Michael Fynes

Using An Android Application For Tracking Working Hours

Helsinki Metropolia University of Applied Sciences
Bachelor of Engineering
Media Engineering
Thesis
14 April 2015
The purpose of this final year project was to assess the feasibility of using a mobile application for monitoring working hours and to create a proof of concept mobile application to allow this. The application works together with a back end system and a web application. These three components were thus created for this final year project.

To be able to carry out the assessment a comparison was needed. Thus benchmarking was carried out on two mobile systems, one aimed at employees and the other aimed at employers. An analysis was also done on two RFID systems, one requiring manual input and the other using automatic tag detection.

The project involved three types of development, Web design, Java back end systems, and Android development. The web element was controlled by a servlet and built using HTML and JavaScript. The Java back end system was built using JavaEE, and it uses JavaBeans and servlets. The Android client uses a background service to control network interactions with an Activity-Fragment front end UI.

The assessment found that while using a mobile application does bring some benefits if a business is large enough and has a requirement for other features, such as access control, then an RFID system may still be a more effective solution. However for small or medium businesses where implementing an RFID solution may be too expensive a mobile solution coupled with a “bring your own device” policy can be a more effective solution.
Contents

List of Abbreviations

1 Introduction 1

2 Benchmarking 3

2.1 Mobile Applications 3
    2.1.1 Timesheet 3
    2.1.2 Mobile Attendance 6

2.2 RFID Applications 7
    2.2.1 Manual Input System 8
    2.2.2 Automatic Input System 9

3 Application Design 11

3.1 Core Principles 11

3.2 User Types 11

3.3 User Stories 11

3.4 Security and Privacy 12

3.5 Structure of Applications 13

3.6 Technology Used 15
    3.6.1 Wi-Fi Detection 15
    3.6.2 LE Bluetooth Markers 15
    3.6.3 Android 16
    3.6.4 JavaEE vs PHP 19
    3.6.5 Java EE 20

4 UI Design 22

4.1 Design Plan 22

4.2 Android Design 22

4.3 JavaEE Design 27
### List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HR</strong></td>
<td>Human Resources department.</td>
</tr>
<tr>
<td><strong>RFID</strong></td>
<td>Radio Frequency Identification. A set of standards used for a variety of purposes, usually characterised by the combination of a tag and a reader.</td>
</tr>
<tr>
<td><strong>API</strong></td>
<td>Application Programming Interface, which allows the use of a library of pre-defined functions to simplify and standardise otherwise complex tasks.</td>
</tr>
<tr>
<td><strong>JavaEE</strong></td>
<td>Java Enterprise Edition. A set of tools used for web applications and back end services.</td>
</tr>
<tr>
<td><strong>JPA</strong></td>
<td>Java Persistence API. A Java API that allows simple interactions with SQL databases using entity classes.</td>
</tr>
<tr>
<td><strong>CCTV</strong></td>
<td>Closed Circuit Television.</td>
</tr>
<tr>
<td><strong>UI</strong></td>
<td>User Interface.</td>
</tr>
<tr>
<td><strong>UX</strong></td>
<td>User Experience.</td>
</tr>
<tr>
<td><strong>MVC</strong></td>
<td>Model View Controller. A programming model for Object Oriented Programming.</td>
</tr>
<tr>
<td><strong>POJO</strong></td>
<td>Plain Old Java Object. An object in any Java application that contains only Java.</td>
</tr>
<tr>
<td><strong>Wi-Fi</strong></td>
<td>Wireless Network.</td>
</tr>
<tr>
<td><strong>QR</strong></td>
<td>Quick Response. This term is usually used when referring to QR codes, which are printable codes that can be read by a variety of devices (such as cameras on mobile devices).</td>
</tr>
<tr>
<td><strong>LF</strong></td>
<td>Low Frequency RFID tags.</td>
</tr>
</tbody>
</table>
HF  High Frequency RFID tags.
1 Introduction

Working time monitoring systems come in a wide range of formats, from simple paper based formats to complex automatic systems. The aim of this final year project was to create a tool that can be used by small to medium sized companies and even larger companies that cannot implement an RFID solution. RFID was chosen as a benchmark as it is a very common method of tracking working hours. The application developed for this project allows the use of a wireless network or a low energy Bluetooth marker to detect if an employee is at work or not and the interaction is thus automatic. Traditional RFID systems rely on manual user input. This approach ensures the user is aware of the monitoring system and also allows different types of interactions to be recorded.

The focus of this final year project was to create a system that can be used to monitor employees' working hours with minimal user input. Thus certain other factors have been implemented only in a rudimentary fashion, such as security features. This is due to the complexity of security systems and the likelihood of potential client companies having their own security and login systems that would need to be integrated into the application. In cases where the client company does not have their own authentication system an Android for Work login system could be used. This implements a login system using the Google Authentication API, but allows the profiles to be managed by companies rather than individual users. As the Android for Work is still in development at the time of writing this thesis, it was not implemented in this final year project. [1.]

To compare the implementation to other attendance tracking implementations, benchmarking was carried out on two mobile systems, one aimed at employees and the other aimed at employers. An analysis was also done on two RFID systems, one requiring manual input and the other using automatic tag detection. The application will use this benchmarking to implement a series of features that will help potential client companies monitor their employees' work attendance. Keeping an accurate record of these is required in Finland under the Working Hours act 1996. [2.]

Employers must register the hours worked and the relevant remunerations for each employee. The regular, additional, overtime, emergency and Sunday working hours and the relevant remunerations, or all hours worked and overtime, emergency and Sunday hours separately and the increases paid for them must be entered in the register. [2.]
According to the Finnish Occupational Health and Safety Administration (Työsuojeluhallinto) in a scenario where an employer has been negligent in the maintenance of the working time register, the Finnish courts will favour any records or information recorded by the employee. Thus it is of critical importance, from a purely legal standpoint, to ensure a good record is kept of employees’ working hours. [3.]

The application created in this project differs in some respects from other employee monitoring systems in that it does not collect contextual information, only timestamps of the beginning and end of the working period. This addresses some of the concerns raised by employees in a study by Holland, P., Copper, B. and Hecker, R. (2012) in Australia which found that many employees were uncomfortable with the direct monitoring of their activities while at work. [4; 5.]
2 Benchmarking

To ensure the mobile application developed for this final year project addresses problems found in other applications, benchmarking was carried out with four different implementations using a variety of methods to determine the positive and negative aspects of a variety of implementations.

2.1 Mobile Applications

To benchmark the mobile applications available two very different applications were chosen. The first, Timesheet, was chosen as it contains a wide variety of features but is designed for use by an individual user, the second, named Mobile Attendance, as it is specifically designed for use by companies to track employees working time.

2.1.1 Timesheet

Timesheet is an Android application that allows individuals to track the time spent on tasks or projects. The time can be recorded in real time or added later as a block. The implementation allows the creation of projects, which the user can then add items to. These items are very customisable using tags (which can be predefined by the user), a free text input field, the location of the work done and whether or not it was paid work. Thus breaks can be defined as unpaid. [6.]

The user interface is simple and easy to understand with the main page showing the project that is currently in use as can be seen below in figure 1. It also features a drop down menu allowing navigation between features from the toolbar. It is also possible to navigate between these features by swiping right from the main screen. [6.]
The projects can be further investigated to show total time spent, the total cost of the project and a description of the project. Tasks can then be added to the project either manually through the project screen or from the first screen seen upon opening the application. The tasks can be further modified to include breaks, expenses and other notes. An example of a task created can be seen below in figure 2.

**Figure 1.** Screenshot of the main screen of Timesheet application. [6.]

**Figure 2.** Screenshot showing project screen of Timesheet application. [6.]
The currency is customisable with no validation, while this may seem like a weakness it is a very clever implementation as there is currently approximately 167 official currencies in the world and thus implementing them all would be difficult in the extreme. [7.]

<table>
<thead>
<tr>
<th>STRENGTHS</th>
<th>WEAKNESSES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project tracking</td>
<td>Lots of input required</td>
</tr>
<tr>
<td>Export function</td>
<td>Odd hierarchy</td>
</tr>
<tr>
<td>Custom currency</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPPORTUNITIES</th>
<th>THREATS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic tracking</td>
<td>Meeting needs of users</td>
</tr>
<tr>
<td>Expansion into other methods of tracking</td>
<td>Overcomplicating application</td>
</tr>
</tbody>
</table>

Figure 3. SWOT Analysis of Timesheet application

The SWOT analysis shown in figure 3 is mostly positive but as with all applications there is a risk of not pleasing all of the users all of the time as well as becoming too complicated for new users.

Conclusion

This application is a very polished implementation of a tracking system for an individual user and as such would be ideal for a freelancer. However for a large organisation a modified version would need to be used, as well as a back end system to collect all the tasks and projects, though this could be done manually through the export function.
2.1.2 Mobile Attendance

Mobile Attendance is a mobile application that uses QR codes to track employees’ attendance. The application combines QR codes and a face capture image to capture check-in and check out events. [8.]

The UI is kept simple to ensure the users can access the functions they need rapidly, this is shown from the screenshot of the application shown in figure 4.

Figure 4. Screenshot of check-in or out options in Mobile Attendance. [8.]

The navigation options are clear, and they provide a powerful tool for tracking attendance using Google sheets. This integration allows for an easy method of exporting the data into a form that can be used to generate the information needed to complete the payroll. [8.]
Figure 5. SWOT analysis of the Mobile Attendance application.

Real time attendance tracking is a real strength of this application as is the integration into Google sheets. However the use of a fixed point QR code scanner is a risk if the device malfunctions as the employee time can no longer be tracked. An alternative solution could be to have the QR codes fixed in place at the workplace with the employees using their own devices to check-in. However this then provides other challenges such as cross-platforming and login systems.

2.2 RFID Applications

The two implementations chosen were chosen to represent two different types of RFID attendance tracking methods. The first is a manual input system with a tag and a reader requiring the user to hold a tag across the reader to trigger an event. The second is an automatic tracking system requiring only the user to have a tag that is present and in a readable position.
2.2.1 Manual Input System

This system was built for a project described in the article named RFID Based Attendance System. The attendance system described in the article is designed to be used by educational institutions and is an RFID based attendance system where users are issued an ID tag which can be held against a reader to trigger a check-in or check out event. The tag is built using low frequency (LF) RFID standards and as such operates at a 125 kHz frequency. The reader is a passive reader operating on AC power at 12V and is capable of switching to battery power automatically if the main power supply is disconnected. [9.]

The user interface in RFID applications is twofold. The end user has the experience when using the tag and the reader, and the employees that handle the data then have the second part of the experience. The data handling is not described in great detail other than the fact that it is recorded. [9.]

<table>
<thead>
<tr>
<th>STRENGTHS</th>
<th></th>
<th>WEAKNESSES</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy</td>
<td></td>
<td>Manual input</td>
<td></td>
</tr>
<tr>
<td>Cheap</td>
<td></td>
<td>Limited back end system</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPPORTUNITIES</th>
<th></th>
<th>THREATS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Variety of input</td>
<td></td>
<td>Data handling</td>
<td></td>
</tr>
<tr>
<td>Easily transferable</td>
<td></td>
<td>Erroneous Input</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6. SWOT analysis of the manual input RFID system.

This RFID solution uses the tag and reader method to track working hours. Each tag contains an RFID chip that is activated when it is held within the range of the read-
er. LF tags are read using inductive coupling. This method allows the transfer of energy from the reader to the tag from antenna to antenna. As the magnetic field induces a current in the passive tag the tags antenna changes the magnetic field. This change is detected by the reader which is how the data is transmitted. Using inductive coupling means the tag and reader must be quite close to each other for a successful read. In some implementations this may be considered a disadvantage, but in attendance tracking this is in fact a strength as it ensures that accidental readings are kept to a minimum. It also increases the lifespan of the tags as the maintenance required on each tag is minimal if the tag itself can be protected adequately. [9.]

However any system that relies on manual input will suffer from certain drawbacks, the main challenge is simply to ensure the users actually use the system. To overcome this a system or process must be implemented to handle situations that arise where something has gone wrong. The implemented method works well and addresses concerns about privacy by forcing the user to manually swipe the tag against the reader. This allows the organisation running the setup to assume the intent of the user is to have their attendance checked.

2.2.2 Automatic Input System

The Automatic input system was built for the article named The Research and Application of College Student Attendance System based on RFID Technology. The system uses an ultra-high frequency (UHF) tag which is issued to each student. The tags are then read automatically by a reader positioned at the entrance to each classroom. This removes the need for any manual input. [10.]

The users input to the system is recorded automatically. The display on the reader then updates, showing the classroom number, time, date and the number of students present. [10.]
**SWOT ANALYSIS – RFID Automatic input**

<table>
<thead>
<tr>
<th>STRENGTHS</th>
<th>WEAKNESSES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic</td>
<td>Expensive</td>
</tr>
<tr>
<td>No input</td>
<td>Tags must be visible to reader</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPPORTUNITIES</th>
<th>THREATS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adaptable to different scenarios</td>
<td>Privacy concerns</td>
</tr>
<tr>
<td></td>
<td>Erroneous input</td>
</tr>
</tbody>
</table>

Figure 7. SWOT analysis of an RFID system with automatic input.

The method could work well in an environment where the flow of users could physically be controlled. If this is not possible then reading the tags could become problematic. The user experience may also leave something to be desired if the user is unable to determine if they have successfully been checked in or out. In the mobile and RFID solutions that have been described so far in this document this is shown in some form of interface, but if the only indication given is the total number of students this may not reassure the users enough.
3 Application Design

3.1 Core Principles

All three components of the application, the Android client, the Web client and the backend system are designed using the same principles, which are

- All functionality must resolve a problem or need of the users.
- All functionality must be reliable.
- All UI elements must be consistent throughout the application.
- The system must be scalable.

These four principles are to be met by all aspects of the application.

3.2 User Types

Four different user types are defined by the system. These allow the permissions to be set according to the role the employee is performing. The office employee user type is the base user type that all others extend from. This user type can track their working hours using both the Android and web applications. The field employee user type is similar to an office employee but with fewer time restrictions and will use mainly the LE Bluetooth markers. Managers are third level employees and can perform all previous functions and also monitor users within their own team/department. Finally human resources employees can perform all previous functions and also monitor all users.

3.3 User Stories

The first user story is a simple login and is applicable to all user types. The first step is to open the application. The user then logs into the application using the credentials they have been provided. The main screen of the application is then shown.

The second user story involves the user checking in for the first time. This case is also applicable to all user types. After the user has successfully logged in the user enters
the work area, the area in this instance being defined as within range of the pre-defined Wi-Fi. The user is then automatically checked in. The success or failure of the check-in event is then shown to the user.

The third user story affects only field staff using LE Bluetooth markers. The user enables Bluetooth on their device and enters their vehicle where the Bluetooth marker is stored. Upon entering the vehicle the marker is detected by the device. The user is then automatically checked in. The success or failure of the check-in event is then shown to the user.

The fourth user story applies to all users and shows the process of checking previous working times. The user opens the application and is automatically logged in due to a previous login. The user then sees a list of dates in the main screen and selects the desired date. The time worked on this date is then displayed to the user.

The fifth user story applies to all users of the web application. The web application allows users to see more details than the mobile application. The web user logs into the web client and sees the navigation menu. The option Full Month is selected and the last thirty days working time is shown.

The final user story shows the path taken by a user that has both Bluetooth and Wi-Fi disabled on their device. The user opens the application and logs in as normal. However as both Wi-Fi and Bluetooth are disabled no automatic check-in or check out is performed. To compensate the user chooses Manual check-in and this triggers the check-in event. The success or failure of the check-in event is then shown to the user.

3.4 Security and Privacy

The application uses only rudimentary login systems, as such as little personal information is stored or accessible. The security of users is of paramount importance when developing an application. However as this is a topic that is expansive enough to cover several theses it has been side-lined in favour of focusing on the main goal of this project. However as privacy is a key concern for many users certain steps have been taken to ensure that no contextual information is passed between the front end client and the back end service. Thus no geolocation data is used and all information related to
dates is transmitted as a series of numbers to be converted to actual dates only when requested by the front end client.

3.5 Structure of Applications

The application is structured into three parts, the front end clients for Android and Web browsers and a back end application. The Android application is the key part as it handles the main check-in and check out events.

![Diagram of Android application structure]

Figure 8. Structure of the Android application.

As can be seen from the figure above the Android application itself is split into several components, key among them the Application, Activity and Service. These three components run the application logic and ensure a smooth user experience.

The back end service is split into four parts, the servlet that handles requests from Android clients, the servlet that runs the web application and its associated managed
bean (Web Bean in the diagram below), the managed bean that handles all database interactions (DataBean in diagram), and finally the database.

Figure 9. Structure of the JavaEE application.
This structure allows a range of features to be implemented for both the web application and the Android client while ensuring only actions that are allowed for each client are performed.

3.6 Technology Used

3.6.1 Wi-Fi Detection

Check-in is done when relevant Wi-Fi is connected. A check out cannot be done on Wi-Fi disconnect as the user may not have Wi-Fi access. Thus the check out time is saved on the device in the default settings in the application. Then when the user reconnects to a network the check out is completed.

3.6.2 LE Bluetooth Markers

LE Bluetooth markers used for the purpose of this application are Estimote beacons. An example of an estimate beacon is shown in figure 10 below. [11.]

Figure 10. Image of LE Bluetooth beacon from Estimote copied from Estimotes own website. [11.]

These allow a variety of settings to be configured to suit their purpose, such as the range and broadcast frequency. [11.]
3.6.3 Android

The Android application is structured in a three tiered manner using the application class as a singleton to move data between the background service and the front end UI. This structure is crucial as it allows all networking tasks to be run in the service rather than through the UI. This ensures that even if an error should occur the UI remains stable.

The application at the bottom is the class TrackerApp which extends the application class. This allows the application to store certain variables while it is active regardless of UI elements such as activities or fragments being visible.

The application has thus been built to follow the MVC model for Android shown below.

![Android MVC model](image)

Figure 11. The Android MVC model [12.]

In the Android operating system the View is defined by the layout XML file, the controller is the Activity-Application-Service group of classes and the model is the data. In this case the data is retrieved from a back end data source and then added to a list of objects.
The service is a critical part of this application. By using a background service the application can continue to interact with the back end server and other parts of the Android operating system without requiring the users input. This ensures that no manual interaction is required to monitor the working hours of the users. This solves one of the drawbacks of the benchmarked systems.

The service can be initiated from one of three ways. The first is through the application class, which can begin the service to check the login details entered. The second method is from the WifiChangeDetector class. This class is a Broadcast receiver. Broadcast receivers allow a class to receive certain items that are broadcast by the Android operating system. For example when there is a change in the network status a broadcast is sent to all applications registered to listen to that type of broadcast. The name of the broadcast is shown below.

"Android.net.conn.CONNECTIVITY_CHANGE"

This broadcast triggers the WifiChangeDetector class to be instantiated. The class then runs its code and checks first if Wi-Fi is connected, and if it is connected that it is in fact the Wi-Fi name that the user is setup to listen to. If it is not, the broadcast receiver triggers a check out event in the service.

The third method is also a BroadcastReceiver class and is called BluetoothChangeDetector, this class detects changes in the devices that the Bluetooth hardware can detect, if the LE Bluetooth beacon is there a check-in event is triggered in the service; if not, a check out event is triggered.

The check-in process shown in figure 12 below documents the logical flow from each possible beginning to the end of the process.
Figure 12. The process flow of a check-in event.
The check-in process covers several different classes throughout all layers of the application potentially beginning and ending from the UI.

![WorkTracker – Check out process](image)

**Figure 13.** The process flow of a check out event.

### 3.6.4 JavaEE vs PHP

JavaEE was chosen as the technology of this back end system due to the scalability versus PHP. By using the Java enterprise platform a system can be developed that can easily be packaged and put in place on any Tomcat application server.

Additionally the ease of SQL interactions through the JPA API makes it a strong candidate for any back end developer.
However there are certain drawbacks. For example sessions are not handled as simply as in PHP. This issue has been somewhat overcome through the use of servlets in the web application.

A major advantage of using Java over a web scripting language such as PHP is the analysis tools that are available. The Apache Tomcat server package comes with a great deal of analytics regarding for example memory, garbage collection and other services. These tools are not available with PHP without accessing the server directly or through the use of an external API.

3.6.5 Java EE

The servlet class Servlet in the Controller package is a simple servlet with a large switch on its post method. When the servlet receives an HTTP Post request it first determines the method it needs to continue with. This method is a string that is sent with all interactions between the front end and back end. The servlet can then perform a variety of functions, such as adding a user, checking a user's ID or sending back a list of working times. [13.]

The servlet is used because it can receive post requests and send a custom response in a very simple manner. This particular servlet then interacts with the managed bean named DataBean. [13.]

JavaBeans are classes containing other objects. Managed beans are JavaBeans that are managed by a framework. In this application there are two managed beans, the DataBean class which controls all interactions with the database, and the WebBean class which controls all interactions between the web application servlet and the DataBean (and thus the database). [13.]

The Java Persistence API allows a class to be an entity. This is an object that contains attributes based on the table it is linked to. In this application all entity classes are linked to a MySQL table. The class can then also contain annotation for SQL queries that can be run on the table for that class. These named queries allow a list object to be returned of the type of the entity class. For example in this application a common feature is to query the database and return a list of working hours. This returns a list object like the one shown below.
List<Worktime> queriedList =
em.createNamedQuery("Worktime.sortAscending").getResultList();

This will instantiate the variable queriedList, with whatever is returned from the query. Using lists of objects rather than standard arrays provides a useful and more readable method of iterating the lists later. [13.]

The web application works in a similar method to the Android servlet interface. However rather than returning variables to the client, the output is the client. By posting using an HTML form to itself it allows navigation through the application. Each time the page is opened a check is performed to ensure the user has logged in. If the user has not logged in the user is directed to the login screen, otherwise the user is shown the index page, this session is valid for thirty minutes, and after this the user will need to login again. [13.]
4 UI Design

4.1 Design Plan

The original design plan was to utilise the Android ActionBar functionality to create tabs which could be navigated through to the various parts of the application. However as ActionBarTabs are deprecated the application was instead designed to use a simple activity plus fragment UI. The reasoning behind the design is based on a value-added system that allows each feature to be assessed for the value it adds to an application both from a functionality and a UI perspective. This has been implemented while attempting to follow the design guidelines laid out by Google for their Android operating system. [14; 15]

4.2 Android Design

Android application user interfaces are displayed according to an XML file with each item being a view class that can output different things. In the mock-ups below each rectangle or box defines a separate view class. To keep a structure that is scalable for larger devices the root of each view is a LinearLayout. This root view contains several other linear layouts which build the structure of the UI.
Figure 14. Original design concept.

As figure 14 shows the original plan would incorporate tabs in the action bar and content would appear below the action bar.
Instead the main view is built using an activity with a fragment as shown in figure 15. This allows the action bar to be used purely for showing the name of the application and for accessing the settings menu.

Figure 15. Final mock-up of main activity.
Figure 16. Mock-up showing the content activity view.

Once a user clicks on an item another activity is started which loads a fragment which can then utilise the position of the item clicked to find the details needed.
As some differentiation between users is necessary a rudimentary login system was also required. This screen utilises a custom view called FloatingEditText created by Brian Nicholson enabling the hint for the edit text to “float” above the input text, thus removing the need for a bulky title. This feature comes as standard in devices running Android Lollipop but requires the custom view for older operating systems. [16.]
4.3 JavaEE Design

As the only user interface that is shown through the JavaEE application is a web page, a simple mock-up was constructed to reflect the simplicity of the Android UI.

![Diagram of the base mock-up of the web application.](image)

Figure 18. The base mock-up of the web application.

The web application will allow users to see full details of everything that is stored about their working hours. It will also allow users of a certain level to add new users.

The idea behind the design is to keep it simple. This UI will allow the user to interact with the back end system and find the information that has been input by the front end client.

4.4 Navigation Design

The navigation of the application is very simple. Due to the nature and purpose of the application a majority of the actions performed by the application are run in the background and thus only the results are visible to the user through the UI. The user can navigate between screens by clicking an entry in the ListView. This then will display
the details of the item. The user can then return to the main view by either clicking the back button on their device or using the back button on the action bar.

Figure 19. The navigation flow of the application

The difference in the buttons is somewhat complex, the back button in the action bar will always return the user to the activities’ hierarchical parent. However the back button on the device will return the user to whatever activity was last viewed, which may be a different application altogether.

The web application uses a similar logic, but due to the large screen size the list can show the full details of each entry. First the user is presented with a login screen as shown in figure 20.

Figure 20. The login screen of the web application.
This will send the login code to the web app servlet which will handle the UI output for the web application.

Figure 21. A screenshot of the main navigation view of the web application.

A navigation menu is displayed upon a successful login as shown in figure 21, along with the user’s full name (to ensure they are logged in correctly). Once an option is selected the request is posted again to the web app servlet and the chosen item is output. If for example the button “Last Seven Days” is clicked the list shown in figure 22 is displayed.

Figure 22. A list available from the web application.
The list structure shows the last seven check-in events of the user, with the item currently being inspected highlighted to allow easier reading.
5  Data Structure

5.1  Database Design

The back end system persists data using the Java Persistence API (JPA). This allows Java managed beans to interact with entity classes and so store data easily and securely.

The MYSQL server is run on the application servers' localhost. This adds another layer of security as the application can only be accessed by applications running on the server.

Figure 23. SQL database structure.

While there are other possible solutions to problems requiring a database such as NOSQL or other forms of SQL, for this project MYSQL was chosen due to its open source nature and the ease with which it can be tied to a JavaEE application. Additionally by implementing a MYSQL structure from the beginning, the application thus can be scaled to meet growing demand for the service.
5.2 Tables

5.2.1 Codes

This is the first table that a user accessing the application will encounter. The table is a simple table with an ID which is used as the login code. Using this table allows the back end service to provide the front end client with the information it requires to perform the check-in and check-out tasks. For testing purposes it also stores the name of the Wi-Fi the tester will interact with. In a production environment this field would allow a company with multiple locations to utilise a single back end service.

5.2.2 Permission Level

This table contains the various permission levels that a company can set and use to differentiate what lower level employees can see compared to upper management or HR. It is linked to the EMPLOYEE table.

5.2.3 Employee

This table is setup with a variety of purposes in mind. First, the table should store the relevant details of an individual employee. Second it should provide a reference to the employee’s supervisor. This is achieved using a recursive relationship through the primary key. Thirdly, it allows the permission level to be set. Alongside the WORKTIME table this is one half of the central tables that control the application.

5.2.4 Worktime

Arguably the key table in the database, the WORKTIME table stores the dates and times of the employees’ working hours. The date is not stored separately but is calculated from the start time each time the date needs to be shown. This is possible as the start time is saved as a BigInteger type. In Java the dates created are represented as the number of milliseconds since the 1st of January 1970 at
00:00:00 GMT. This allows both time and date information to be parsed from a single value. [17.]

5.2.5 Tracking Type

The tracking type table allows the users to see the method used to track the employees' working hours. In this application the three inputs possible are Wi-Fi (default), LE Bluetooth Beacon, and manually checking in.
6 Code Structure

6.1 Android

The Android application is structured into three key components, the application class, the service and the activities. However, there are twenty classes in total that are required to ensure the application functions as designed. These come in several forms, application, service, activity, fragment, asynchronous tasks, POJOs and interfaces. The TrackerApp class is a singleton class and is the first class instantiated when the application is run. Secondly, there is the service class TrackerService. This class runs any background tasks that need to be performed. The third type of class are activities. These are classes that control the views and handle any fragments used. There are four activity classes in the application, LoginActivity, MainActivity, DetailActivity and the SettingsActivity. These activities all extend the Android class ActionBarActivity. The Fragment classes control and display the actual views and handle any input from the user. Each activity has its own fragment class, Login, ManualFragment, DetailFragment and the SettingsFragment. Asynchronous task classes allow tasks to be carried out in an asynchronous thread. The information is sent to the task and the result is returned to the thread that started it. These classes allow the application to run network tasks without interfering with the main thread the application is running on. There is also a single POJO class that is used to fill lists retrieved from the back end system. This class, named Worktime, is modelled on the entity class of the same name in the JavaEE back end system. Finally, there are the interface classes that allow the different parts of the application to communicate. [18.]

To ensure consistent functionality, readability and backwards compatibility, the structure and design of the code was built using a set of best practice guidelines developed by Futurice. [19.]

6.1.1 TrackerApp

This class is a singleton and thus can only have one instance per application. This ensures the UI elements and the service are always communicating with the same in-
stance of the application. While it contains a number of variables none are instantiated by the application class. Thus when accessing from other classes the variable need to be checked to avoid null pointer exceptions. Notable methods include setLastSevenDays. This function either simply updates the list that can be viewed from the UI, or, if certain conditions are met starts the MainActivity class.

```java
public void setLastSevenDays(ArrayList<WorkTime> newList) {
    if(getLoginStatus() == true){
        this.lastSevenDays = newList;
        sender.changedStatus();
    }
    else{
        this.lastSevenDays = newList;
        if(serviceAction.contentEquals("getUserId") { 
            loadUserInterface();
        }
    }
}
```

Listing 1. Function from the class TrackerApp which can start the UI.

What the function in listing 1 checks is the current login status. Thus if the user is already logged in, the list is updated and the message that the list has been updated is sent to the MainActivity. This then triggers the list to update.

Another key method in the application class allows the service to be started. This method is called sendMethodToService. First the method is set, for example “getUserId” which is used to get the ID and Wi-Fi name associated with a login code.

```java
public void setLoginCode(String loginCode) {
    this.loginCode = loginCode;
    setServiceAction("getUserId");
    sendMethodToService();
}
```

Listing 2. The function called when a login is initiated.

Now that the method has been set, as shown in listing 2, the sendMethodToService function is called.
public void sendMethodToService()
{
    Intent intent = new Intent(this, TrackerService.class);
    startService(intent);
}

Listing 3. The sendMethodToService function. Initiates the background service.

The actual method is read by the service once it has been started. This function simply starts the service. By starting the service in this manner the application triggers the onStartCommand to be run by the service.

6.1.2 TrackerService

The methods used by the service vary depending on the method set in the application class. The method set depends on how the application was launched. For example if the application was launched by user input through the login screen the service method will be "getUserID" as seen in the previous example. However if the application is started after a broadcast receiver begins the service the method can also be check-in or check out. To avoid a complex if/else statement a switch on the string method is used. While it is usually considered bad practice to hard code a string into an application in this case the string used must be constant and so using a string assigned in a string resource file is not possible.

When the startService(intent) method is called, from the application or a broadcast receiver, the onStartCommand function is automatically called.

    public int onStartCommand(Intent intent, int flags, int startId) {
        String dataString = intent.getDataString();
        tA = (TrackerApp) getApplicationContext();
        prefs = PreferenceManager.getDefaultSharedPreferences(this);
        int userId = prefs.getInt(getString(R.string.pref_user_key), 0);
        switch(tA.getServiceAction()){
            case "checkIn":
                if(userId>0){
                    tA.setLoginId(userId);
                    checkIn();
                    }
break;
case "checkOut":
    if(userId>0){
        tA.setLoginId(userId);
        checkOut();
    }
    break;
case "getUserId":
    fetchUser();
    break;
case "updateList":
    getDetailsForList();
    break;
default:
    break;
}

Listing 4. The switch in the service controlling the output of the service.

The switch function shown in listing 4 ensures that only the relevant functions are run and instantiated by the service.

These functions run asynchronous tasks, the results of which are sent back to the service through an interface called AsyncListener.

For example if the service is started with the method set to "updateList" the getDetailsForList(); method is called.

    public void getDetailsForList(){
        SharedPreferences sharedPrefs = PreferenceManager.getDefaultSharedPreferences(this);
        String listSize = sharedPrefs.getString(getString(R.string.pref_list_key),
            getString(R.string.pref_list_default));
        String empId = Integer.toString(tA.getLoginId());
        AsyncListGrabber listGrabber = new AsyncListGrabber();
        listGrabber.SERVER = tA.getServerURL();
        listGrabber.listener = this;
        int MAX_LIST_SIZE = 25;
if(Integer.parseInt(listSize) > MAX_LIST_SIZE){
    listSize = Integer.toString(MAX_LIST_SIZE);
}
listGrabber.execute(empId, listSize);

Listing 5. An asynchronous task being initiated.

The function begins by defining the size of the list the user has requested. This is stored as a string as all HTTP Post communications are sent as strings.

The asynchronous task is then instantiated, the server destination is set. The desired destination for the result is set. Finally before execution a check is run to ensure the maximum list size is not exceeded. The employee ID and list size are then sent to the task to be executed.

This structure is used throughout all other interactions with the back end.

When the task is complete, the result returns to the service, and is then sent to the application by calling the setLastSevenDays function seen in listing 1.

```java
@override
public void sendList(ArrayList<WorkTime> result) {
    tA.setLastSevenDays(result);
}
```

Listing 6. An interface method executed through the TrackerApp class.

In this manner data can be transferred between a back end database and the UI of an Android without interfering with the UI thread.

6.1.3 MainActivity and ManualFragment

The MainActivity class runs the fragment and controls the action bar and menu that is visible while the fragment is active.

When the activity is started from an intent the onCreate method is called. This begins the fragment as well as instantiates various tools the fragment may require.
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    tA = (TrackerApp) getApplicationContext();
    setContentView(R.layout.activity_main);
    if (savedInstanceState == null) {
        getSupportFragmentManager().beginTransaction()
            .add(R.id.frag_container, new ManualFragment())
            .commit();
        prefs = PreferenceManager.getDefaultSharedPreferences(this);
    }
}

Listing 7. The onCreate method from the MainActivity class.

For example the tA variable allows the fragment to access variables and functions that are present in the application class. The variable named prefs allows the local preferences to be accessed from the fragment, as shown in listing 7.

The fragment ManualFragment then inflates the layout defined in the XML and sets up the list adapter to be used for displaying the list.

    hoursAdapter = new ArrayAdapter<String>(getActivity(),
        R.layout.list_item_time,
        R.id.list_item_time_textview,
        new ArrayList<String>());
    fillList();

Listing 8. The adapter used to create the list in the main view.

The adapter takes several parameters, first, a context, second, the layout that contains the list structure, third, the id of the item to input into the list, and finally an array of strings. As the application adds to, and clears, the list depending on the users input, or any automatic input, a blank array list is instantiated in its place. This blank list is then filled each time a change is made using the function shown in listing 9.

The list is then filled by a separate function. This is kept separate to ensure it is reusable by other parts of the fragment such as the onResume function.
public void fillList()
{
    hoursAdapter.clear();
    ListView listView = (ListView) rootView.findViewById(R.id.listView_time);
    if(((MainActivity) getActivi-
yty()).tA.getLastSevenDays().size()>0){
      listView.setVisibility(View.VISIBLE);
      for (int i = 0; i < ((MainActivity) getActivi-
ty()).tA.getLastSevenDays().size(); i++) {
        String dateFormat = ((MainActivi-
ty)getActivity()).prefs.getString(getString(R.string.pref_d-
ate_key), getString(R.string.pref_date_default));
        SimpleDateFormat sdf1 = new SimpleDateFormat(dateFormat, Locale.ENGLISH);
        sdf1.setTimeZone(TimeZone.getTimeZone("UTC"));
        String date = sdf1.format(((MainActivity) getActivi-
ty()).tA.getLastSevenDays().get(i).getEndTime());
        hoursAdapter.add(date);
      }
      listView.setAdapter(hoursAdapter);
      listView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
          String time = Integer.toString(position);
          Intent intent = new Intent(getActivity(), DetailActivity.class);
          intent.putExtra(Intent.EXTRA_TEXT, time);
          getActivity().startActivity(intent);
        }
      });
    }else{
      listView.setVisibility(View.GONE);
    }
}

Listing 9. The fillList function from the MainActivity class.

The list in the array is filled first by being cleared. This ensures that no duplicate data is shown, and then by taking each item from the lastSevenDays list and adding it to the array used in the adapter.
The adapter is then attached to the listView defined in the layout for the fragment. An onClickListener is added to the list view which allows the position of the clicked item to be sent to the next activity, in this case the DetailActivity.

### 6.1.4 Detail Activity and DetailFragment

The DetailActivity class is a very simple activity. It simply initiates the layout, and then runs the fragment class. The use of fragments and activities is preferred as it allows for easier scalability of the application towards devices of different sizes. [19.]

The DetailFragment can then read the position sent to the DetailActivity as an intent extra; this position allows the item to be read from the lastSevenDays list and the various parts output.

### 6.1.5 SettingsActivity and SettingsFragment

The SettingsActivity class is another activity used to instantiate a fragment. The reason the settings menu is not simply run from the activity as opposed to a fragment is to ensure the visibility of the action bar, allowing a hierarchical structure to be imposed. If this was not used the back button would not be visible.

The final fragment is the SettingsFragment. This fragment is automatically generated by AndroidStudio and contains many examples of how settings can be implemented. For the purpose of this application only a few preferences were implemented. These are the date format and the size of the list, the maximum size of which is twenty five. If any value is input that is larger than this only twenty five records will be shown.

### 6.1.6 Asynchronous Tasks

Asynchronous tasks allow the class they are run in to asynchronously perform a task while continuing other tasks. This ensures that the main thread of a class does not become unstable. The WorkTracker application utilises four different asynchronous tasks for checking in, checking out, logging in and retrieving data. [18.]
The AsyncLogin class is the first asynchronous class called by user interaction. It sends the code entered by the user to the back end system where it is checked by the application. If a corresponding user ID is found, this is returned to the task along with the users' corresponding Wi-Fi name. These items are then returned to the service by the onPostExecute function which is automatically called once the doInBackground function is completed.

```java
@Override
protected void onPostExecute(String result) {
    super.onPostExecute(result);
    if(result != null){
        int id = Integer.parseInt(result);
        listener.loginSuccess(id, getWi-Fi());
    } else{
        listener.loginFail();
    }
}
```

Listing 10. An onPostExecute function from an Asynchronous Task class.

The function first checks that the result from the doInBackground function is not null, as this can result in a null pointer exception if attempted. Instead in case of a null result the loginFail method is called from the interface. The result and the name of the Wi-Fi is then sent through the interface method loginSuccess if the result is valid. This then calls further methods from the service, eventually leading to the update of the last-SevenDays list and the loading of the UI through the MainActivity and ManualFragment classes.

The second asynchronous task to be called after a user login is the class named AsyncListGrabber. This class requests the work time data stored in the database, adds the strings it receives to an object, and then finally adds the object to a list.

```java
if (response.containsHeader("backEnd")) {
    int size = Integer.parseInt(response.getFirstHeader("size").getValue());
    if (size != 0){
        int parserSize = Integer.parseInt(listSize);
        if(parserSize>size)
        {
```
parserSize = size;
}
try {
listTime = new ArrayList<WorkTime>();
for (int i = 0; i < parserSize; i++) {
    long wId = Long.parseLong(response.getFirstHeader("id" + Integer.toString(i)).getValue());
    int tracker = Integer.parseInt(response.getFirstHeader("trackingType" + Integer.toString(i)).getValue());
    long sTime = Long.parseLong(response.getFirstHeader("start" + Integer.toString(i)).getValue());
    long eTime = Long.parseLong(response.getFirstHeader("end" + Integer.toString(i)).getValue());
    WorkTime item = new WorkTime(wId, Integer.parseInt(id), tracker, sTime, eTime);
    listTime.add(item);
}
Listing 11. An if statement from the asynchronous task named AsyncListGrabber.

The function checks if the response exists as all responses from the backend will contain a named value pair with the parameter “backEnd”. This avoids null pointer exceptions if there is an error in the interface. The size of the received list is then checked as the size of the list is also a named value pair sent by the back end system. To ensure the loop does not attempt to read more values than is possible a check is done to ensure the requested list size is not larger than the actual list. Once all conditions are met successfully the list begins to parse. This for loop performs the reverse actions of the loop that is used to send the data from the back end.
Figure 24. De-serialization and serialization of objects from the back end system to front end.

The objects in the back end are taken from a list one at a time. They are then broken into a series of strings that can be sent using an HTTP Post response. These are then read by the Android client and added into a plain Java object and finally added to a list to be sent onwards into the application.

Finally there are the two tasks named AsyncCheckIn and AsyncCheckOut, the check-in task sends data to the back end system that is required by the back ends check-in methods. A check-in is then performed and the response returns the ID of the worktime entry created by the action. If the check-in was not successful the worktime entry ID is set to 1. This can then be identified as an erroneous check-in by the system and ignored. The rest of this class functions in the same manner as the AsyncListGrabber class as it is formed of a doInBackground method. A doPost method and finally a
onPostExecute method which sends the work time entry ID back to the application. The check out process runs the AsyncCheckOut task and sends the work time ID and the timestamp of the check out time to the back end system where a check out is performed. Once completed the onPostExecute method is called and it then informs the application that the check out has been completed. This is another method by which the list the user can see in the main view is updated.
Figure 25. Data flow from fragment to asynchronous task and back to fragment.
The coloured arrows in figure 25 represent interface actions. These interface classes allow interaction between the classes. The interface classes thus play a crucial role in ensuring the various information is passed correctly between active classes. The interface methods also ensure that the actions taken by the application occur in the correct order, by calling the interface methods only in the onPostExecute methods of tasks the subsequent methods in the service that are called by the interface method cannot be called until the task has been completed.

6.1.7 Broadcast Receivers

Broadcast receivers are classes that can receive implicit intents. These are broadcast by the operating system when an event occurs that can be handled by a number of different applications. In the case of Wi-FiChangeDetector class, the intent it is registered to listen to in the manifest is shown below in listing 12. [20; 21.]

```xml
<receiver Android:name="Wi-FiChangeDetector" >
  <intent-filter>
    <action Android:name="Android.net.conn.CONNECTIVITY_CHANGE" />
  </intent-filter>
</receiver>
```

Listing 12. The receiver registered in the Android Manifest of the Android application.

This will allow the receiver to be run whenever the intent is broadcast. The receiver then runs some further checks to ensure the Wi-Fi name matches the users’ assigned Wi-Fi name.

```java
if (networkInfo != null && Wi-FiInfo.getSSID().contentEquals(""+tA.getWI-FI_NAME()+"") )
```

Listing 13. This if statement is run when the broadcast receiver that listens for Wi-Fi changes is instantiated.

This allows the service and check-in process to verify if the Wi-Fi name is correct and there is a network connection.
The Bluetooth broadcast receiver BluetoothChangeDetector is in many ways similar. The receiver is registered to pick up a variety of intents to ensure backwards compatibility with older Android versions.

```xml
<receiver Android:name="BluetoothChangeDetector">
  <intent-filter>
    <action Android:name="BluetoothDevice.ACTION_FOUND" />
    <action Android:name="Android.Bluetooth.device.action.ACL_CONNECTED" />
    <action Android:name="Android.Bluetooth.device.action.ACL_DISCONNECTED" />
    <action Android:name="Android.Bluetooth.device.action.BOND_STATE_CHANGED" />
    <action Android:name="Android.Bluetooth.device.action.FOUND" />
  </intent-filter>
</receiver>
```

Listing 14. The receiver registered for Bluetooth intents in the Android Manifest.

Thus the receiver runs whenever a change occurs in the Bluetooth state. The receiver then checks if a beacon is one of the detected Bluetooth devices.

```java
BluetoothDevice device = intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
```

Listing 15. The object named device is instantiated in the BluetoothChangeDetector class.

The device name is collected in the object named device. These are then checked to ensure it matches the name of the beacon used. The first registered intent, BluetoothDevice.ACTION_FOUND, causes the receiver to run for each device found in the list. This ensures that the whole list of Bluetooth devices is checked for matching beacons.

These two receivers are the classes that allow the application to function in the background without the need for user input through a UI.
6.2 JavaEE

The Java application is split into three packages, the controllers, the models and the web packages. The classes are organised into three distinct packages. In the Control package the managed beans are handled, these classes control the logic of the application. In the Model package the entity classes are handled, these provide the model for the database entity objects. And finally, in the Web package, the servlets that control all outside interaction are handled. This structure allows the MVC model to be followed for the back end system.

6.2.1 Web

The Web package servlets are the main entry points to the application and control which methods are called from the managed beans. The servlet handling all method calls from the Android front end client is called simply Servlet. It has no actions for a GET HTTP call so no data is accessible unless the relevant POST headers are received. All post requests to the servlet must contain a POST header with the name "method", the value of which determines the actions and methods called by the servlet through a switch.

```java
dataHandler = new DataBean();
String method = request.getParameter("method");
```

Listing 16. The initiation of the DataBean class and accessing the value of a POST header.

On receipt of a request the doPost function first initialises a new instance of the DataBean class as this will allow it to read and write to the database if necessary.

The method then defines the next methods to be called.

```java
switch(method){
    case "checkUser":
```

Listing 17. The switch method in the class Servlet is called.

In this case if the string checkUser is equal to the string value of the method parameter the checkUser case will be run.

```java
    case "checkUser":
```
response.addHeader("backEnd", success);
out.println("<p>Method " + method + "</p>");

    int id = Integer.parseInt(request.getParameter("id"));

    List<Worktime> list = dataHandler.getWorkersTime(id);
    response.addHeader("size", Integer.toString(list.size()));
    out.println("<p>size " + Integer.toString(list.size()) + "</p>");
    for(int i = 0; i < list.size(); i++){
        String numerator = Integer.toString(i);
        Worktime item = list.get(i);
        out.println("<p>Numerator" + numerator + "</p>");
        response.addHeader("id"+numerator, (list.get(i).getId().toString()));
        response.addHeader("start"+numerator, String.valueOf(list.get(i).getStarttime()));
        response.addHeader("end"+numerator, String.valueOf(list.get(i).getEndtime()));
    }

Listing 18. The process triggered when the value of the POST header named method is checkUser.

The checkUser case allows a list of work time to be returned to the Android clients’ task AsyncListGrabber. This loop runs by taking a list, finding an object in the list, then iterating the object to strings to be sent to the front end.
Figure 26. Data flow in servlet designed to interact with the Android client.

The structure used for the checkUser method is carried into the other cases, with each case providing a unique set of instructions and inputs to the DataBean class.
The other servlet in this package functions in a very similar manner. However instead of outputting to an HTTP Post response object, the output is directed to the browser as a string containing the HTML DOM structure and content.

Each time a request is received, the user login is checked using the HTTP session

```java
HttpSession session = request.getSession();
webBean.setCodeEntered(request.getSession().getAttribute("code")
.toString());
```

Listing 19. The users HTTP session is referenced to the session object.

This method allows the session to be read and the login code entered to be checked before any further action is done.

6.2.2 Control

The Control package contains two enterprise beans. The first is a stateless bean named DataBean. This class handles all interactions between the database and the other classes by implementing the JPA API class EntityManager.

```java
public DataBean() {
    em = Persistence.createEntityManagerFactory("ServletPU").createEntityManager();
}
```

Listing 20. A public constructor of the DataBean class, instantiating the entity manager.

In the constructor of the bean the persistence manager is instantiated. This allows interactions with the database without requiring long SQL statements to be written each time. For example when a new user is added from the WebApp servlet the addEmployee method is called.

```java
public String addEmployee(){
    try{
        Employee toAdd = new Employee();
        toAdd.setId(toAdd.getId());
        toAdd.setFname(getFirstName());
        toAdd.setLname(getLastName());
    }
    
```
toAdd.setRole(findPermissionLevel(getRoleId()));
toAdd.setSupervisor(findEmployee(getSuperId()));
t = em.getTransaction();
t.begin();
em.persist(toAdd);
t.commit();
return "success";
}
catch(NullPointerException e){
return "fail";

Listing 21. The addEmployee function, which interacts with the database.

The function is run inside a try and catch bracket to ensure that all fields are present. Otherwise an erroneous entry may be input into the database. The entity manager can also be used to find an item already in the database as shown in the findEmployee function shown in listing 22.

    public Employee findEmployee(int id){
        Employee found = em.find(Employee.class, id);
        return found;
    }

Listing 22. A function that allows the details of an employee to be retrieved from the database.

The find function in the EntityManager class takes the entity class to use and its primary key to find a match. This object can then be used to retrieve any other information stored regarding the employee in the database.

The final function of the EntityManager used in this class is the createNamedQuery function. This allows an SQL query to be defined in the entity class, which can then be called. The function can then return a list with the object type defined as the entity class chosen. This can be seen in the getter functions for the various lists as shown in listing 23.

    public List<Worktime> getListWorkTime() {
        listWorkTime = em.createNamedQuery("Worktime.sortAscending").getResultList();
        return listWorkTime;
    }

    public List<Worktime> getListWorkTime() {
A getter function for a list.

In this function the work time list has been requested, the variable listWorkTime is then instantiated as the result of the named query, which is defined in the entity class Worktime.

The second bean is named WebBean and is a session scoped managed bean that handles certain requests from the WebApp servlet before interacting with the DataBean instance. These functions are specific to the output required for the WebApp class. The functions are kept separate from the DataBean class itself to avoid interference. This can be seen most clearly in the function userNameGrabber shown in listing 24.

```java
public String userNameGrabber(){
    setLoggedUser((dBean.fetchUserId(getCodeEntered()).getEmployee()));
    try{
        String userFull = loggedUser.getFname()+" "+
                            loggedUser.getLname();
        return userFull;
    }catch(NullPointerException e){
        return "null";
    }
}
```

Listing 24. The function userNameGrabber from the WebBean class.

The function sets the user that is logged in and returns the full name of the user to be output in the browser of the user. If the user has not logged in the function returns a string that the servlet reacts to by re-directing the user to the login screen.

6.2.3 Model

The model classes provide a structure for the data read from the database. Each class matches a table in the database and contains setters and getters for each attribute. As the database is hosted on the localhost of the server the application is running on, this
ensures that the only interactions possible with the database are through the entity classes, this structure adds a level of security not found with databases stored on more open servers. The standard named queries created can be added to with simple SQL statements, for example to ensure the work time objects are sorted with the latest completed entry first a custom named query can be created as shown in figure 25. [22.]

```java
@NamedQuery(name = "Worktime.sortAscending", query = "SELECT w FROM Worktime w ORDER BY w.endtime DESC")
```

Listing 25. A named query from the WorkTime entity class.

This works by selecting all from the Worktime table and sorting it in a descending manner. This ensures the largest entries are listed first, as the entries are note dates but a number, specifically the number of milliseconds since the 1st January 1970, this allows us to find the most recent entries to the database first. [17.]
7 Testing and Results

Testing of the application was completed in two phases. Phase one a functional testing program was to ensure that all functionality was in place and communicating correctly and was performed on several different devices. Phase two was carried out with a more open approach and would allow feedback regarding usability and user experience.

7.1 Phase One

Phase one required testing on several platforms and screen sizes to ensure the functionality of the UI was not compromised by scaling. The devices chosen for this test are listed below.

- Samsung Galaxy S4 Active - Android Lollipop 5.0.1 Touchwiz
- Samsung Galaxy S4 4G - Android Lollipop 5.0 Touchwiz
- Samsung Galaxy Fame - Android JellyBean 4.1
- Google Nexus 7 - Android Lollipop 5.1

Each user was assigned a tester code and then asked to use the device for several days, performing manual check-in events, manual check out events and attempting to trigger automatic events.

The results indicate that the functionality works well across the devices used in testing. However it was found that users with devices running Android Lollipop 5.0 could not trigger a check-in event using the LE Bluetooth beacons. This was found to be due to a different intent broadcast by the operating system compared to older versions of Android. The new broadcast intent was thus added to the intent-filter list of the receiver to ensure newer versions of Android were also compatible with the application.

Several users reported difficulties with connecting to the Bluetooth beacons. This was eventually resolved by increasing the power of the broadcast of the beacons. The inability to connect was determined to be caused by thick cases on the mobile device. Once a case was removed the beacon could be detected even at the low-
est power setting. This leads to a compromise being implemented of beacon range versus battery life. However as the power usage continues to be quite low, battery issues should not become a problem for shorter ranges.

7.2 Phase Two

This test phase focused on the user experience and user interface elements. The users were asked for open feedback and to rate the user interface, how often they checked if a check-in or out was successful and if they manually checked in at any time. While the results provide an interesting insight into how the application can be developed in the future the results cannot be used to draw any larger conclusions due to the small sample size.

Phase two was performed with a larger test group running the following devices:

- Samsung Galaxy S4 Active - Android Lollipop 5.0.1 Touchwiz
- Samsung Galaxy S4 4G - Android Lollipop 5.0 Touchwiz
- Samsung Galaxy S5 - Android Lollipop 5.0.1 Touchwiz
- Samsung Galaxy Fame - Android JellyBean 4.1
- Google Nexus 7 - Android Lollipop 5.1
- HTC One Plus One - Android KitKat 4.4

An interesting point to investigate further would be the anxiety some users expressed about the automatic check-in. It appears that many of the logins were performed specifically to check if a check-in or out had occurred. To avoid this problem in future development a widget could be developed which shows both the current status and the method that was used to achieve it.

7.3 Open Feedback

An issue that was raised by a tester was the problem of working from home. This is working time that can be difficult to monitor. The application cannot currently adapt to a different work environment than is setup for the user. Thus a manual check-in is re-
quired for any work time reported outside of normal working locations where a beacon has not been provided.

The application handles events that fit neatly into the two automatic categories well. However the manual check-in needs to continue to be available to handle unexpected events as there are many possible work time events that could occur outside of the normal work environment.

As the application can be used for payroll purposes it has become apparent through testing that a method of manually editing and changing the times shown may be necessary to correct any errors or problems with the system. It should also be possible to insert times where the user should be paid, but may not have been present at work. This will allow sick leaves and other such absences to be accounted for in the system.
8 Conclusion

The application developed for this project shows that there are alternatives to traditional methods, and that by utilising existing infrastructure working hours can be tracked without requiring any physical installation of readers. This would allow a company that does not have the resources to implement an electronic, automatic working time monitoring system in compliance with the relevant law.

The application is not without its problems. The main problem of the application is an issue that many other Android applications face, namely one of fragmentation. As the popularity of the operating system has increased so has this problem. Currently to achieve maximum reach applications need to target Jelly Bean (Android 4.1.x). This will allow the application to be compatible with over eighty-five percent of all Android devices as shown by the fragmentation chart in figure 27. [23]

![Fragmentation Chart](image)

Figure 27. A chart detailing the fragmentation of the Android operating system. [23]

This requirement lead to the decision to not use the Estimote SDK as it requires Android 4.3, as this would exclude approximately fifty percent of all Android devices. Instead standard Bluetooth features were implemented. This implementation is also different in the various Android operating systems in use, leading to several intent filters being required for the broadcast receiver designed to listen to changes in Bluetooth activity. [21]
However the mobile application approach may not be practical for all companies. If access control is also required and it can be combined with the hour tracking, then this can lead to a useful implementation where an employee can use the same tag for two purposes. It may also be very expensive to provide compatible devices for all employees. This can be resolved by further development of the application onto other platforms such as iOS and Windows devices.

The back end system can also be further upgraded to adapt to changing consumer needs. For this project a JavaEE application was utilised with a Tomcat server, but the application could equally have been built and designed using ASP.NET in C# and Azure hosting. This approach could be favoured if the resources were available to develop in this direction as using Microsoft’s Azure tools are not free, but provide a great deal more tools, compared to Tomcat which is a free open source project developed by the Apache Software Foundation. [24.]

For future work monitoring applications a paradigm shift could be considered. The current model for mobile applications requires the user to have a compatible device, and access to an internet connection to connect to a remote service. As the cost of LE Bluetooth beacons reduces it becomes possible for each user to be assigned an individual beacon, with a reader stationed in the workplace. This would bring many of the benefits of a mobile application as well as many of the benefits of an RFID system. One major advantage of a mobile application is that the user does not need to take any action other than ensuring the beacon is on their person. Meanwhile the burden of cost for the hardware and setup is shifted to the employer. A system such as this would bring additional benefits in industries where the specific location of each employee is important, for example for security guards in a warehouse. The beacons would allow the location of each guard to be known to an operator in case of emergencies. However such systems would require a significant increase in the security setup of the beacons to ensure the information could not be abused by third parties.
References


13 Your First Cup: An Introduction to the Java EE Platform [online], Oracle Corporation 2012.

14 Material Design, Google. 2014


16 FloatingHintEditText [online], Brian Nicholson May 2014.

17 Date [online], Java Platform,

18 AsyncTask – Android Developers [online], Google


20 Intents and Intent Filters – Android Developers [online]. Google

21 BroadcastReceiver - Android Developers [online]. Google

22 Java Persistence API [online]. Java, Oracle Corporation

23 Dashboards – Android Developers [online]. Google
24 Apache Tomcat [online]. Apache Software Foundation
Implementation Guide

Back End System

Server requirements:

Apache Tomcat server, minimum requirement version 8.0.15

MySQL server running on localhost of Apache Tomcat server, minimum version 5.5.42 - MySQL

First ensure a MYSQL database is running on the local host of the Tomcat server the application will run on. Then download the application from GitHub and run the following in your git terminal.

```bash
cd my/project/path
git clone https://github.com/MikeFynes/WorkTrackerBackEnd.git
git fetch && git check out demoFinal
git pull
```

Listing 1. Git code for creating a local repository of the back end system.

Where my/project/path is wherever you intend to save the project on your device.

Open the project in Netbeans or your preferred IDE and change the database path, username and password from the file persistence.xml. This will ensure the application connects to your database and does not attempt to connect to a database that does not exist.

Clean and build the application. If you are using Netbeans you can now find the .war package in your project path folder named “dist”. This is the file you will need to upload to your Tomcat application server.
For testing purposes codes for accessing the front end application have been added manually to the MySQL table named codes, replace this system with your login system of choice.

**Front End Client**

Device requirements

Android JellyBean 4.1 or higher

Download the application from GitHub and run the following in your git terminal.

```bash
cd my/project/path
git clone https://github.com/MikeFynes/HourKeeper.git
git fetch && git check out demoTwo
git pull
```

Listing 2. Git code for creating a local repository of the front end client.

Open the project in Android Studio or your chosen IDE and edit the class named TrackerApp, amend the following string:

```java
private final String SERVER_URL = "PATH TO YOUR SERVLET HERE"
```

Listing 3. The string with the server path that needs to be updated.

for example:

```java
private final String SERVER_URL = "http://my.server.com/Servlet/Servlet";
```

Listing 4.

You can then build an APK that meets your criteria and use the application using the codes defined in the back end implementation.