**EE4023 TicTacToe**

**JCT2**

Niall Dillane 13132911

Joseph Tobin 16163842

Adam O’Mahony 16187504

# 

# 

[**Abstract**](#_fpats2u2wc5n) **3**

[**Introduction**](#_q25490h1jti6) **3**

[Game rules](#_m5taw4mli16) 4

[**Method**](#_trun7c43f2os) **4**

[Use Cases](#_lj69r7f8lyte) 4

[fig. 1](#_9drk2ob1r7wc) 4

[fig. 2](#_tamh25um06x0) 5

[fig. 3](#_fa9gw3p9aixk) 5

[Architecture](#_ckf81s1js4mh) 6

[fig. 4](#_2trdtalx38oz) 6

[Design](#_vmco79avb3nn) 7

[fig. 5 fig. 6](#_wam7jln3532f) 7

[fig. 7](#_ixnwrkufwhqb) 8

[fig. 9](#_wo5hqlefucuw) 9

[fig. 11](#_cs0huid9lbjc) 9

[fig. 12](#_qc0fkvbd6dl8) 10

[Threading and Synchronisation](#_qz0c75u2el8w) 11

[fig. 14](#_7rsh1k3m7hvz) 11

[Reflections and Going Forward](#_sad7wunix681) **13**

[Appendix](#_mfczq5y1o29n) **13**

# 

# Abstract

Given a predefined WebService, develop a multiplayer TicTacToe game with two clients attached - a web based PHP client and a Java client. A user should be able to register and log into the system. Once logged in, a user should be able to see any open game lobbies that are available or create a new game. A scoreboard of each client registered in the system should also be visible and frequently updated which will show each user’s win to loss ratio.

# 

# 

# 

# 

# Introduction

The use of a WebService is a common design used to hide or protect data from clients by adding an extra layer of abstraction between client’s and the data persistence layer, in our case a database. The WebService also provides a common interface through which a client on any platform can access the data stored in the database. This interface allows us to write clients in different languages, specifically Java and PHP, which are capable of communicating with each other. One of the biggest difficulties in the project is synchronising the two clients. While they communicate through the database, via the WebService, how can develop clients which are dynamic? How will we keep pages up to date with the information in the database and consistent across all connected clients? How can we design the clients to allow for easy comprehension of the codebase and maintenance in the future.

Starting with the overall system architecture and design of the system, we have opted to follow a version of the Model View Controller (MVC) design. MVC is commonly used in software development, especially in web development, and allows for fast implementation and more efficient maintenance. The design pattern allows for multiple views reducing the overall page count and saving on network activity. In order to keep clients on different platforms synchronised in this asynchronous application, we will use Session variables, timeouts and updater threads which will poll the WebService and keep the database up to date.

For the purpose of this report, the word *client* refers to the application while *user* refers to someone that is using the client.

## Game rules

The game will follow the rules of a normal game of Tic Tac Toe. Beside the base game rules, there are rules regarding a client interacting with the game and another client. First of all, if nobody joins a newly created game within a given time (e.g. 10 seconds) the game should be deleted without assigning a win, loss or draw to the creator’s match history. Secondly, if a user leaves in the middle of a game, the game should be terminated and a loss be assigned to the user that left.

# 

# 

# 

# Method

## 

## Use Cases

Although, the application is relatively small, we have identified a number of use cases that can be seen in *fig. 1* below.

##### fig. 1

These are some of the most important pieces of functionality that must be implemented by TicTacToe clients. Both the Login and Register use cases are closely related as can be seen in *fig. 2*. When a user opens the application they are prompted to either login with their account or register a new account. Regardless of the choice the user makes, they are then brought forward into the application where other use cases can be accessed.

##### 

##### fig. 2

The Play Game use case follows the sequence described in *fig. 3*. This use case is where we will see a large majority of the game logic and much of the synchronisation between clients through the database.

##### fig. 3

This diagram depicts the Create Game extension on the Play Game use case from *fig. 1*. We can see that once the game has been created and added to the database, the client begins looping, constantly updating it’s local board state. This allows a client to check if it is their turn and whether or not the game has reached a finished state, win, draw or loss.

## Architecture

Model-View-Controller (MVC) is a common architecture used in software development as a way to decouple parts of an application fulfilling different requirements. It is especially prevalent in web-based applications like this project. The file structure shown in *fig. 4* is taken from our PHP client and demonstrates the MVC design.

The Model is the information behind an application, in this application the Model is comprised of the webservice and the database. Together they model information for use and describe what functions are possible.

The View components are all of the base PHP files seen at the bottom of the file structure in *fig. 4.* These files are what the user sees on their screen. Information is changed and updated but still presented on these static pages.

The controller is comprised of all the php files in the ‘actions’ folder and the ajax files in the ‘js’ folder. These files are how information is passed from the View component to the Model and visa versa.

##### fig. 4

This ‘js’ folder contains Ajax handlers specific to different view pages. These handlers then call the necessary php files in the ‘actions’ folder, most of which map to functions defined in the webservice. The way in which we have designed the system allows us to keep the number of static php files (views) low, needing only the index, home and game pages, and to update the information displayed to the user through asynchronous calls by the controller component.

## Design

The design of the two clients is based around a number of dynamic pages which update the information that is displayed to the user as it changes in the database.

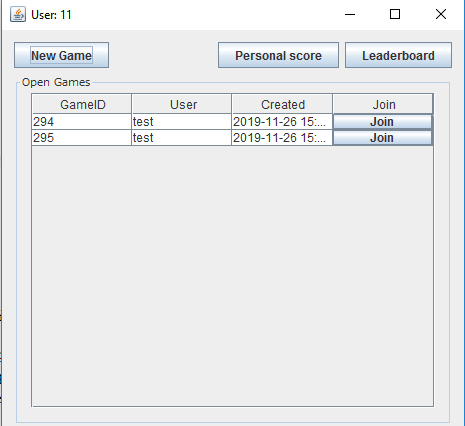
The first of these pages is the login and register page. In our PHP client, both register and login is displayed on the *index.php* page, with the necessary input form being selected by choosing the correct tab. *Fig.5* shows the index page used in our PHP client. This is the entry point of the system and contains the login and register tabs. The Java client, in contrast, offers the same two functions but changes view depending on the action the user wishes to take. The Java client’s login view can be viewed in *fig. 6.*

##### 

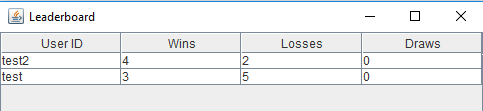
##### fig. 5 fig. 6

Once logged in, the user is brought to a home page where they can access open games, leaderboards and match history. A new game is also created from this home page. Again, the PHP client will update the *home.php* page depending on the tab that is selected by the user, while the Java client will open a new view. *Fig. 7* to *fig. X* show the different possible views that are shown to the user, including the open games, leaderboard and match history.

##### fig. 7

 *fig. 8*

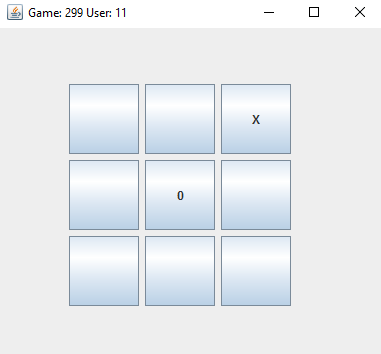
##### fig. 9

 *fig. 10*

##### fig. 11

Finally the game page is displayed when a user creates a new game or joins a game displayed in the open games list. In both PHP and Java clients, a new view is displayed for the duration of a game. On this page the user interacts with a 9 x 9 grid to play the game, with any action a user makes locally being reflected in the opponent's client. One notable difference between the two clients is that in Java, a client always displays X for the user of that client regardless if they take the first or second turn, whereas the PHP client displays X for the user that makes the first move and O for the other. While this difference in representation exists between the clients, it does not affect their ability to play against each other. The game pages of the two clients can be seen below in *fig. 13* and *fig.14*.

##### fig. 12

 *fig. 13*

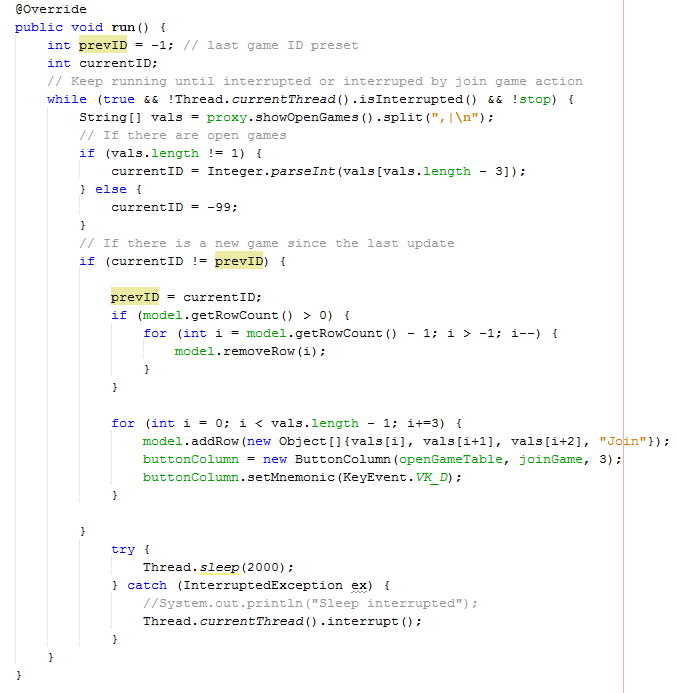
## 

## Threading and Synchronisation

The way in which synchronisation is handled between clients differs slightly between the PHP and Java clients. Unlike Java, PHP has no concept of a Thread, instead a after a function call, a timeout is set and the function is called again. In order to continuously update the leaderboard we used the code shown in fig. 15. Initially a timeout is set on the function *getLeaderboards()* on line 261. Upon completion of the function, another timeout is set on the function. This looping calls to the function will continuously update the view with information obtained from the database through the WebService. This code structure is used multiple times throughout the source for the PHP client for updating views, such as open games and the game a user is currently playing.

##### fig. 14

Java, in contrast to our web app using ajax, uses synchronous execution. Because of this, the use of threads to update user interfaces is necessary. We opted to use Runnable classes in order to both make a call to the WebService and update the UI with the result received. The implementation of the open games runnable class can be seen in *fig. 16*. A reference to the view item that is being updated, in this case the open games table, and a reference to WebService proxy are passed as arguments when creating the runnable object. Passing the table object allow us to update what is being displayed to the user without blocking the UI thread, meaning that the user is not stopped from interacting with the application when the UI is being updated. The proxy is passed which allows connection from the newly created updater thread to the WebService.



*fig. 15*

# Reflections and Going Forward

The project was split into three sections:

* Java development - Joseph
* PHP development - Niall
* Report & Tech Lead - Adam

While we split the project in this manner, all members had input and made contributions to each part of the project.

Reflecting on this project, there are a few things that we as a group would have done differently. Splitting the work in a different manner so that each person would have more influence in the each of the different sections would be the first change. Changing how the workload was divided could lead to a better developed application with each person applying their knowledge across the entire project.

Another problem we encountered was dealing with edge cases. If we were to start again, there would be a greater effort put into upfront planning and exploring what edge cases might exist. Along the same line of thinking, throughout development, the lack of a test suite was missed, so redoing the project following a more test driven development mindset would benefit the overall quality of the application.

Going forward with the application as it was developed, we would like to add additional functionality. Specifically the ability to have multiple games open versus different opponents at once. The idea of a tournament between users would also be an interesting idea to look into going forward.

# Appendix

Fig. 1 - Use Case Diagram

Fig. 2 - Login/ Register Activity Diagram

Fig. 3 - Sequence Diagram of the Play Game use case

Fig. 4 - File structure demonstrating MVC

Fig. 5 - PHP login view

Fig. 6 - Java login view

Fig. 7 - PHP open games view

Fig. 8 - Java open games view

Fig. 9 - PHP leaderboards view

Fig. 10 - Java leaderboards view

Fig. 11 - PHP match history view

Fig. 12 - PHP game view

Fig. 13 - Java game view

Fig. 14 - PHP synchronisation code

Fig. 15 - Java updater runnable code