller which then orchestrates a series of executions to complete the given task. These executions involve interacting with various other entities such as game, team, and the player updating attributes within. Once all the data manipulation executions have been executed the other object(Player 1, Player 2 and Player 3) are all notified of the reported game outcome.

3.3. Sequence Diagram 3

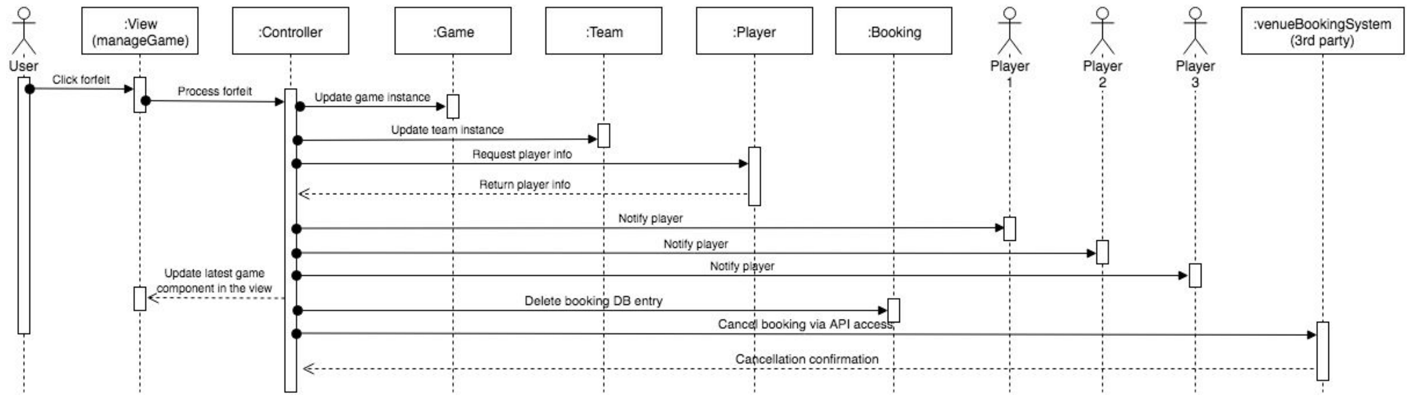


Figure 5 - Forfeiting Match Sequence Diagram (Nerces)

In figure 5 the actors are the User, Player 1 (Entity), Player 2 (Entity), and Player 3 (Entity) while the objects are the View (Boundary), Controller (Control), Game, Team, Player, Booking, and the 3rd party venueBookingSystem. The main messages sent between the objects are the “player info” and “cancellation confirmation”.

4. Design Constraints

4.1. Priority

1) End User Criteria
2) Reliability
3) Maintenance
4) SQL injection prevention (Dependability)
5) Performance

4.2. Descriptions

4.2.1. Performance

In order to achieve our high performance goals, outlined in the RAD document, some constraints will have to be placed on our system. Therefore, players have been limited to only being able to set and edit their availability for the current and following month. This is to help limit the length and amount of data that needs to be processed as having too much data will increase the search time hence negatively impacting system performance.

4.2.2. SQL injection prevention (Dependability)

Due to the nature of the environment which the project will operate in it is important to consider the security of the system. Interaction with the database requires a design such that SQL injection attacks would not work. Fortunately, the chosen framework (Django) already has a syntax designed for this purpose which ensures that injection actions cannot be done.

4.2.3. End User Criteria

The system will be used by people with varying degrees of technical knowledge, it is vital that the user interface is easy to use and navigate, especially since we are assuming users will not be provided with any training. In order to achieve this, the user interface will need to be designed using the principle of least astonishment, which states that user interfaces are easiest to use when they are not surprising i.e. when a first-time user sees the interface they are not surprised by how it works. This therefore reduces the learning time for users. This is achieved by making the user interface resemble interfaces the user has likely encountered before such as Facebook and Netflix. The design of our user interface is then constrained by having to be both simple and familiar to users.

It is also of some importance that users are protected from making errors. Therefore the user interfaces of the system should prevent users from issuing incorrect requests. For example, users will be prevented from requesting to reschedule a game to a time when they have specified they are unavailable, by having this option unavailable on the user interface.

Having multiple users concurrently making requests of the system could also cause problems, and thus this will need to be constrained. For example, if one player requests to reschedule a game, and then a few minutes later another player playing in that game (so either the partner or one of the members of the opposing team) is wishing to reschedule the same game before the initial reschedule request has been completed, then the user interface should automatically prevent them from doing this, while also informing them of the reason why they cannot request a reschedule at that time.

4.2.4. Maintenance

The system will have a designated administrator who is responsible for the maintenance of the system and resolving any issues or conflicts that are outside the control of the system (such as team disputes). This administrator has access to edit all databases such as the leaderboards and the player details. Hence, the design has to encompass this notion of user privilege access to certain functions.

4.2.5. Reliability

To achieve reliability goals, proper programming practices need to be implemented to avoid system issues from arising that could cause a failure. Such failures that could arise from poor practices include memory leaks, security issues that could cause downtime, and caching issues that slowly build up over time. Hence, it is crucial that programming practices are listed as a design constraint.

5. Subsystem Decomposition

5.1. Controller component subsystems

5.1.1. Model subsystem

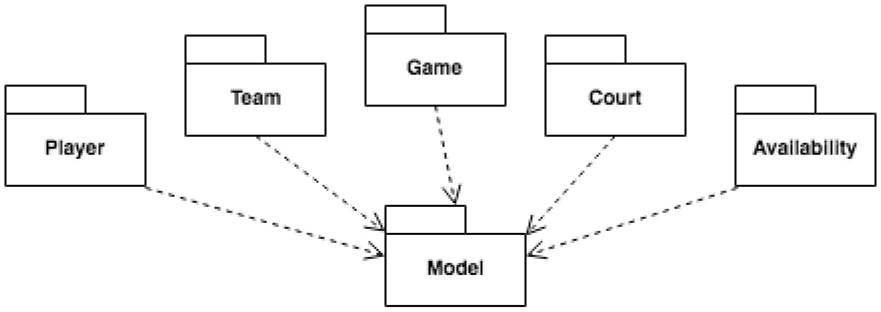


Figure 6 - Model Subsystem

The use of a model view controller software architecture allows for logical grouping of classes. Classes such as player, team, game, court, and bookings contain references to each other and as such are related which allows for this subsystem to offer high cohesion. The model subsystem is also responsible for the domain knowledge of the overall system.

5.2. User interface

5.2.1. Web and mobile interface

5.2.1.1. Login page

The idea was to design the login screen with a minimalistic approach to minimise the potential for user errors while still having all the integral components such as the login and password retrieval and an option to remember the user. Refer to Appendix A and F.

5.2.1.2. Homepage

The minimalistic approach was also applied to the homepage. The aim was to have all the important content in plain sight as to avoid users having to play around with menus to find what they need. The homepages consists of the leaderboard, details of the player's next match and a menu bar that will be shown across all the pages. The menu bar consists of the following:

- Home button

    - When clicked, will direct a user back to the homepage
- Profile
    - This is where users can update their details such as name, number, and availability
- Manage Games
    - Users can see details of their games, record a win, forfeit games, request game reschedules, and view their game history
- Account Settings
    - Users can change account details such as their email address, password, and deactivate/delete their account
- Logout
    - When clicked, will safely logout a user from the system

Refer to Appendix B and G.

5.2.1.3. Profile page

The profile page is where the user can change their details such as Name, Surname and Mobile. The player's availability is also updated on this page. As seen below, available days are marked with a green circle around the date. If a player would like to add or remove a date, they will simply click on the date and a circle will either be added (available) or removed (unavailable). Clicking the "update" button will then save their changes. Refer to Appendix C and H.

5.2.1.4. Manage Games page

The manage games page allows users to see their match history, who they versed and whether they won or lost. The latest match section shows the player's most recent match details such as date, time, court number, and opposing team. This is also where the player can mark their match as a win or a loss. This page also shows the player's next match details such as date, time, court number, and opposing team. Players can also forfeit and request to reschedule their matches here. Refer to Appendix D and I.

5.2.1.5. Account Settings page

The account settings page is where players can change their details such as email address and password. They can also deactivate their account or delete it permanently. The difference is that deactivating your account could be for just a season that you do not want to compete in while deleting your account is permanent. Refer to Appendix E and J.

6. Dynamic modelling Part B

6.1. Reschedule Game Statechart

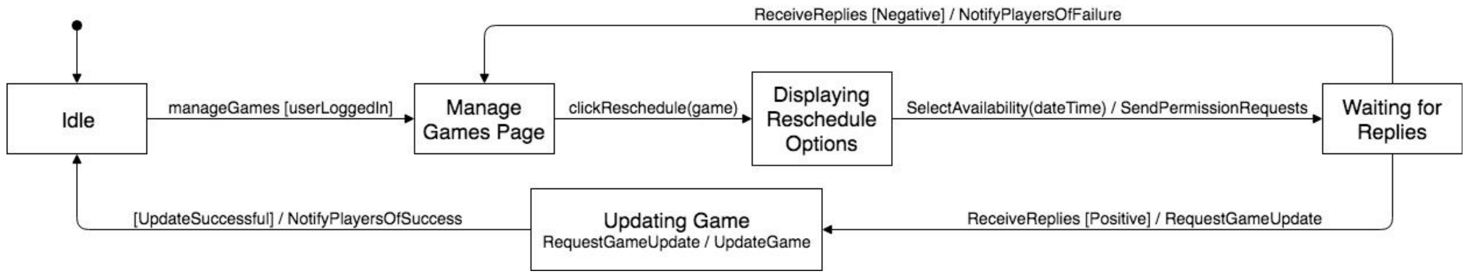
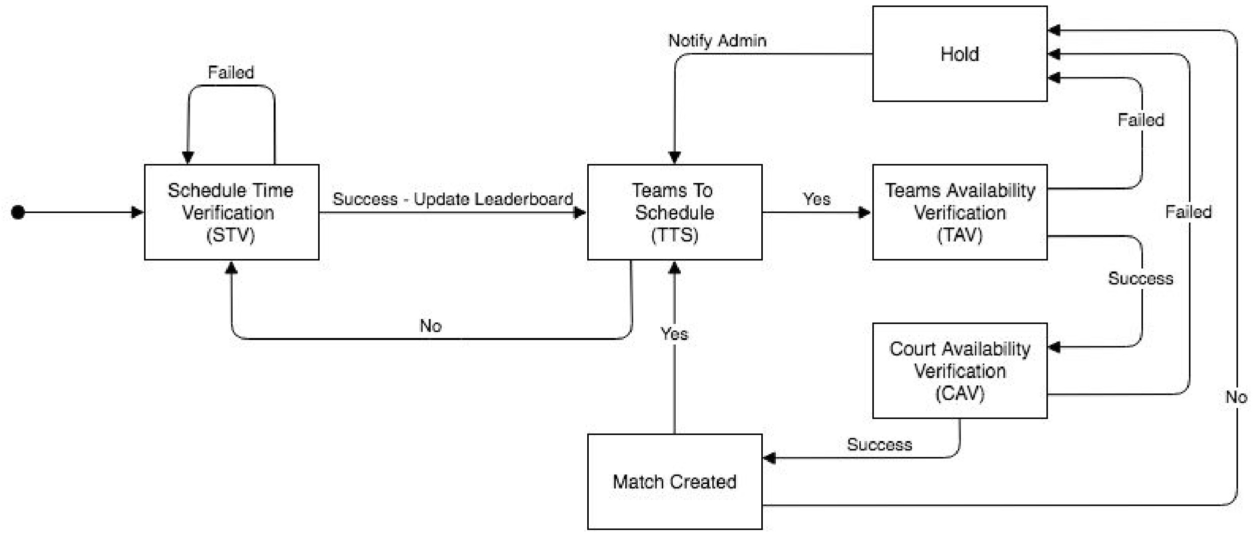

Idle
manageGames [userLoggedIn]
Manage Games Page
clickReschedule(game)
Displaying Reschedule Options
SelectAvailability(dateTime) / SendPermissionRequests
Waiting for Replies
ReceiveReplies [Negative] / NotifyPlayersOfFailure
ReceiveReplies [Positive] / RequestGameUpdate
Updating Game
RequestGameUpdate / UpdateGame
[UpdateSuccessful] / NotifyPlayersOfSuccess

Figure 7 - Reschedule Game Statechart (Sarah)

The above statechart is modelled according to the RescheduleGame use case. The first state of the process is "Idle" which is just the system waiting. Then the user will navigate to the Manage Games Pages. Obviously it is a condition that they are logged into the system in order to do this. They would then select to reschedule the game, putting which game they wish to reschedule as an attribute. This changes the state to Displaying Reschedule Options which essentially displays the available reschedule dates to the user. The user will then select the available date they wish to reschedule to. This event causes the action of SendPermissionRequests, which simply sends a request to reschedule to the other three players involved in the game. The state will then change to Waiting for Replies, while the system waits for the player's replies.

The next event will then be ReceiveReplies, which is just receiving all three replies. There are two options here. The first is that one or more of the replies will be negative (ReceiveReplies [Negative]) in which case this will trigger the action NotifyPlayersOfFailure which just sends a notification to the three players and the user, informing them that the reschedule has failed. The state will then return to the Manage Games Page, at which point the user can repeat the process again if they wish. The other outcome is that all replies will be positive (ReceiveReplies [Positive]) in which case the action caused will be to request a game update and the state will change to Updating Game. Within the Updating Game state, the game will be updated, and then once the condition of the update being successful has been met, this will trigger the action of NotifyPlayersOfSucess, which sends a notification to the three players and the user, informing them the reschedule has been successful. The state then returns back to Idle.

6.2. Scheduling games statechart



Hold
Notify Admin
Failed
Schedule Time Verification (STV)
Success - Update Leaderboard
Teams To Schedule (TTS)
Yes
Teams Availability Verification (TAV)
Failed
Failed
Success
No
Yes
Court Availability Verification (CAV)
No
Success
Match Created

Figure 8 - Scheduling Games Statechart (Damon)

The subsystem of BAGSS which handles the scheduling component enters a state known as **Schedule Time Verification (STV)**. During STV this subsystem is placed in a holding state where it waits until the scheduling time has been reached where it then proceeds to the **Teams To Schedule (TTS)** state. During the transition between those states, the system updates the leaderboard which alters the position of the teams based on their previous game outcomes. Within TTS state the system checks if there are any teams remaining to be scheduled. If there is, the system continues to the **Team Availability Verification (TAV)** state else the system returns to the STV state. During TAV, the system computes a common set of availabilities between the competing teams for the new match to be scheduled based on the common set of availabilities within the teams. If the system is unable to compute any common set of availabilities, the match is placed in the holding state. From the holding state the system transitions back to TTS notifying the administrator of the failure. If the system was able to establish a common set of availabilities it proceeds to the **Court Availability Verification (CAV)** state. During CAV, the court schedules are analysed based on the common set of availabilities formed in TAV state. If there are more than one possible court option then one is chosen at random and the system creates the match. If there are no available courts or the match creation failed the match is placed on hold where the administrator is notified during the transition from the holding state to the TTS state.

6.3. Expired Booking Checker

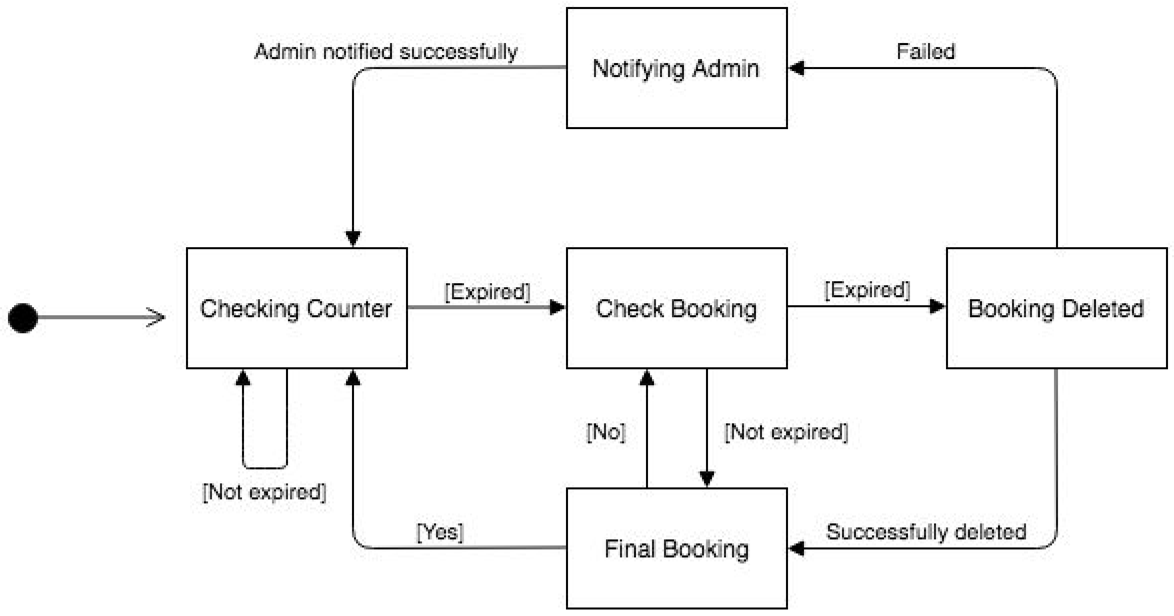


Figure 9 - Expired Booking Checker (Nerces)

Figure 8 is a model of the expired booking checker. From the initial state it will move forward to check if the 2 hour counter has expired. If so, it moves to the “check booking” state where it then checks whether the booking is expired or not. If the booking is expired, it moves to the “booking deleted” state and confirms whether the delete was successful or not. If the delete was successful it then moves to the “Final Booking” state which checks if it is the final booking in the list of bookings. If it is the final booking, it goes back to the “checking counter” state and if it is not the last booking, it then moves to the “check booking” state and starts the process over from that point. This is repeated until all bookings are checked.

7. Design Patterns

7.1. Facade Design Pattern

Table 15 - Facade Design Pattern (Nerces)

Name	Facade Design Pattern
Problem Description	Reduce coupling between a set of classes and the rest of the system. An example of this would be scheduleGame
Solution	The Facade class for the scheduleGames subsystem will work by invoking the methods of the lower level classes within

	scheduleGames. This means that the administrator will not need to access the lower level classes directly. Using a Facade pattern means that the system will be layered.
Consequences	- Shield from low level classes of the subsystem - Helps simplify use of the subsystem - Allows use of closed architecture

7.2. Strategy Pattern

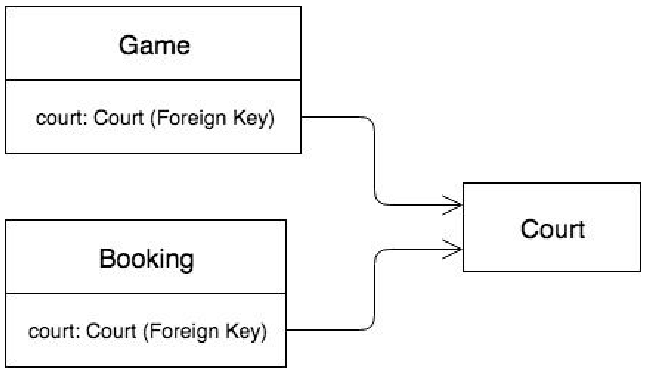
Table 16 - Strategy Design Pattern (Damon)

Name	Strategy Design Pattern
Problem Description [2]	Encapsulating algorithms to allow for selection at runtime. The main notion behind this involves referencing code from a data structure and retrieving it at runtime].
Solution	The strategy pattern is already implemented within the view and the controller systems that compose the Model View Controller architecture. The view component of the system is defined as being an object that is setup with a main strategy surrounding it. In this instance the view is only concerned with the rendering of the website alongside all its visual elements. Any decisions that have to be made in respect to the actual behaviour of the application gets delegated by view to the controller. This results in the view itself being decoupled from the model since it does not know how to get the task done rather it simply passes it to the controller who is responsible for this logic.
Consequences [3]	● Ease of introducing new algorithms ● Implementation of algorithms does not affect context ● Application needs to be aware of all the strategies ● Strategy objects can be shared between multiple context objects

7.3. Observer Design Pattern

Table 17 - Observer Design Pattern (Sarah)

Name	Observer Design Pattern
Problem Description	In our system there is a lot of dependency between the classes (e.g. four of the eight classes need information from other classes),

	therefore keeping attributes consistent across the classes would pose a problem. For example, Court has a one-to-many relationship with both the Game and Booking Class and both the these classes are dependent on Court. Therefore, if the Court address is updated, there is a possibility it will not be updated with Game and Booking and players may turn up for their game at the wrong address.
Solution	To solve this problem, the observer design pattern can be used. When a class attribute is updated, all dependent classes that contain this attribute need to be updated automatically. Therefore for the issue between Court, Game and Booking classes, both Game and Booking need to be updated if the Court information is changed. To solve this problem we have made it so there is a direct reference to the Court object within the Game and Booking objects, as simply depicted below.  Game court: Court (Foreign Key) Booking court: Court (Foreign Key) Court
Consequences	● Both the Game and Booking classes contain an attribute called “court” that is a Court Object and is thus a direct reference to a Court Object as a foreign key. ● Therefore, it is a Court Object contained in the Game and Booking classes is updated, there is essentially no need to update the Game and Booking classes as they will automatically now reference the changed Court object.

8. System Design Rationale

8.1. Choice of Architecture

The initial issue encountered by the development involved determining a suitable software architecture in which to construct a solution to the proposed system concept. Two main design architectures were evaluated. [4]

8.1.1. Object oriented architecture

Object oriented approach involves treating every system and components of that system as an individual object. Most local application utilises such an approach. The advantages surrounding such an architecture involves high code reusability, real world modeling capabilities, and reduced maintenance. However, for a web based application a difficult aspect is to ensure certain objects are not directly accessible by the user [4,5].

8.1.2. Model View Controller architecture

Model View controller architecture involves separating application logic of the system into 3 main parts. This notion allows for modularity, collaborations as well as reuse. The View component of the system defines the graphical user interface which the user interacts with. The Controller component implements the logic of the system accepting inputs received through the view. The Model component defines the structures within the system and is usually referred to as the domain knowledge of the system [1]. The development team settled on this architecture as it allowed for scalability, modularity, and reusability.

8.2. Choice of frameworks for implementing the architecture

Once the issue was resolved surrounding the architecture choice, the next issue that arose involved the choice of framework for implementing the architecture. There were many possible choices however the main two that were examined are provided below.

8.2.1. Ruby on Rails

Ruby on Rails is one of the most common model view controller frameworks used throughout the industry however the development team's limited experience using it as well as various time constraints made it an unsuitable choice for the project.

8.2.2. Python and Django

Django is another popular model view controller framework that has also been proven within industry. One of the main advantages of this framework is, it uses Python which the development team had a reasonable level of expertise in so the learning time was limited to simply exploring the documentation of the Django framework.

8.3. Representing availabilities

Due to framework limitation the original idea of storing availabilities as an array was not going to work and as such a different approach was required. The development team finalised it to two main approaches which will be explored in detail below.

8.3.1. Grouping all availabilities into one table

This approach involved constructing a table which contains columns that specified the following attributes: type (Player, Court), game identifier values if the type is of court otherwise it just remains null, start date and an end date. However this approach suffered from unclarity within the table.

8.3.2. Separate Tables and incorporating availabilities into classes

The second approach involved splitting the availabilities into separate tables; court bookings table and a player availability table. Within the court bookings table, only the dates which the court is booked for gets stored. When a match is forfeited or the date has passed that of which the booking was made for, it gets removed from the table. Any new bookings can be validated by simply checking if it conflicts with any of the durations between the start and the end timing of the bookings present. The player availability based on prior assumptions on the day which they are available are recorded as a string within the player object for the current month as well as the following month.

9. Implementation/Prototype

9.1. Requirements and instructions

9.1.1. Dependencies

The prototype is based on Python3 and Django files and as such in order to run and test its functionality, ensure Python3 and Django has been installed on the system.

9.1.2. Prototype execution information

Navigate to folder called Django Website followed by mysite using terminal. Then execute the following line of code: Python3 manage.py runserver. This should result in a message in the terminal window which will indicate whether or not it was a success. To access the prototype navigate to your browser and enter in the following: http://localhost:8000/. The prototype contains various simulated information in order to demonstrate certain functionality and as such a special user has been created alongside the necessary information needed to showcase the functionality. Instructions will be provided to access implemented functionality below.

9.2. Accessing implemented functionality

9.2.1. Leaderboard

When initially navigating to the base url, the user is provided with the homepage which contains a navigation bar with two buttons (Home, Login) as well as a leaderboard which publicly displays the current team's statistics such as their rank, overall number of wins, team number, and the names of the team's players. The navigation bar changes depending on the login status of the user. If the user is logged in the navigation bar will feature more options including profile, manage games and account settings.

9.2.2. Login system

From the homepage the user can either click on the Login button or access the login page through the following url: http://localhost:8000/login/. On this page the user is prompted to enter their username and password. If they do not match the corresponding entries in the system the user is informed to enter a correct username and password. For majority of the following features below Jen has to be logged in which can be achieved by entering ***user3*** into the **username field** and ***user3password*** into the **password field**. On successful login the user is redirected to the homepage where a welcome back message is displayed.

9.2.3. Profile information

From logging into user3's account (done by following the details noted above) the profile page can viewed by clicking on the profile button within the navigation menu. The current logged in user's information such as username, first name, last name, email, team they are apart of, and last login attempt is retrieved and displayed.

9.2.4. Account settings

While being logged in as user3 the account settings page can be accessed by either clicking on the account settings button within the navigation bar or navigating to http://localhost:8000/accountSettings/. On this page an interface is provided for changing the user details however the backend functionality has not been implemented in its current state.

9.2.5. Manage games

While still being logged into user3's account, the manage games page can be accessed through the navigation bar. On this page the user will be presented with their match history of completed games, details of the latest match where match outcome can be set using the buttons (functionality behind them has not been implemented yet) and upcoming match details if it has been scheduled alongside providing buttons for forfeiting and rescheduling (functionality behind them has not been implemented yet).

9.2.6. Administrator functionality

Due to the prototype utilising the Django framework, the administrator functionality is already built in. The administrator login portal can be accessed using http://localhost:8000/admin/ where the **username field** has to be filled in with ***admin*** alongside the **password field** containing ***adminpassword***. All the data of the different classes can be directly accessed and modified from within this portal.

10. References

[1] "MVC architecture", MDN Web Docs, 2017. [Online]. Available: https://developer.mozilla.org/en-US/Apps/Fundamentals/Modern_web_app_architecture/MVC_architecture. [Accessed: 01/06/2018].

[2] T. Sellarès, "The Model View Controller: a Composed Pattern", Girona, 2006. [Online]. Available: http://ima.udg.edu/~sellares/EINF-ES1/MVC-Toni.pdf. [Accessed: 01/06/2018].

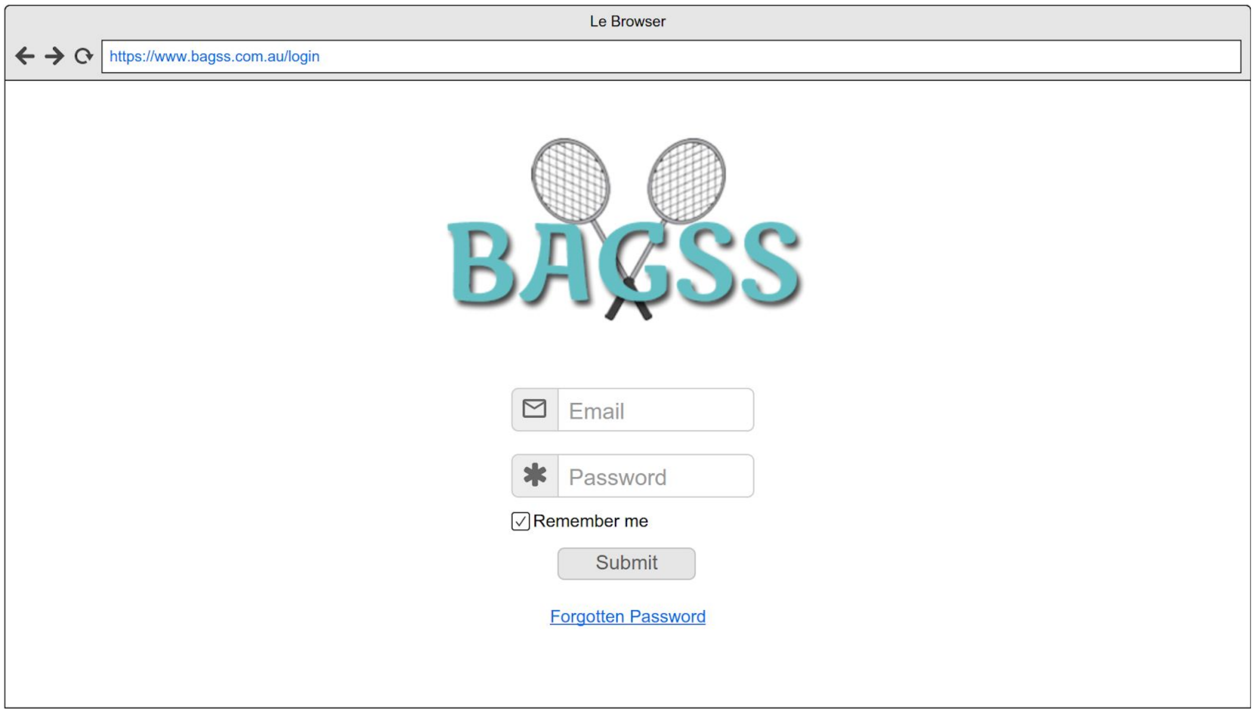
[3] K. Ako-Aboagye, "Strategy Pattern", Bobcash20.blogspot.com, 2007. [Online]. Available: http://bobcash20.blogspot.com/2007/05/strategy-pattern.html. [Accessed: 01/06/2018].

[4] D. Burleson, "Advantages and Disadvantages of Object-Oriented Approach", Dba-oracle.com. [Online]. Available: http://www.dba-oracle.com/t_object_oriented_approach.htm. [Accessed: 01/06/2018].

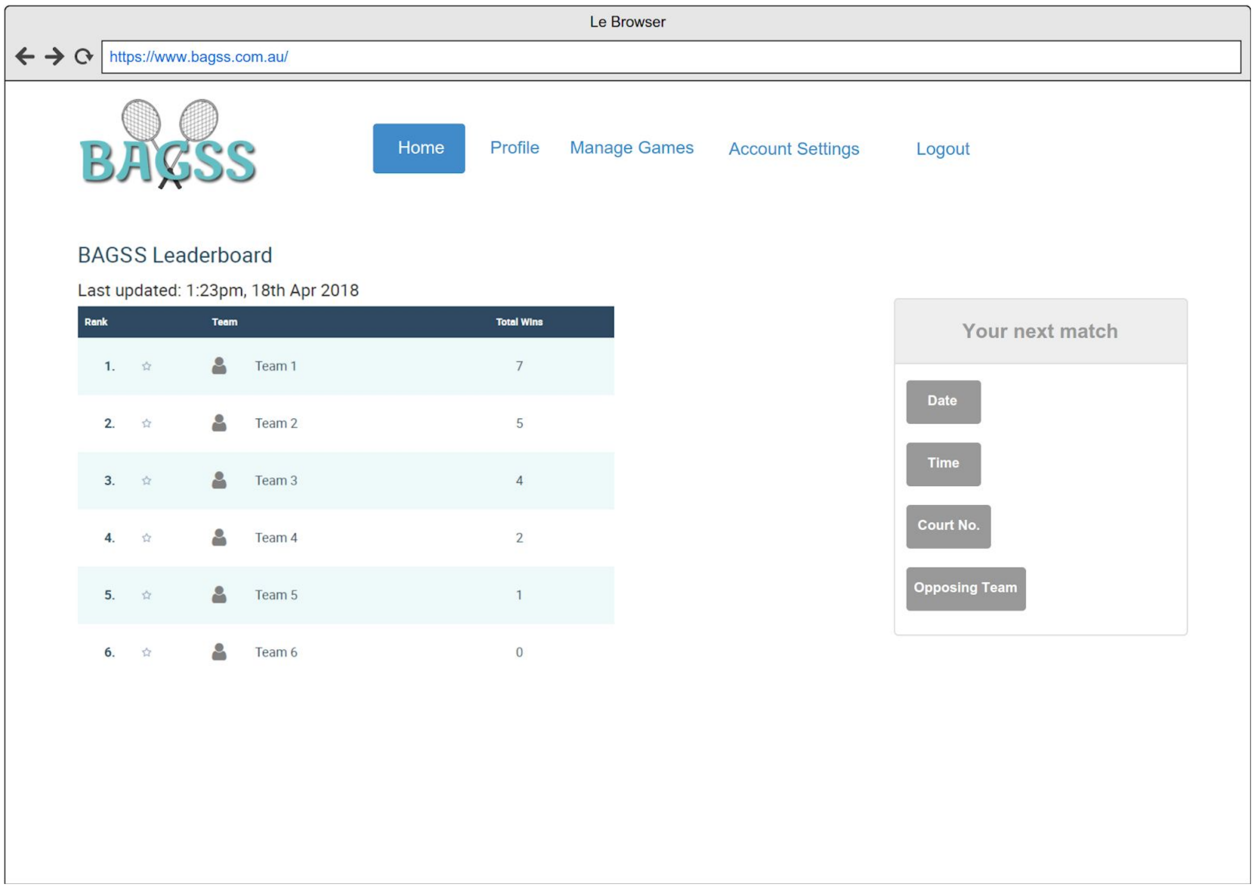
[5] D. Schwabe and G. Rossi, "An object oriented approach to web-based applications design", Theory and Practice of Object Systems, vol. 4, no. 4, pp. 207-225, 1998.

11. Appendix

11.1. Appendix A - Web interface - Login Page



11.2. Appendix B - Web interface - Homepage

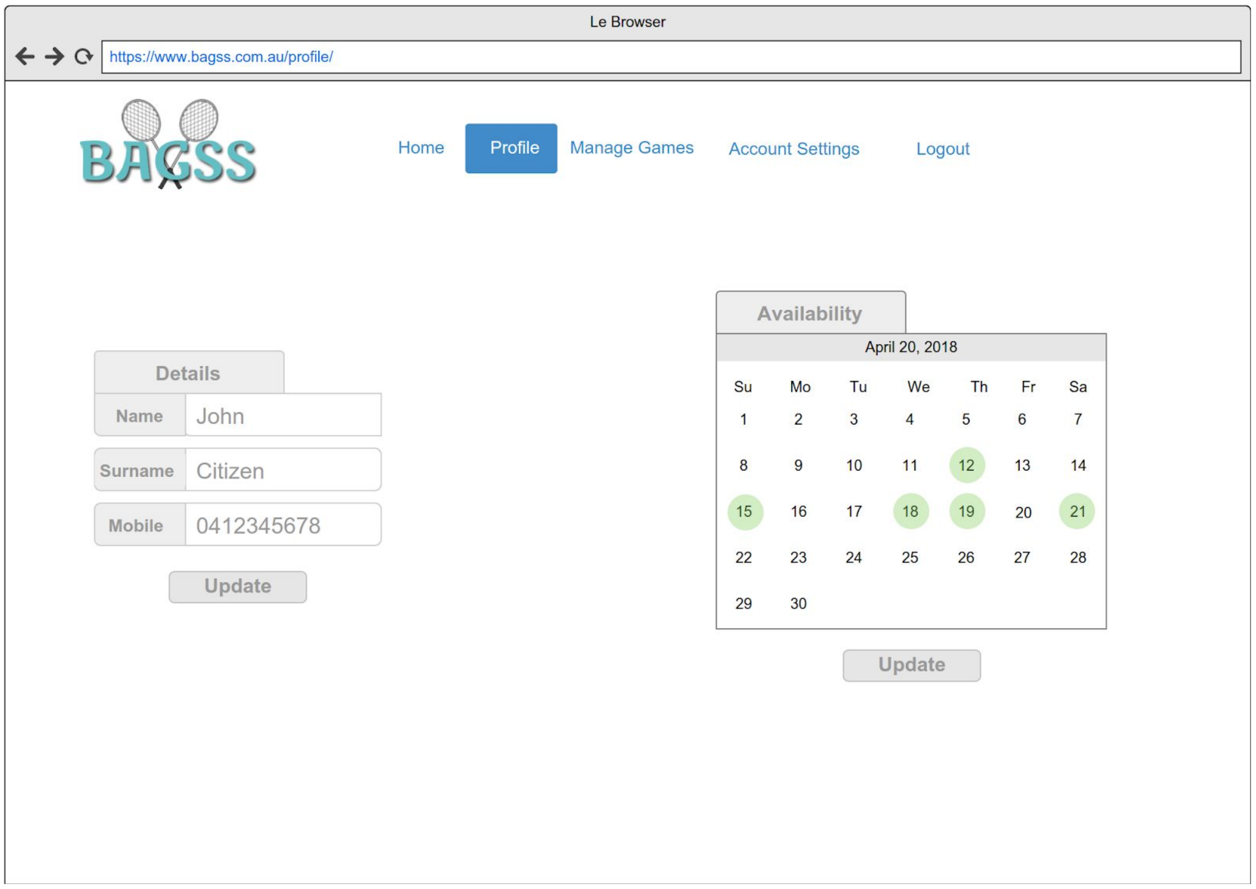


BAGSS Leaderboard

Last updated: 1:23pm, 18th Apr 2018

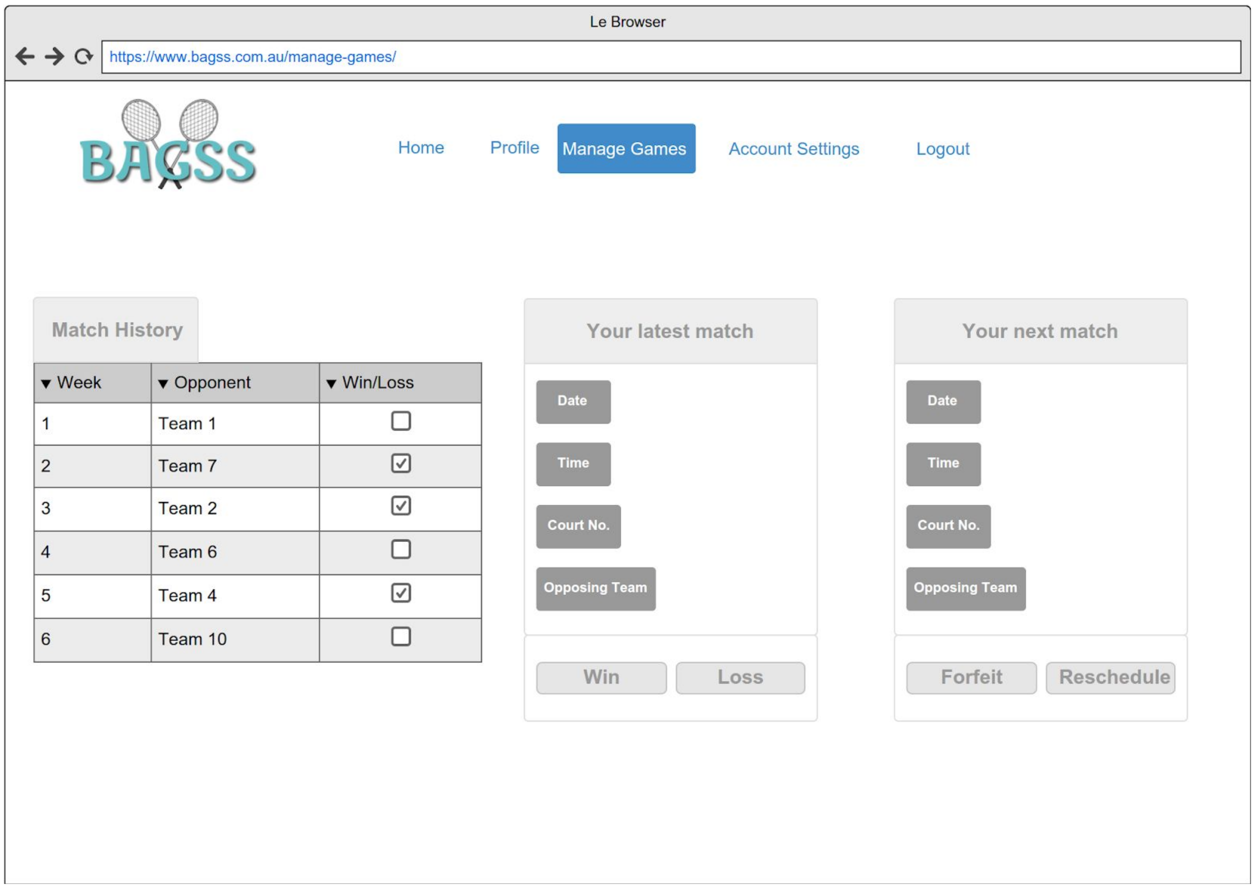
Rank		Team		Total Wins
1.	☆	👤	Team 1	7
2.	☆	👤	Team 2	5
3.	☆	👤	Team 3	4
4.	☆	👤	Team 4	2
5.	☆	👤	Team 5	1
6.	☆	👤	Team 6	0

11.3. Appendix C - Web interface - Profile Page



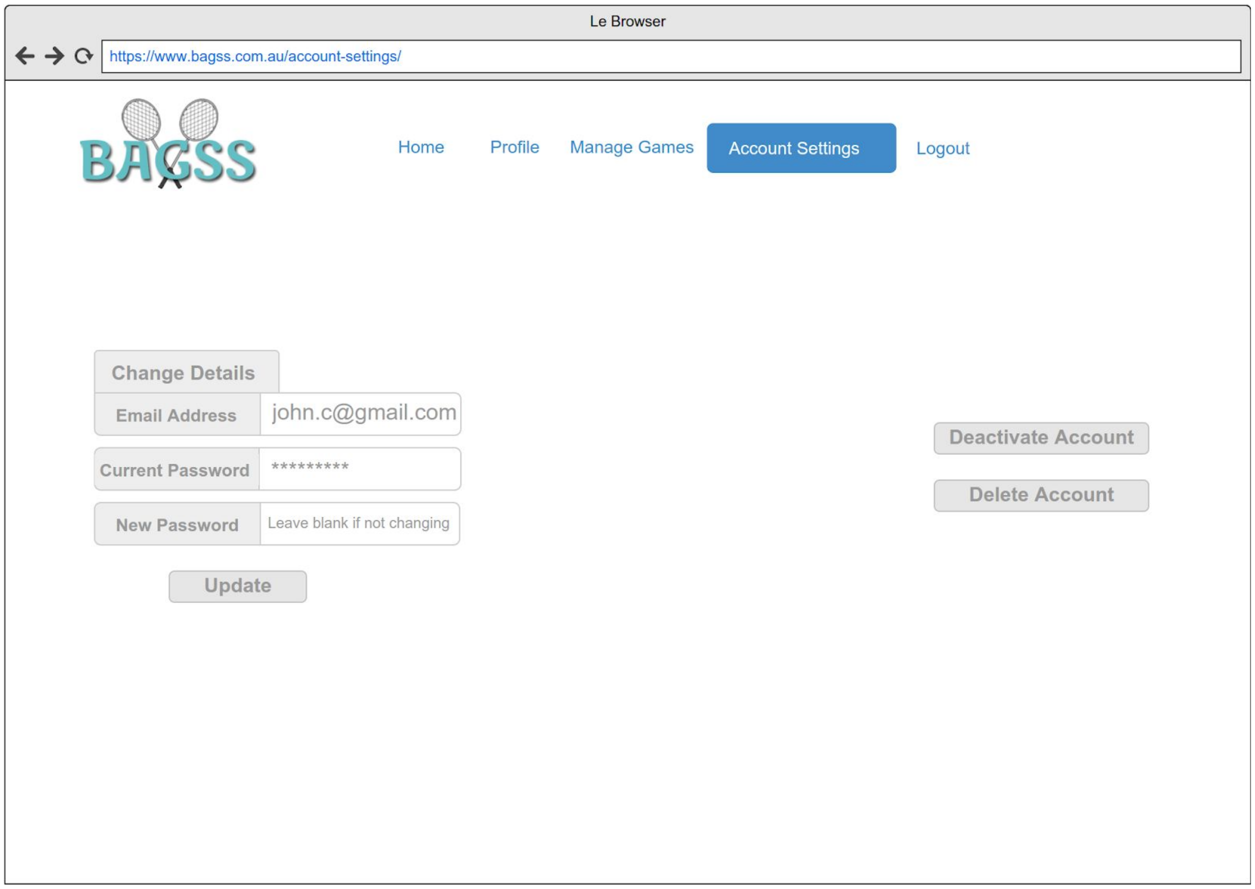
Su	Mo	Tu	We	Th	Fr	Sa
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

11.4. Appendix D - Web interface - Manage Games Page

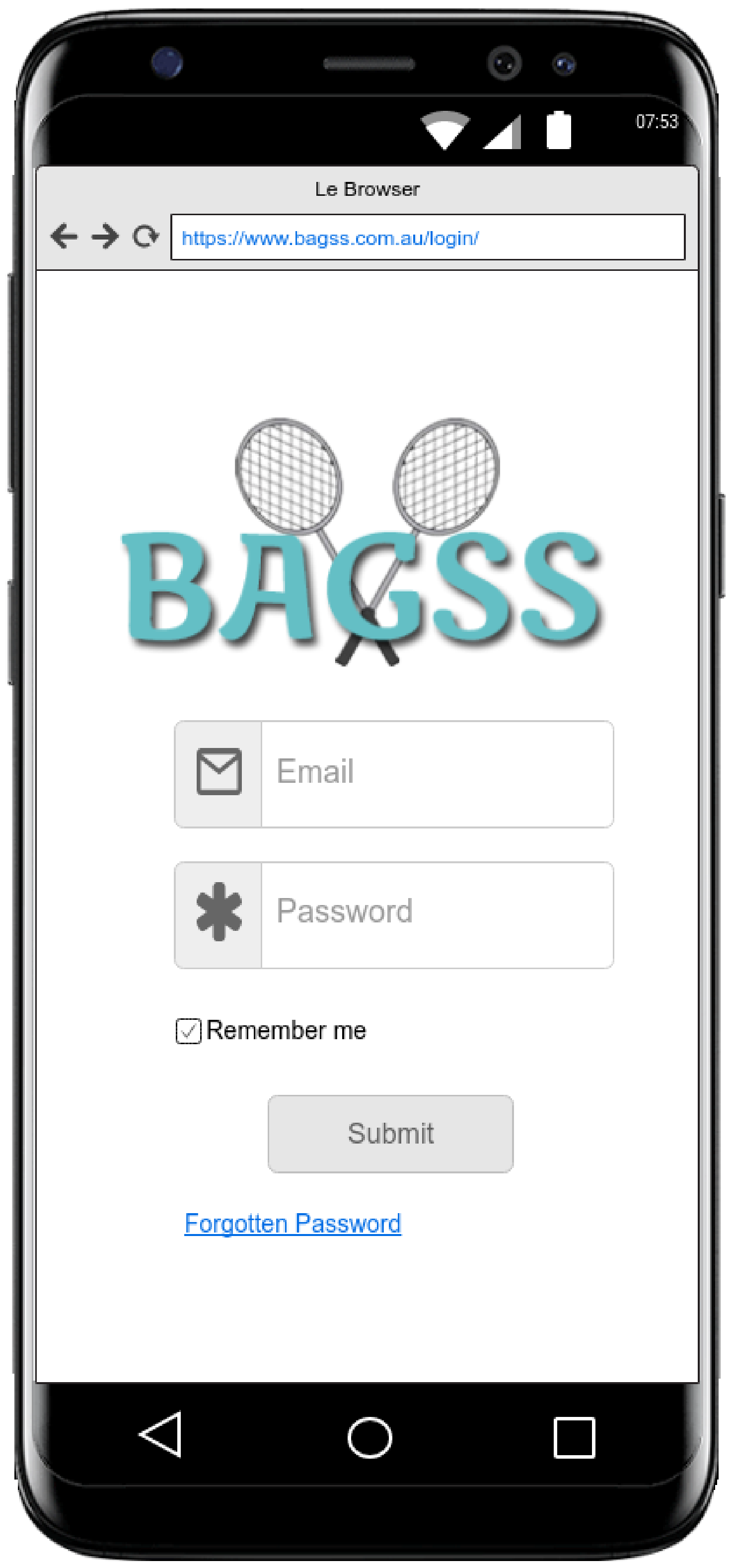


▾ Week	▾ Opponent	▾ Win/Loss
1	Team 1	☐
2	Team 7	☑
3	Team 2	☑
4	Team 6	☐
5	Team 4	☑
6	Team 10	☐

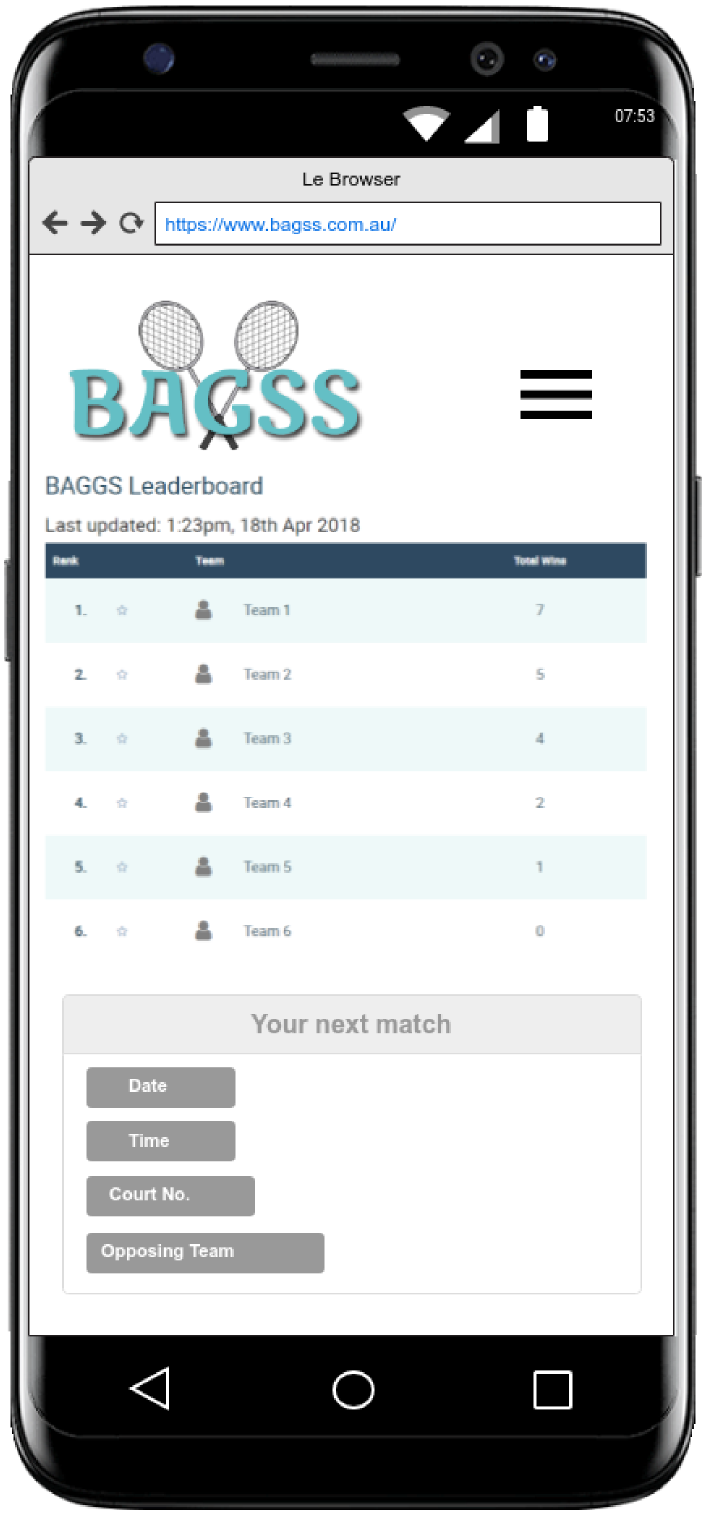
11.5. Appendix E - Web interface - Account Settings Page



11.6. Appendix F - Mobile interface - Login Page



11.7. Appendix G - Mobile interface - Homepage



BAGGS Leaderboard

Last updated: 1:23pm, 18th Apr 2018

Rank		Team		Total Wins
1.	☆	👤	Team 1	7
2.	☆	👤	Team 2	5
3.	☆	👤	Team 3	4
4.	☆	👤	Team 4	2
5.	☆	👤	Team 5	1
6.	☆	👤	Team 6	0

Your next match

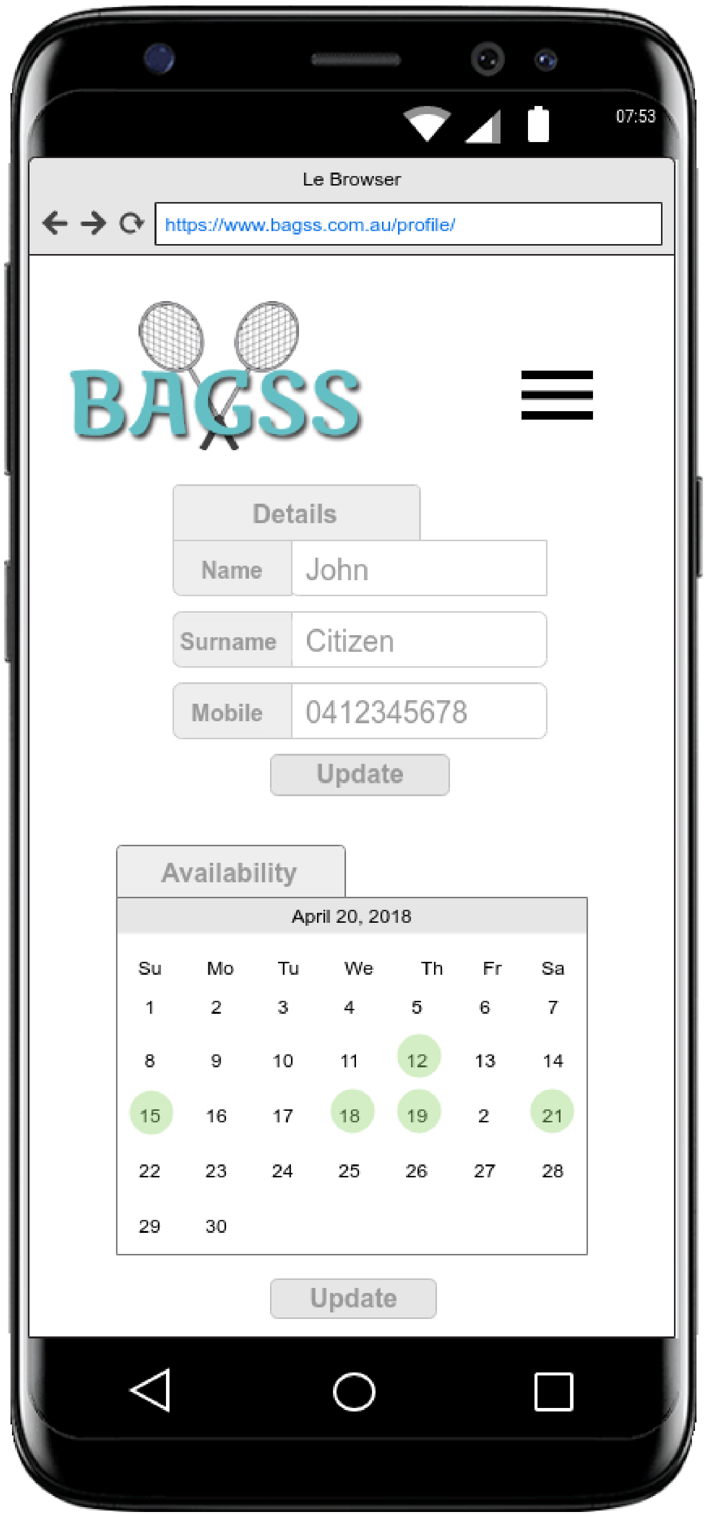
Date

Time

Court No.

Opposing Team

11.8. Appendix H - Mobile interface - Profile Page

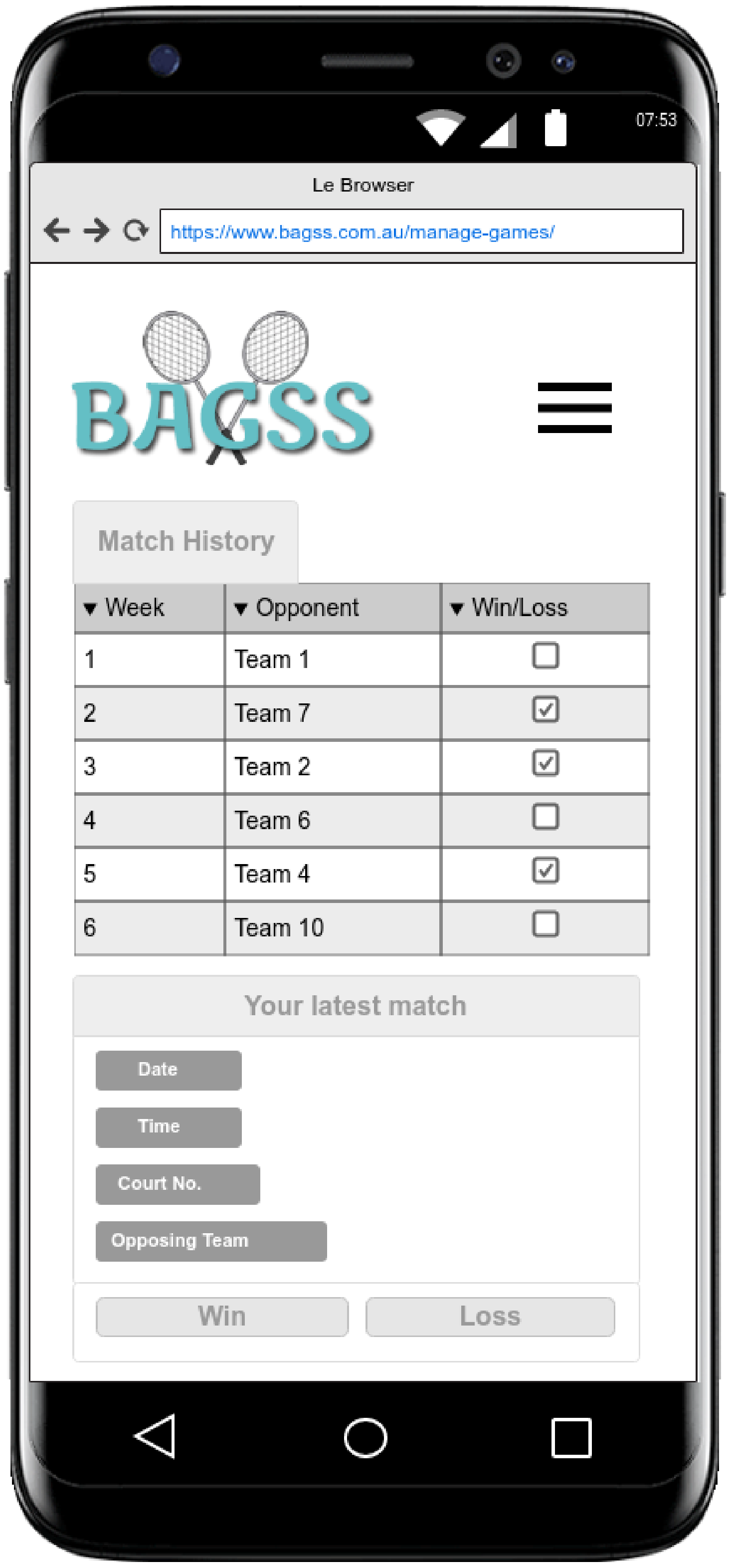


Name	John
Surname	Citizen
Mobile	0412345678

April 20, 2018

Su	Mo	Tu	We	Th	Fr	Sa
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	2	21
22	23	24	25	26	27	28
29	30					

11.9. Appendix I - Mobile interface - Manage Games Page pt.1



07:53

Le Browser

https://www.bagss.com.au/manage-games/

BAGSS

Match History

▾ Week	▾ Opponent	▾ Win/Loss
1	Team 1	☐
2	Team 7	☑
3	Team 2	☑
4	Team 6	☐
5	Team 4	☑
6	Team 10	☐

Your latest match

Date

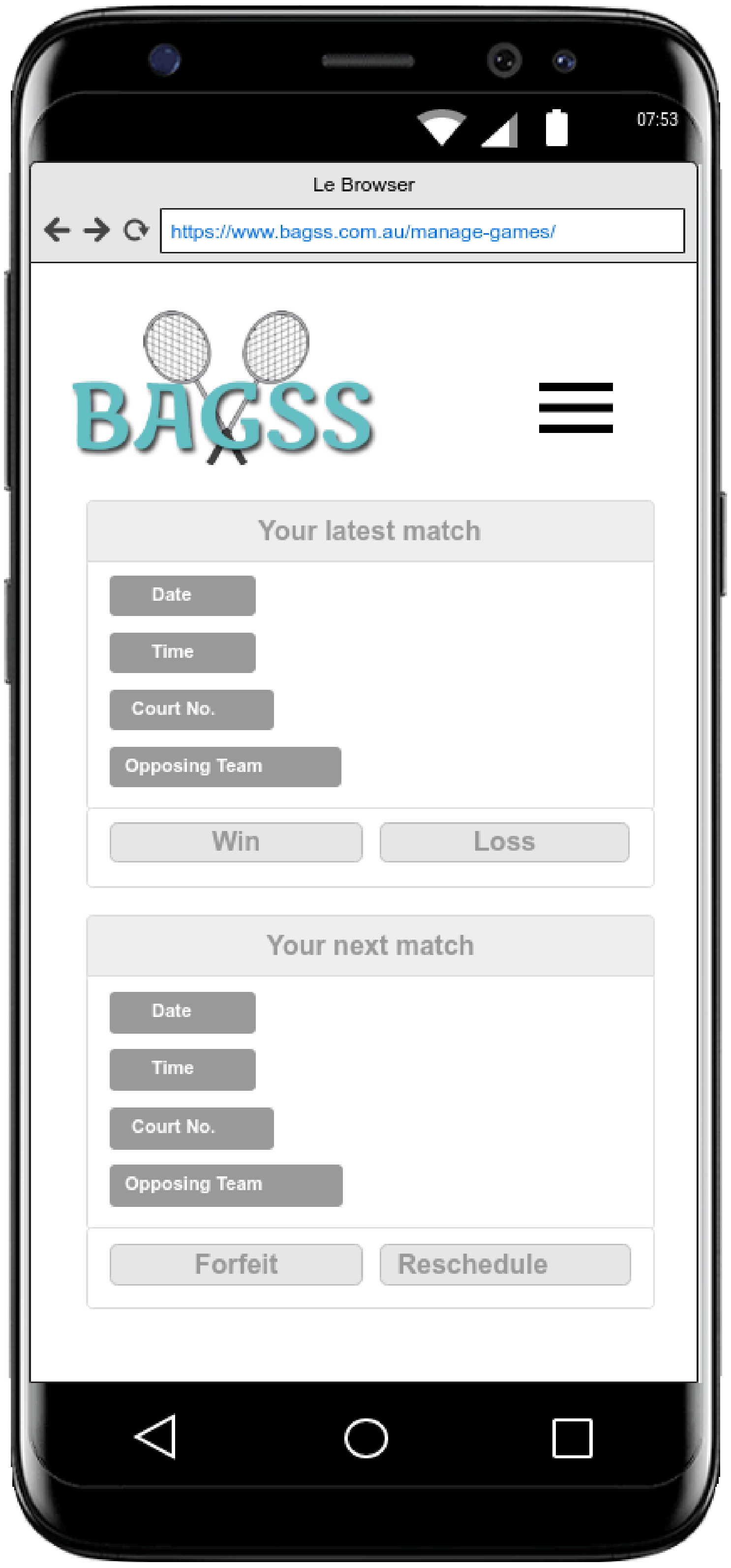
Time

Court No.

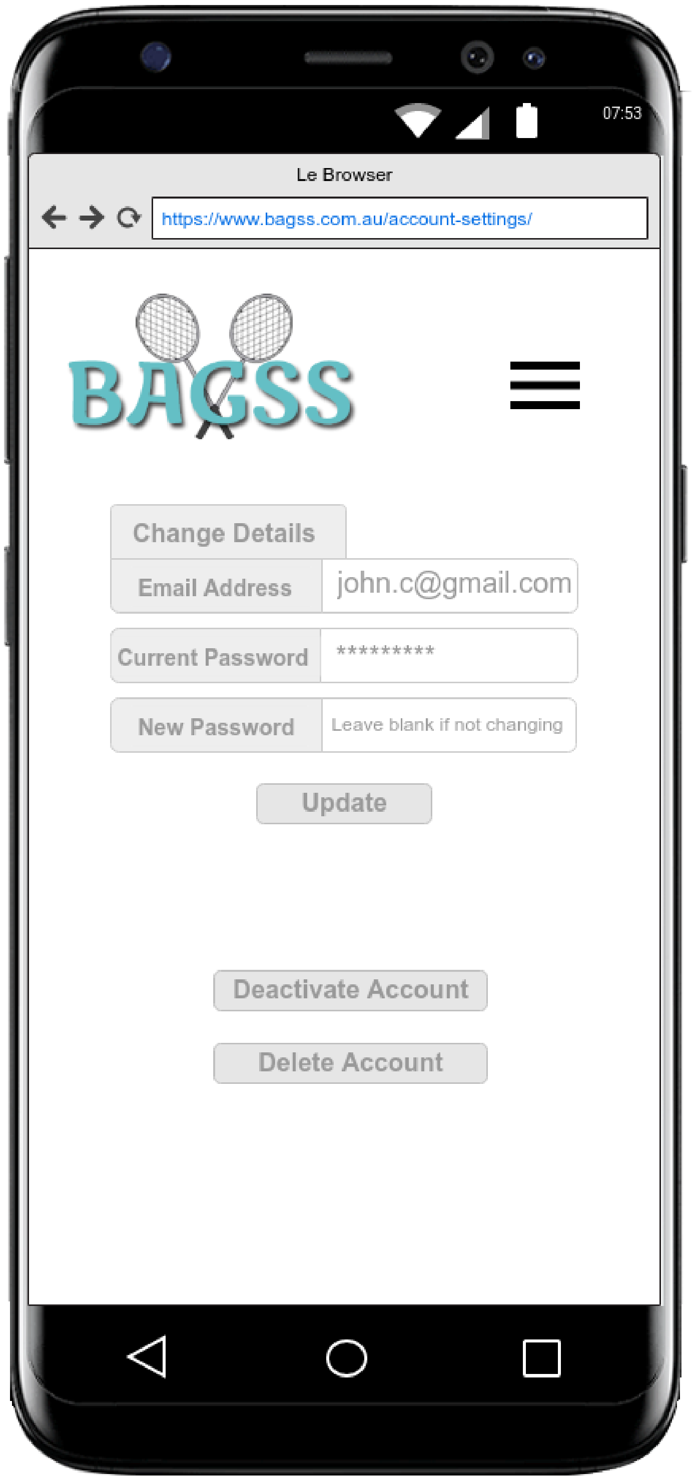
Opposing Team

Win Loss

11.10. Appendix I - Mobile interface - Manage Games Page pt.2



11.11. Appendix J - Mobile interface - Account Settings Page



Change Details

Email Address	john.c@gmail.com
Current Password	*********
New Password	Leave blank if not changing