**System Requirements and Design**



Attendance System

Kevin Brueland – Chi Moen Noel Lam – Morten Liknes

Preface

This document is a combination of two documents which are normally separated; Software Requirements Specifications and Software Design Documents. This merger has been decided due to the relatively small size of the project, and will provide both the development team and stakeholders with a better overview.

For the sake of brevity, a significant portion of the source code has been omitted from this document. However, the entire source code for all modules of the CheckPoint Reading Terminal is available online by visiting the project’s homepage: <http://tfweb.hit.no/2017/Checkpoint> and navigating to the “Documents” tab.

Current document version and its historical changes can be viewed in *Table 1‑1: Document version control.*

Table ‑: Document version control

|  |  |  |
| --- | --- | --- |
| **Document version** | **Dated** | **Whats new?** |
| 1.0 | 17.01.2017 | Document created |
| 1.1 | 24.01.2017 | Added flowcharts and figures, minor structural changes |
| 1.2 | 31.01.2017 | Added database model figures, added database and UML chapters |
| 1.3 | 13.02.2017 | Added GUI mockups |
| 1.4 | 29.02.2017 | Added chapter on REST API |
| 1.5 | 23.03.2017 | Added chapter on Reading Terminal |
| 1.6 | 04.05.2017 | Sequence diagrams moved to downloadable file on project website |

Nomenclature

API – Application Programming Interface

DAL – Data Access Layer

DTO – Data Transfer Object

GPIO – General-Purpose Input/Output

GUI – Graphical User Interface

HTTP – Hypertext Transfer Protocol

JSON – JavaScript Object Notation

MVC – Model View Controller

MVP – Model View Presenter

NUC – Next Unit of Computing

PC – Personal Computer

REST – Representational State Transfer

RFID – Radio Frequency Identification

RS232 – Recommended Standard number 232

SDD – Software Design Documents

SRS – Software Requirements Specifications

SQL – Structured Query Language

UML – Unified Modelling Language

Table Of Contents

Preface 2

Nomenclature 3

Table Of Contents 4

1 Introduction 6

1.1 Purpose 6

1.2 Scope 6

1.3 Similar Existing Attendance Systems 6

2 System Overview 8

2.1 Module Breakdown 8

2.1.1 Computer 8

2.1.2 Web Application w/ Database 9

2.1.3 Reading Terminal 9

2.1.4 Arduino Microcontroller 9

2.1.5 RFID Reader 9

2.1.6 RFID Tag 9

2.2 System Flow 10

3 Technical Requirements 11

3.1 Functional Requirements 11

3.1.1 System 11

3.1.2 Client 11

3.1.3 User 11

3.1.4 Attendee 11

3.1.5 Host 12

3.1.6 Appointment 12

3.1.7 Course 12

3.1.8 Reading Terminal 12

3.2 Non-Functional Requirements 13

3.3 Graphical User Interface Specification 13

3.3.1 Non-client specific GUI 13

3.3.2 User GUI 14

3.3.3 Host GUI 17

3.4 Inter-Module Data Communication 22

4 System Architecture 23

4.1 Three-Tier Multi-layer Structure 23

4.1.1 Presentation Layer 24

4.1.2 Business Layer 24

4.1.3 Data Access Layer 24

5 Database 26

5.1 Logical and Physical Data Model 26

5.2 Database Description 27

5.2.1 APPOINTMENT Table 27

5.2.2 CLIENT Table 28

5.2.3 CLIENT\_TYPE Table 28

5.2.4 CLIENT\_TAG Table 28

5.2.5 ATTENDEE\_STATUS Table 28

5.2.6 ATTENDEE Table 29

5.2.7 COURSE Table 29

5.2.8 READER Table 29

5.2.9 READER\_APPOINTMENT Table 29

5.2.10 ADDRESS Table 29

6 RESTful service 30

6.1 CheckPoint REST API 30

6.1.1 Clients and Roles 31

7 Reading Terminal 33

7.1 Hardware Configuration and Circuit Diagram 33

7.2 Transfer Protocol 33

8 UML 34

8.1 Use Case Diagram 34

8.2 Use Case Documents 35

8.2.1 Register 35

8.2.2 Log in 36

8.2.3 Become Attendee 37

8.2.4 Manage Profile 37

8.2.5 Create Appointment 38

8.2.6 Manage Appointment 39

8.2.7 Create Course 40

8.2.8 Manage Course 41

8.2.9 Manage Attendance 42

8.2.10 Attend Appointment 43

8.2.11 Create Report 44

8.3 Sequence Diagrams 45

8.3.1 User Registration 46

8.3.2 Host Registration 47

8.3.3 User Login 48

8.3.4 Host Login 49

8.3.5 User Update Profile 50

8.3.6 Host Update Profile 51

9 Class Diagram and Structure Maps 52

9.1 Class Diagram: High-Level Overview 52

9.2 Presentation Layer Class Diagram: Model View Presenter 54

9.3 CheckPoint Business Layer & Data Access Layer Class Diagram: Repository Pattern 60

9.4 StructureMap Inversion of Control: Inversion of Control Container 65

# Introduction

Some text

## Purpose

The purpose of this document is to present a detailed description of the CheckPoint Attendance System. It will explain the purpose and features of the system, the interfaces of the system, what the system will do and the constraints under which it must operate. This document is intended for both the stakeholders and the developers of the system.

## Scope

The Checkpoint Attendance System will be a web based attendance system capable of registering and verifying attendance to a wide range of event types. This could include students attending obligatory course events or workplaces holding compulsory meetings for employees. Often, such systems are implemented with attendances being manually recorded on paper. This approach can be both error prone and time consuming, especially when collating data and calculating individual levels of attendance for larger events.

In general, the system will allow anyone hosting an appointment or event to efficiently track attendance, removing the cost-ineffective and time-consuming task of manually tracking and managing attendance. In addition, it will allow users to easily discover and sign up for local and national events thought their web based user panel.

An extension to this is the ability to use CheckPoint as a ticket system, allowing users to pay a host of an event to be included on the attendance list and then using their unique RFID tag identifier to verify their payment and gain access to the event.

A second extension to this is the usage of CheckPoint as a safety measure in the industry, by allowing employees to register their attendance at the fire assembly point in the case of a fire or accident. This would allow the shift team leader to keep track of all the employees and make sure everyone is accounted for.

## Similar Existing Attendance Systems

There are a few similar solutions already on the market, one prominent system being “myAT”, found at [www.myattendancetracker.com](file:///C:\Users\Bruker\Downloads\www.myattendancetracker.com), which is a web application specifically targeting attendance tracking in schools. It features an online system that allows a teacher to create classes and add students. A teacher can request attendance reports and perform more school-specific tasks like creating grading scales and evaluation assignments.

The main drawback with “myAT” is that it requires you to log the attendance manually, marking students either present, absent or late while being logged in to the web page. If the teacher does not have access to a computer connected to the internet, or is unable to be present at the beginning of a class, the system does not perform its intended task. The CheckPoint Attendance System automatically registers the attendance using unique RFID tags, eliminating the need for manual recording. All attendance is checked and verified post-appointment, assuring all attendance is recorded even if the host is absent.

Another similar attendance system is “uAttend”, found at [www.uattend.com](file:///C:\Users\Bruker\Downloads\www.uattend.com). It is also a web based attendance software marketed towards small businesses, more specifically for handling working hours and breaks. The manager is able to create work schedules and add employees, which works similarly to creating repeating appointments and adding the employees as attendees. It utilizes biometric registration of attendance through fingerprint scanning and allows employees to quickly and easily log in and out.

The requirement for physical attendance registering is more in line with the main idea behind the CheckPoint Attendance System, but the use of biometric parameters for registering attendance is controversial in terms of privacy and laws regarding its use may be subject to change. Therefore, the use of RFID tags is a safer and more sustainable solution.

# System Overview

An overview of the general system structure is illustrated in *Figure 2‑1: General system structure.* This maps the different modules of the system and shows how they are connected. The “Host Package” refers to the modules that is delivered to a host managing an appointment. A more detailed breakdown of each of these modules can be found in *2.1 Module Breakdown.*

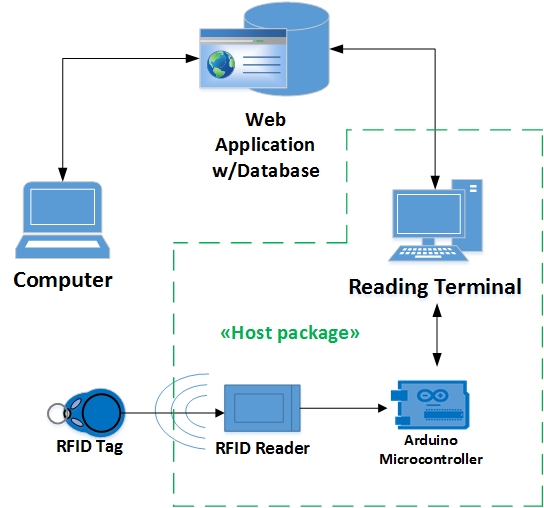


Figure 2‑1: General system structure

## Module Breakdown

This section gives a more detailed breakdown of the modules found in *Figure 2‑1: General system structure,* explaining each modules purpose and technical specifications. An overview of the data flow between the different modules can be found in *3.4 Inter-Module Data Communication* .

### Computer

The computer module represents the entry point for the CheckPoint Attendance System, and is the main tool for employing the applications functionality. An active internet connection together with a web browser is required to use the application. The computer can be of any type; desktop, laptop or mac.

### Web Application w/ Database

The web application represents the main software module and the part of the system which interacts directly with the user. It hosts the user interface in the form of web forms and contains the applications business logic and DAL. For a more detailed explanation of the applications architecture, see *4. System Architecture.*

The web application and the associated SQL database resides on an external Microsoft Azure server. This assures a stable uptime, good backup security and a dynamic storage solution to meet an increasing user base.

### Reading Terminal

The reading terminal represents the module that is deployed at the appointment location and contains the software responsible for validating and logging the attendance to the database. The host managing the appointment rents the reading terminal, and it is part of the “Host Package”.

A standard terminal consists of a windows based NUC mini-pc, and has no external monitor attached. It requires an active internet to connect and validate with the database, but can also store attendance locally on file if the internet connection fails.

The reading terminal is described in greater detail in *Chapter* *7 Reading Terminal.*

### Arduino Microcontroller

An Arduino microcontroller is attached to the reading terminal and is responsible for interpreting the RFID tag id registered by the RFID reader. It then transfers this id to the reading terminal. The microcontroller is also responsible for displaying the validation status to the attendee after the RFID tag has been scanned.

The microcontroller is an Arduino Uno Rev 3 and is part of the “Host Package”.

### RFID Reader

The RFID reader registers the RFID tag and transmits it to the Arduino microcontroller

The reader is of type MFRC522.

### RFID Tag

The tag is issued to each newly registered user and is used to uniquely identify the attendee at an appointment. The attendee swipes his tag on the RFID reader to register his attendance.

The tag is a passive type with a frequency of 13.56MHz.

## System Flow

The flow chart seen in *Figure 2‑2: Detailed System Flow* describes the system flow in the CheckPoint Attendance System.

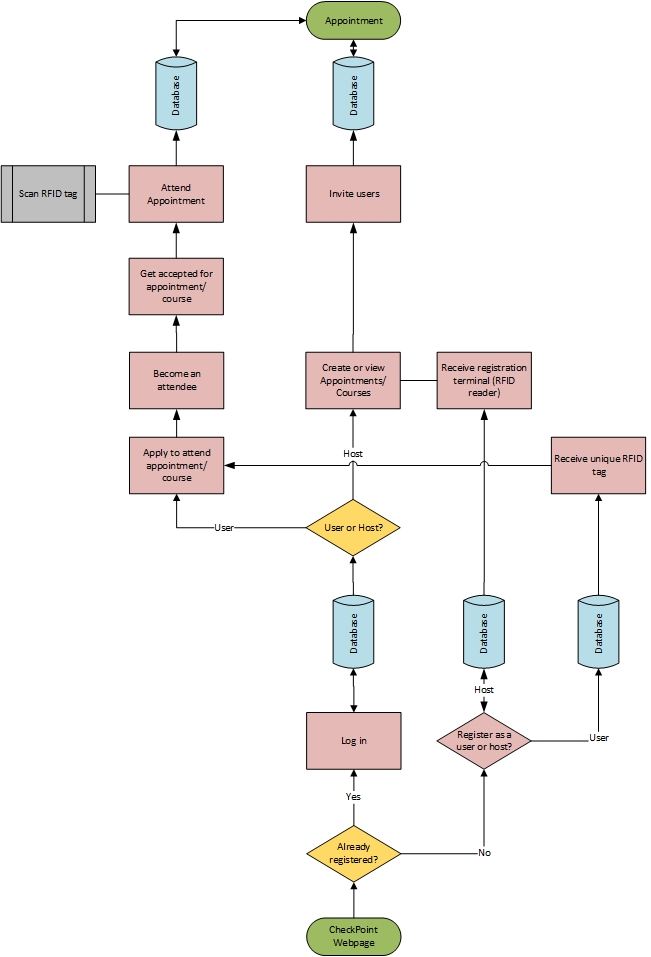


Figure 2‑2: Detailed System Flow

# Technical Requirements

The system requirements can be divided into two main groups: functional and non-functional.

## Functional Requirements

These requirements are essential to the functionality of the software and will transform into use cases, which are handled in *8.1 Use Case Diagram.*

### System

The system is defined as the application software in combination with the database. The following represents the functional requirements for the system:

* The system must be a web-based application.
* The system must support being accessed by multiple simultaneous users without disruption.
* The system must be able to run on Google Chrome, Mozilla Firefox and Opera web browsers.

### Client

A client is defined as an unregistered person and does not yet have an active account in the CheckPoint system. The following represents the functional requirements for a client:

* The client must be able to register and create a new account either as a user or a host.

### User

A user is defined as someone who has registered with the system, but has not yet applied to attend an appointment. A user receives an RFID tag, which functions as the user’s unique identifier, after registering. The following represents the functional requirements for a user:

* Each user of the system must be uniquely identified by his or her RFID tag number.
* All data entry must be validated to avoid duplicate identification fields.
* In the event that data entry validation is unsuccessful the user must be provided with the necessary feedback about this through the webpage.
* A user must log in to the system with a username and password.
* A user must be able to log out of the system.
* A user must be able to view all non-private appointments
* A user must be able to apply to attend any non-private appointments
* A user must be able to change their login details.
* A user is required to submit a username, first name, last name and email.
* A user must have the ability to edit their details whenever necessary.

### Attendee

An attendee is defined as a user who has applied to attend an appointment and is either awaiting acceptance or has been accepted by the host of that appointment. Once the appointment(s) are finished, the attendee goes back to being a user. The following represents the functional requirements for an attendee:

* An attendee must be able to log their attendance at an appointment with his or her RFID tag.

### Host

A host is defined as someone who has registered with the system as a host, and is someone who hosts/creates appointments for attendees to attend. A host will receive a reading terminal to use at his/her appointment(s) when registering. The following represents the functional requirements for a host:

* A host must be able to create/read/update/delete a new appointment or course.
* A host must be able to add users to and remove attendees from appointments and courses.
* A host must log in to the system through the website with a username and a password.
* A host is required to submit a username, first name, last name and email.
* A host must be able to log out of the system.
* A host must not have the ability to edit the attendance logs.
* A host must be able to request/ order an RFID reader.
* A host must be able to generate a customizable attendance report.

### Appointment

An appointment is defined as a single event or happening that attendees can attend. The following represents the functional requirements for an appointment:

* An appointment must be created with a name, location, date, starting time, ending time, description and the host’s username.
* An appointment must be possible to set as either public or private.
* An appointment must be able to be set as obligatory or not.

### Course

A course is defined to be a collection of appointments allowing appointments to be grouped together. The following represents the functional requirements for a course:

* A course must be able to add and remove appointments.
* A course must be possible to set as either public or private.
* A course must be able to be set as obligatory or not.

### Reading Terminal

The reading terminal is defined as a device responsible for registering an attendee’s attendance, by reading the RFID tag and registering the attendance in the database. The following represents the functional requirements for a reading terminal:

* The reading terminal must be able to store registrations locally to a text file as backup.
* The reading terminal must be able to connect to the internet and communicate with the server.
* The reading terminal must display the validation status on a screen to the attendee.

## Non-Functional Requirements

These requirements are related to performance and convenience, and are equally subject to testing. They might eventually become functional requirements.

* The reading terminal should be responsive enough to process and register attendance within 3 seconds.
* The system should have an uptime of 22 hours per day.
* The GUI should not require the user to have any prior technical experience in order to navigate the system.
* The database storage should be easily scalable to meet future demands of a dynamic user base.

## Graphical User Interface Specification

The GUI will be a webpage accessed through a browser and is the primary point for users and hosts to interact with the Checkpoint System. All available functions should be self-explanatory and made simple to navigate. Included in this chapter is a set of first drafts for the Checkpoint system GUI.

### Non-client specific GUI

The following drafted UI pages are pages that anyone entering the checkpoint systems webpages will be able to access.

Figure 3‑1 shows the homepage of the Checkpoint system. From here, it is possible to navigate to the “log in” or “register” page, read about the latest features, contact information and learn more about the Checkpoint system.

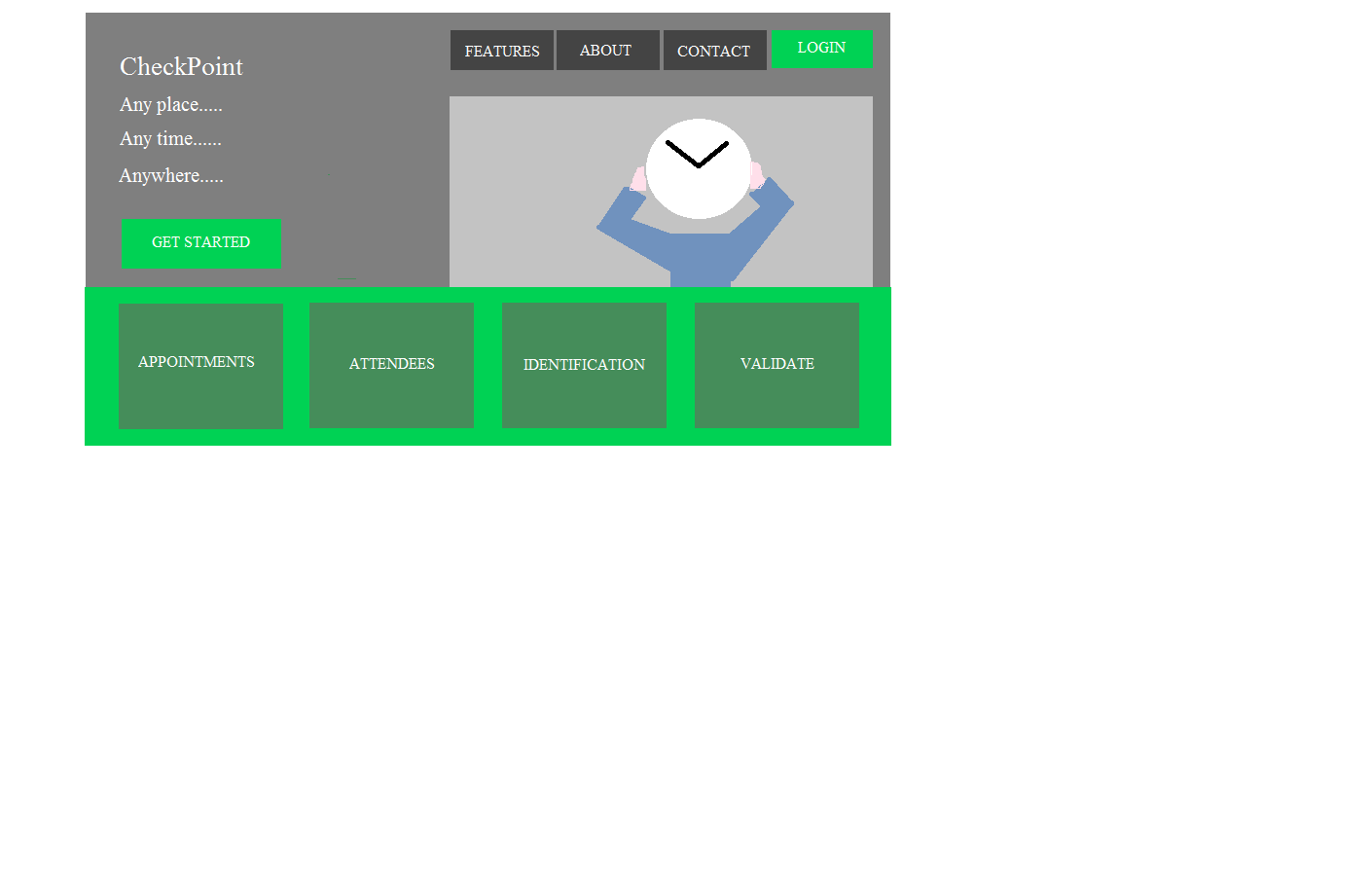


Figure 3‑1 : Draft of home page

Figure 3‑2 shows the register page, here one can choose to either register as a user or host.

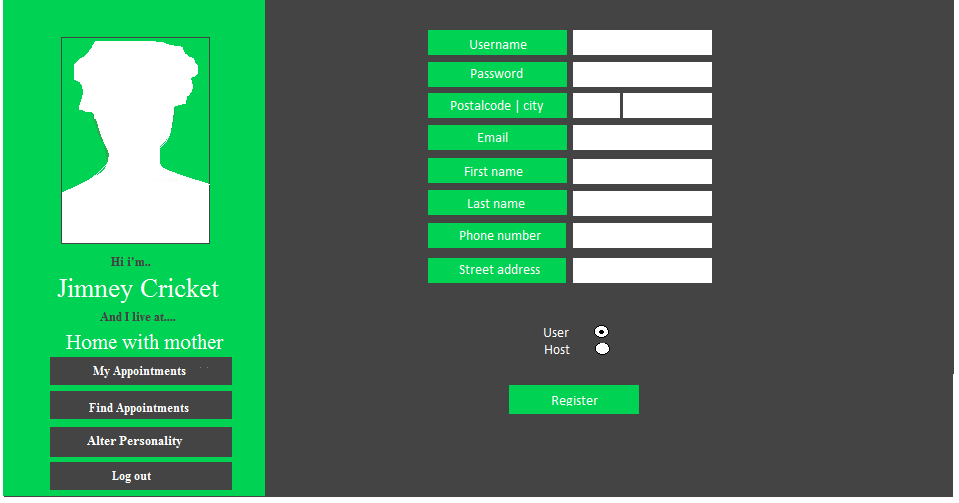


Figure 3‑2: Draft of register page

Figure 3‑3 shows the log in page for both a user and host.

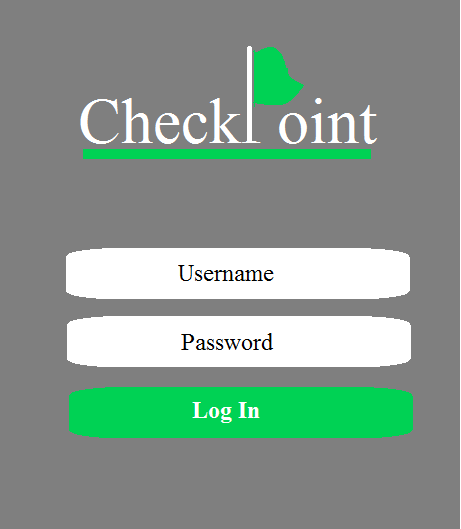


Figure 3‑3: Draft of log in page

### User GUI

The following drafted UI pages describe the pages that only a successfully logged in user will be able to access.

Figure 3‑4 shows the first page a user will see when he/she has logged on to the Checkpoint system. On this page the user can choose to get a view of his/her future appointments or courses with the relevant details. The left side bar will display some user details and provide the user with the opportunity to navigate other pages that contain extended functions or log out.



Figure 3‑4: Draft of user page

Figure 3‑5 shows the “find appointments page”, this page gives the user opportunity to search and apply to attend different public and private courses and/or appointments. The page can be accessed from the left side bar.



Figure 3‑5: Draft of find appointments page

Figure 3‑6 shows the edit user page which can be accessed from the left side bar.

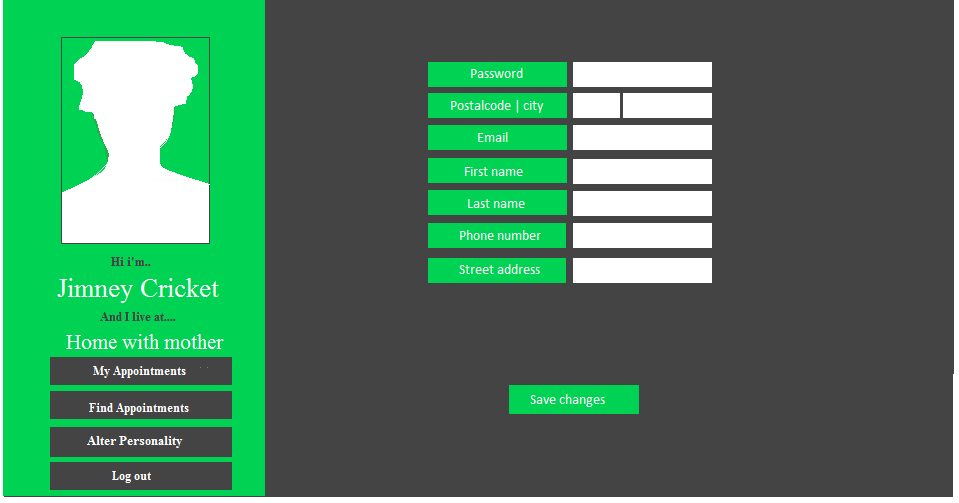


Figure 3‑6: Draft of edit user page

### Host GUI

The following drafted UI pages describe the pages that only a successfully logged in host will be able to access.

Figure 3‑7 will be the first page a host will see after he/she has logged in. Through this page the host will see his/her appointments or courses and get access to the update or manage attendance page of the selected appointment or course. The host also gets access to the create course, appointment or report page.

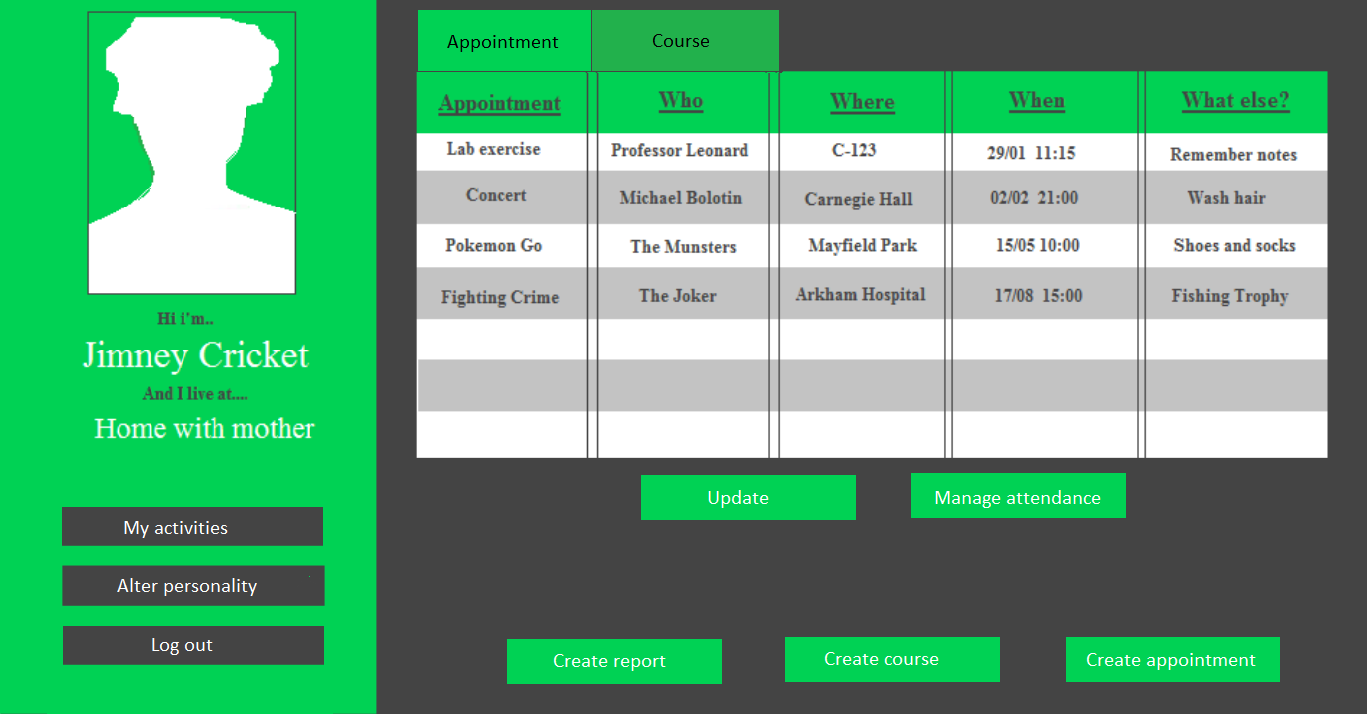


Figure 3‑7: Draft of host page

Figure 3‑8 is a draft of the “create appointment” page. Here the host fills in the appointment details before creating an appointment.

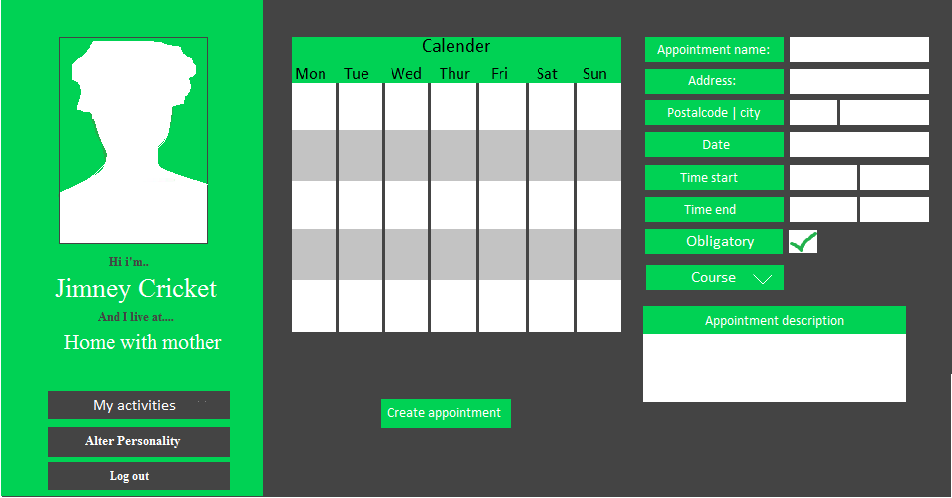


Figure 3‑8: Draft create appointment page

Figure 3‑9 shows the “update appointment” page. Here the host can edit his appointment details and cancel the appointment.

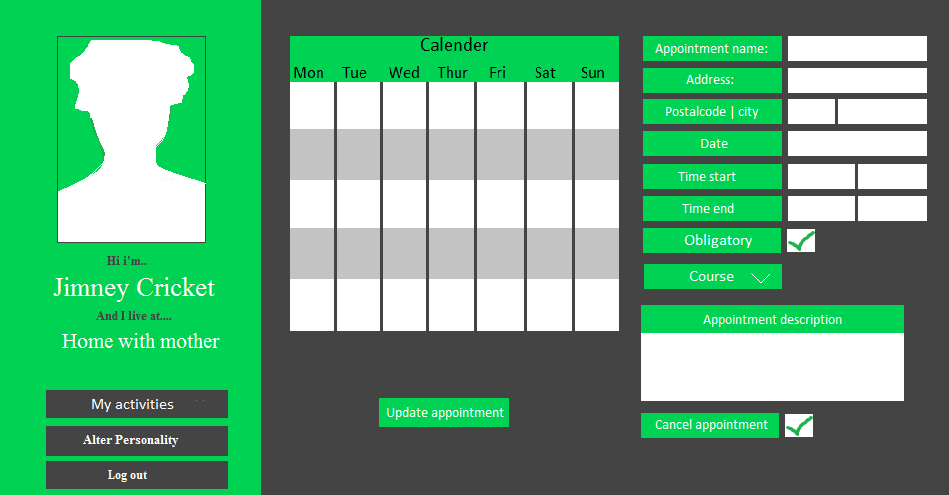


Figure 3‑9: Draft for update appointment page

Figure 3‑10 Is a draft of the “create course” page. Here a host can create a course with appointments that already exists, or he/she can choose to create default appointments on certain dates in the course.

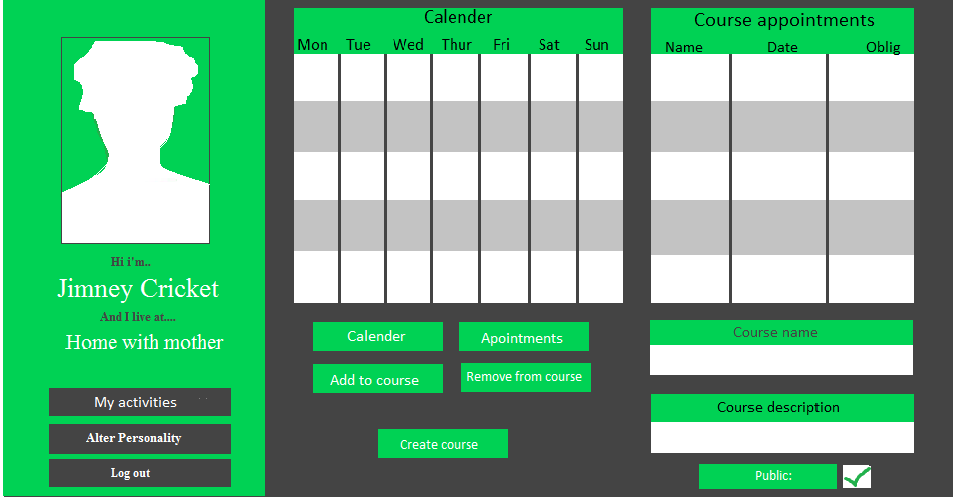


Figure 3‑10: Draft create course page

Figure 3‑11 is a draft of the “update course” page, the host can update course details and cancel his course in this page.

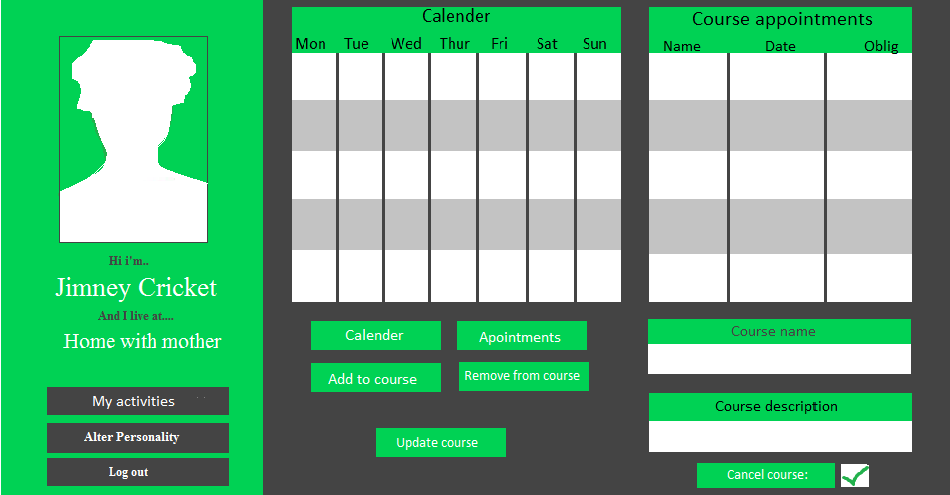


Figure 3‑11: Draft update course page

Figure 3‑12 is a draft of the “create report” page. In this page a host can choose to create a report by appointments or courses

.

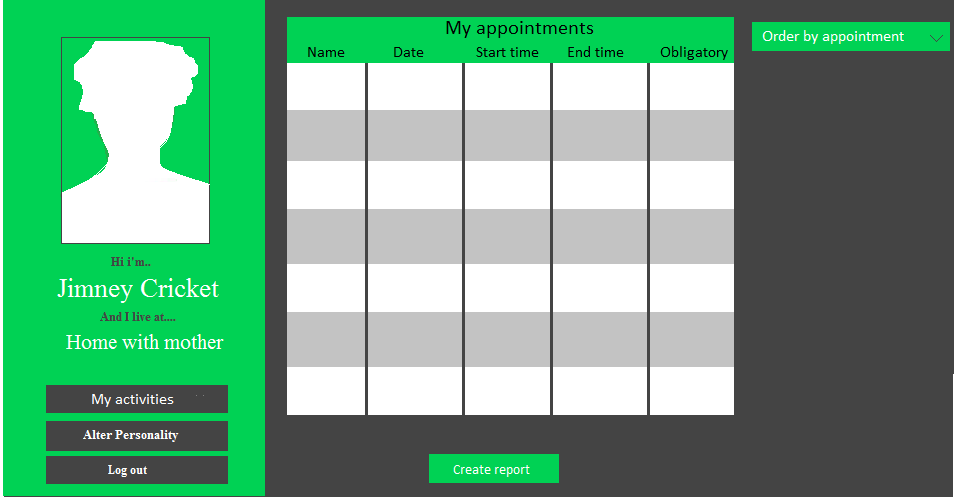


Figure 3‑12: Draft create report page

Figure 3‑13 shows the “edit host details” page.

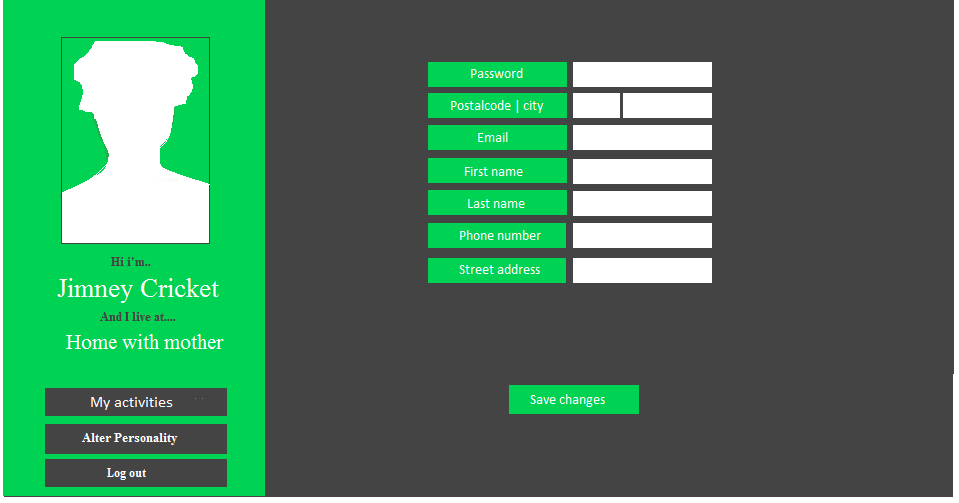


Figure 3‑13: Draft edit host page

Figure 3‑14 shows the “manage attendance on course page”. Here the host can add applicants to the course or distinct appointments in the course.

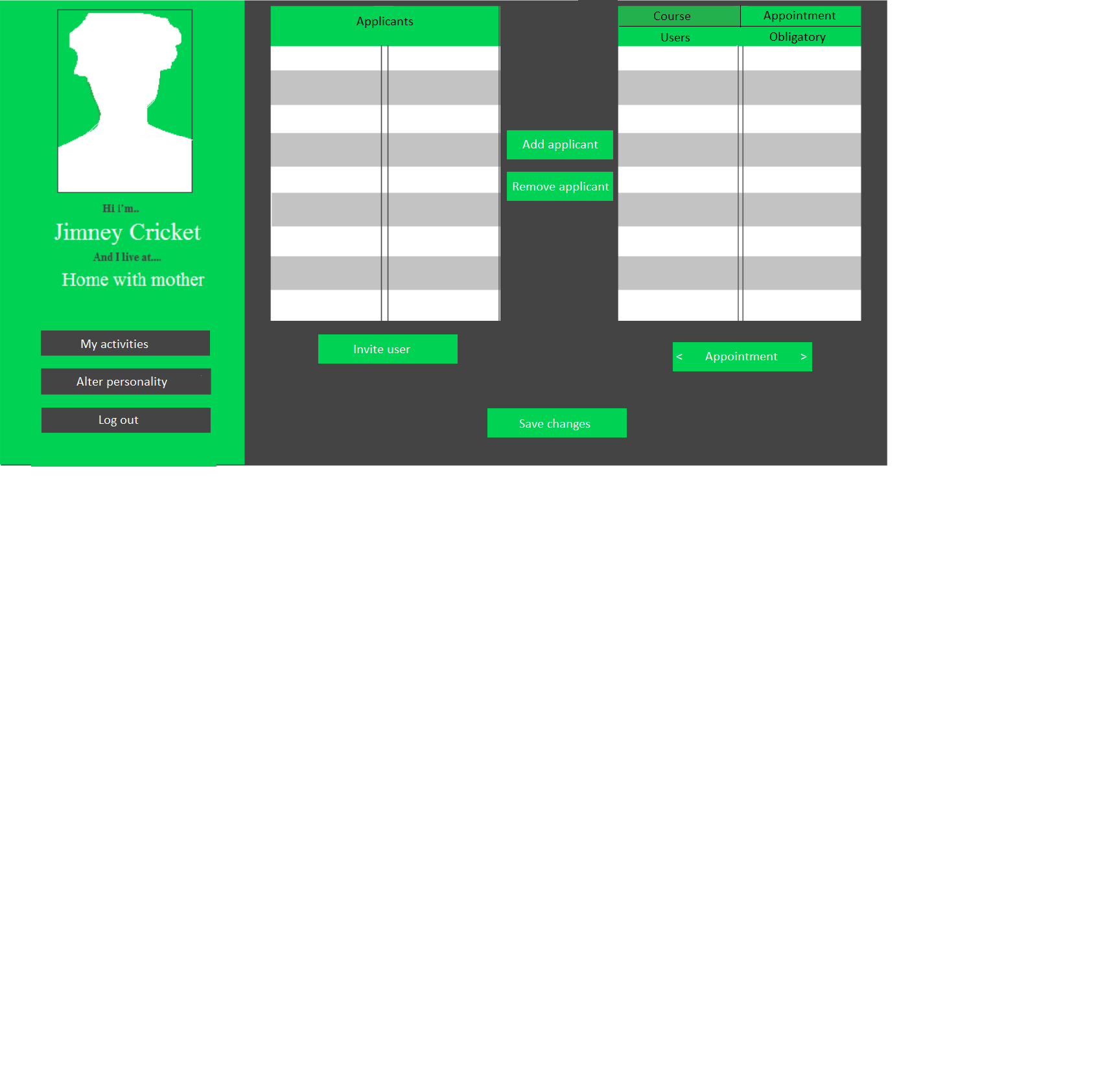


Figure 3‑14: Draft of manage attendance on course page

Figure 3‑15 shows a draft of the “invite user page”. This page will be displayed if the host chooses to invite a user to an appointment or course.

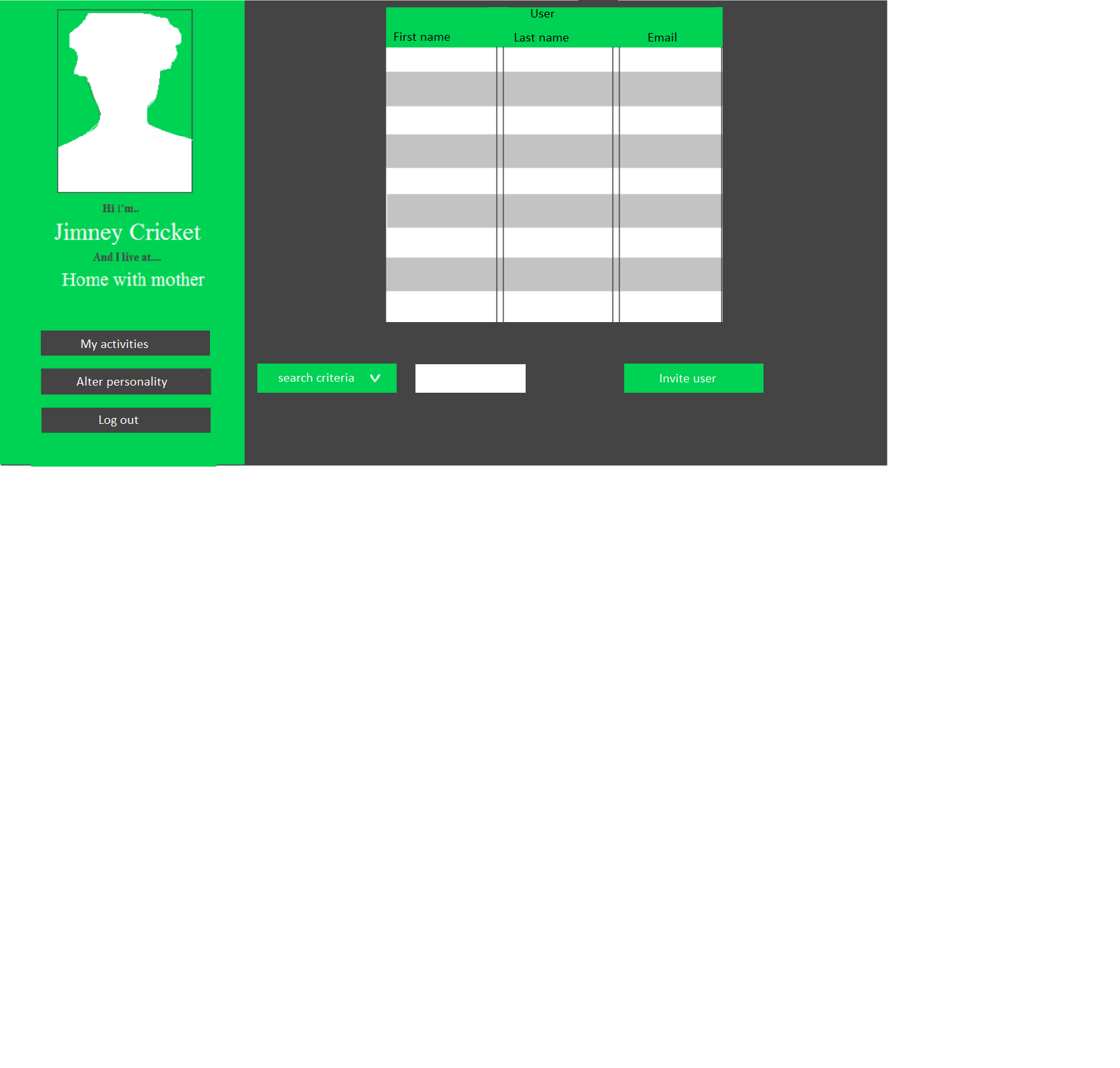


Figure 3‑15: Draft of invite user page

## Inter-Module Data Communication

The data communication flow between the different modules in the system is illustrated in *Figure 3‑1: Communication between modules.*

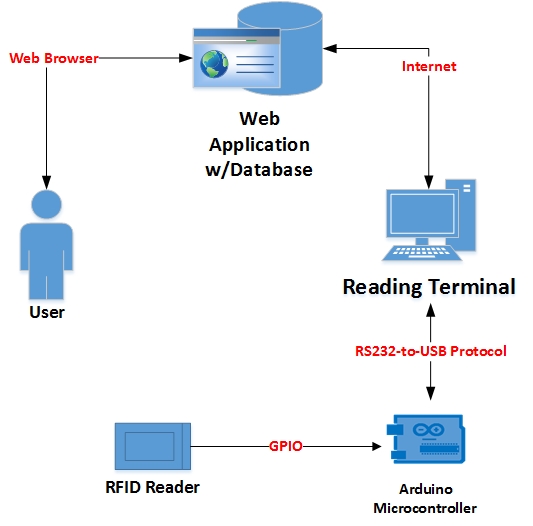


Figure 3‑1: Communication between modules

The main communication entry point for the user will be with the web application through a web browser. The reading terminal will communicate with the database server through an internet connection, allowing it to insert and extract the necessary data. The Arduino microcontroller communicates with the reading terminal though a RS232-to-USB cable and a custom transfer protocol made specifically for this application. Finally, the RFID reader communicates with the Arduino microcontroller through the microcontrollers GPIO pins and associated protocol.

# System Architecture

The following chapter will give a brief explanation of the general system architecture of the CheckPoint Attendance System.

## Three-Tier Multi-layer Structure

The CheckPoint system will adhere to a 3-tier structure. This involves different parts of the application running as separate entities either physically or logically in different locations. In the case of the CheckPoint system, this will mean having a separate database server, web application server and browser. The primary aim of this structure is to provide flexibility and easier maintenance since each layer may be modified without it being necessary to make changes throughout the entire application.

Moreover, a layered structure provides an easier pathway to scalability and performance since it is possible to dedicate extra resources only where they are needed instead of globally. If more servers were required to handle increasing web requests this could be achieved without adding extra servers to the database or web-server application layers. *Figure 4‑1: 3-Tier System Diagram* below shows the basic layout of a 3-tier system.



Figure ‑: 3-Tier System Diagram

### Presentation Layer

The top tier is often referred to as the presentation layer. This layer is the point of interaction between the system and the client and includes the UI of the application. The main role of the presentation layer is to provide a convenient user-friendly way to use the application and access the functionality that the system affords. The presentation layer of the Checkpoint system will primarily consist of an ASP.NET Webform application. This will be delivered to a browser such as Google Chrome or Mozilla Firefox running on a client’s individual desktop PC or laptop. Additionally, the Checkpoint system will includes an RFID reading terminal comprising a NUC, Arduino-UNO and an RFID scanner. This part of the system will be a point of interaction at the location of all appointments. The terminals will facilitate scanning the attendees RFID tags and will provide feedback on status of the scanning procedure.

### Business Layer

The business layer is responsible for the main logic of the application. Data handling, calculations, processing requests and co-ordination of the application are all tasks performed by the business layer. It is effectively the brain of the system translating the user requests into actions and defining what operations can and cannot be performed on the data stored in the system. The CheckPoint business layer will be a C# .NET program capable of interacting with the ASP.NET Webform presentation layer and the database in the data access layer.

### Data Access Layer

The Data Access Layer describes the part of the application that communicates with the persistent storage location, usually a database server. It contains the services capable of collecting and transferring data between the storage facility and delivering them to the business layer. All interaction with the database should take place in this layer such that the other layers are unaware of the type of database or storage system being used. SQL Management Server will be the data access layer of the CheckPoint system. *Figure 4‑2: CheckPoint Architecture Overview* beneath shows a basic graphical overview of the CheckPoint system architecture.



Figure ‑: CheckPoint Architecture Overview

# Database

The database in the CheckPoint Attendance System is a Microsoft Azure SQL database and is an intricate part of the functionality of the software. This section explains the database structure and its tables and attributes. The database was modelled using the ERwin database-modelling tool.

## Logical and Physical Data Model

The database can be represented in two states; the abstract logical state and the physical “as implemented” state. These can both be seen in *Figure 5‑1: Database logical data model* and *Figure 5‑2: Database physical data model* respectively. A detailed explanation of the tables and attributes can be found in *5.2 Database Description*.

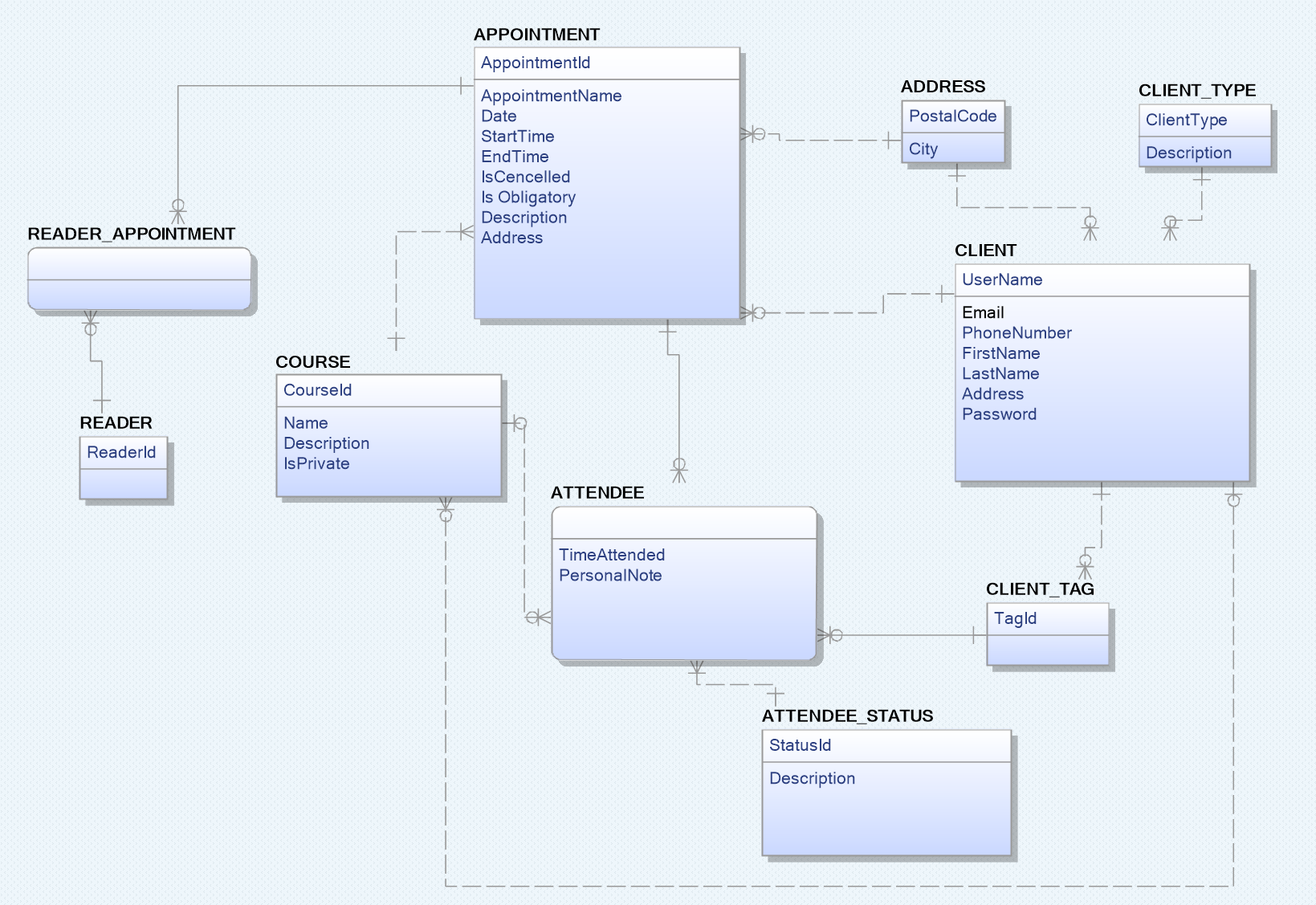


Figure ‑: Database logical data model

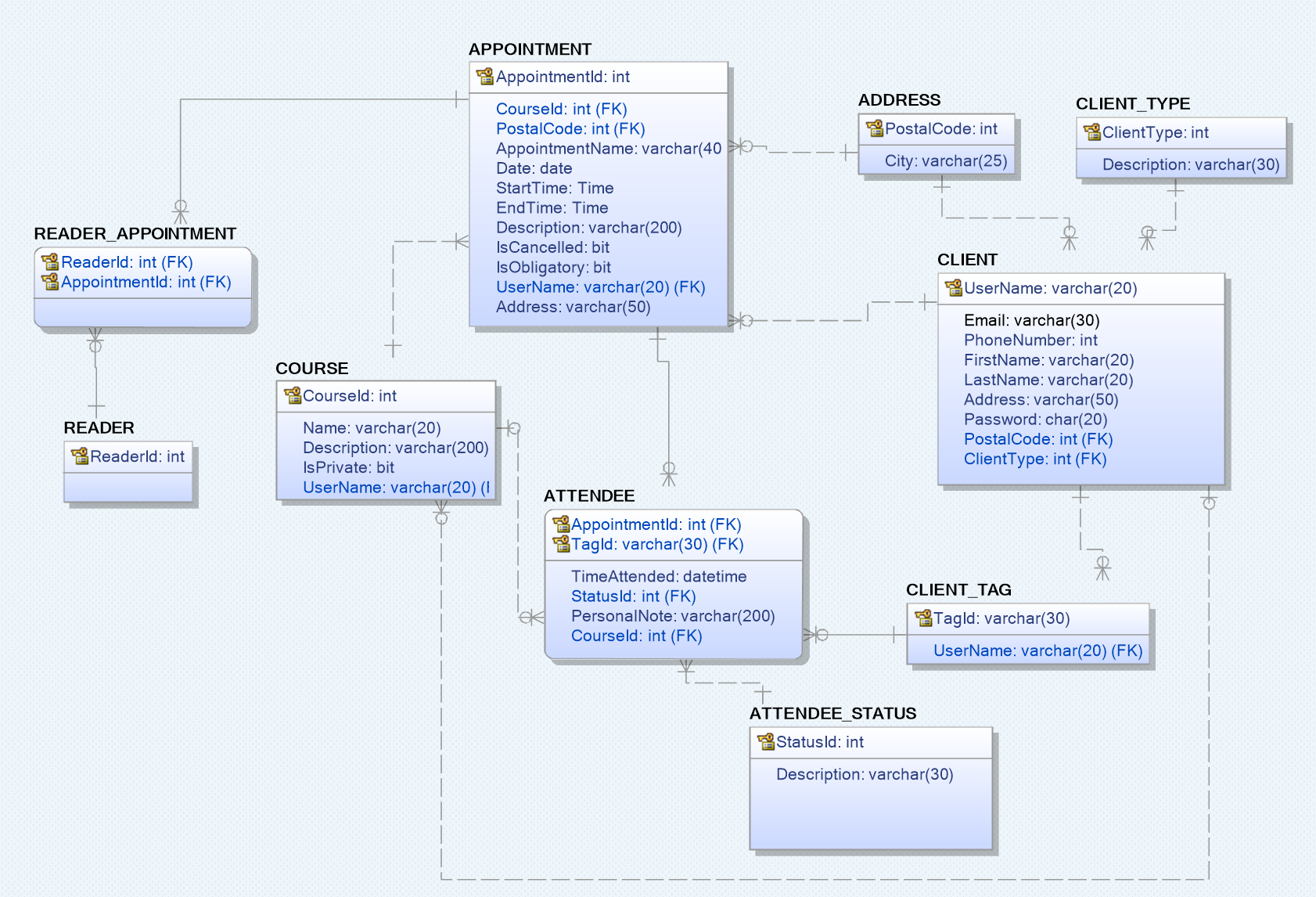


Figure ‑: Database physical data model

## Database Description

This gives a more detailed breakdown of the database structure by explaining the tables and attributes, as well as their relationships, keys and constraints found in the physical data model illustrated in ***Error! Reference source not found.****.* Primary keys are marked with an underscore and foreign keys are marked with an asterisk, as per standard notation.

### APPOINTMENT Table

The appointment table handles all data related to an appointment.

* **AppointmentId** identifies an appointment and is automatically incremented for each newly created appointment.
* **CourseId\*** references the COURSE table.
* **PostalCode\*** references the ADDRESS table.
* **AppointmentName** allows the host to give the appointment a name.
* **Date** holds the date of the appointment.
* **StartTime** holds the time when the appointment starts.
* **EndTime** holds the time when the appointment ends.
* **Description** allows the host to add additional information about his/her appointment.
* **IsCancelled** is used to check whether the appointment is active or has been completed/cancelled.
* **IsObligatory** is used to set a default obligatory or not obligatory status when a host is adding an attendee to an appointment.
* **UserName\*** references the CLIENT table.
* **Address** allows the host to add a location for the appointment.

### CLIENT Table

This table holds the data associated with a client.

* **UserName** identifies a client and is also used by the client to log into his/her account.
* **Email** holds the client’s email address.
* **PhoneNumber** holds the client’s phone number.
* **FirstName** holds the client’s first name.
* **LastName** holds the client’s last name.
* **Address** holds the client’s address.
* **Password** holds the client’s password for accessing his/her account.
* **PostalCode\*** references the ADDRESS table.
* **ClientType\*** references the CLIENT\_TYPE table.

### CLIENT\_TYPE Table

This table holds the data of the different possible client types.

* **ClientType** dictates the type of client with integer values witch are mapped in the following order:
  + **0**: User
  + **1**: Host
* **Description** holds information regarding the different types.

### CLIENT\_TAG Table

This table holds a client’s tag id.

* **TagId** holds a RFID tag number.
* **UserName\*** references the CLIENT table.

### ATTENDEE\_STATUS Table

This table holds an Attendee’s current status and keeps track of his/her current status.

* **StatusId** dictates the user/attendee’s current status with integer values which are mapped in the following order:
  + **0** : User has requested to attend an appointment and is awaiting approval by the host of the appointment.
  + **1**: The user’s request has been approved by the host, and the user is now an attendee. **The appointment has been marked by the host as NOT obligatory.**
  + **2:** The user’s request has been approved by the host, and the user is now an attendee. **The appointment has been marked by the host as obligatory.**
  + **3**: The user has attended the **NON-OBLIGATORY** appointment.
  + **4**: The user has attended the **OBLIGATORY** appointment.
  + **5**: The user **has been Invited to a private** **appointment.**
* **Description** holds information regarding the different states.

### ATTENDEE Table

This junction table holds the data associated with an attendee.

* **AppointmentId\*** references the APPOINTMENT table.
* **TagId\*** references the CLIENT\_TAG table.
* **TimeAttended** holds the information on when the attendee registered his attendance at the appointment.
* **PersonalNote** allows the attendee to append notes to the appointments he/she is attending.
* **StatusId\*** references the ATTENDEE\_STATUS table.
* **CourseId\*** references the COURSE table.

### COURSE Table

This table allows the host to establish courses associated with appointments, which allows for grouping of appointments.

* **CourseId** identifies the specific course.
* **Name** allows the host to give the course a name.
* **Description** allows the host to add a description to the course.
* **IsPrivate** is used to check whether the host has made the course public or private.
* **\*UserName** references the CLIENT table.

### READER Table

This table holds reference to all the RFID reading terminals.

* **ReaderId** identifies a specific reading terminal.

### READER\_APPOINTMENT Table

This junction table allows more than one RFID reader to be associated with a single appointment and keeps track of which reading terminal is deployed to which appointment.

* **ReaderId\*** references the READER table.
* **AppointmentId\*** references the APPOINTMENT table.

### ADDRESS Table

This table facilitates normalization since a city can be associated with several postal codes.

* **PostalCode** holds all the postal codes registered in Norway.
* **City** holds the city associated with the specific postal code.

# RESTful service

The Checkpoint attendance system consist of two separate modules: the web application and the reading terminal. The web application resides on an azure server and therefore maintains a secure server-to-server communication, while the reading terminal serves as a public device and becomes reliant on client-to-server communication. This can become problematic if the database it needs to access for information lies behind a strict firewall. In addition, it introduces potential security risks and might expose database connection information on the device itself.

The solution to this problem is a RESTful service capable of controlling and restricting the public device’s access to the main database server.

## CheckPoint REST API

The Checkpoint REST API is built in ASP.net with an MVC structure for URI routing. An overview of the service’s data flow can be seen in *Figure 6‑1: RESTful Service flow* below.

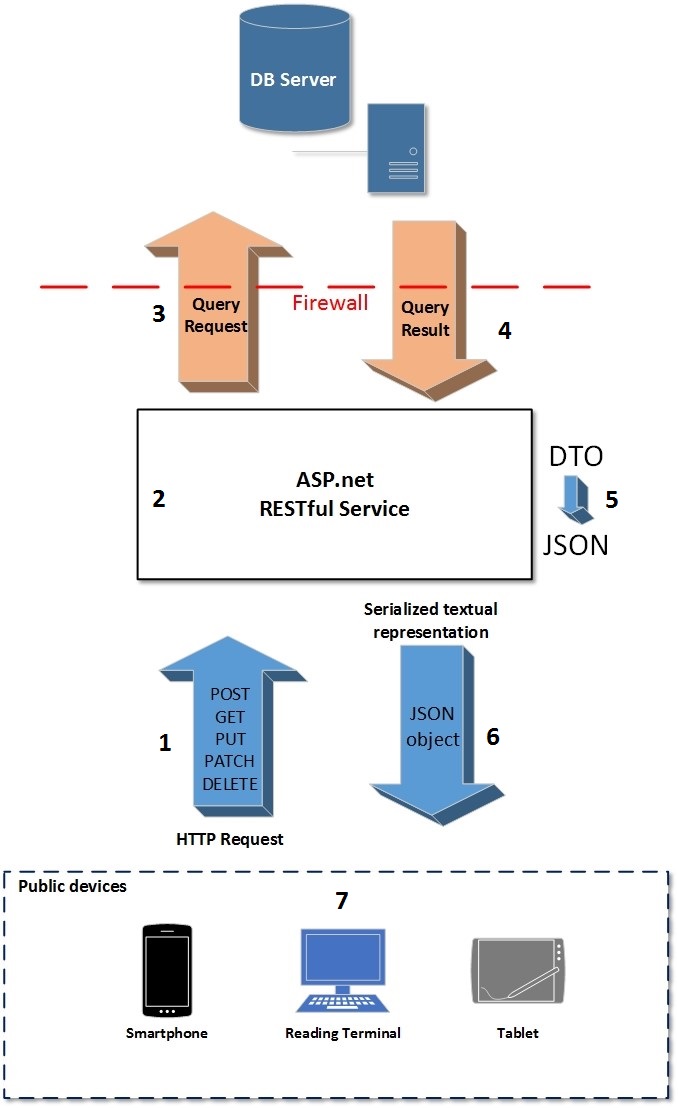


Figure 6‑1: RESTful Service flow

The following explains what happens through the RESTful service flow in *Figure 6‑1: RESTful Service flow*:

1. An HTTP request is made to the RESTful service from the public device. It contains the request method, request-URI for the resource upon which to apply the request and an access token for validation in its request header field.
2. The RESTful service reroutes the request to the appropriate action via its controller, based on the request-URI and the request method.
3. The corresponding query request to the database for the resource is made through a repository, queued and controlled by a unit of work in order to avoid concurrency issues.
4. The data returned in the query result is converted into a DTO. This allows the data returned by the HTTP response to potentially be filtered.
5. This DTO is serialized into a JSON, a textual representation of the data, within the RESTful service.
6. The HTTP response is returned to the public device, containing the filetype, here JSON, in its header and the actual data in its body.
7. The JSON is then deserialized by the public device API and used as intended.

### Clients and Roles

The RESTful service defines the access level of its clients through their assigned roles. This allows the API to be consumed by multiple different clients and devices, and even allows safe transactions to and from the database for third-party applications. The CheckPoint Restful service is configured with the clients and roles illustrated in *Table 6‑1: CheckPoint RESTful service clients and roles.* The “Role Definition” column represents the database resource in TABLE.attribute form.

Table 6‑1: CheckPoint RESTful service clients and roles

|  |  |  |
| --- | --- | --- |
| **Client** | **Role** | **Role definiton** |
| reading\_terminal | reading\_terminal\_write | Grants PATCH access to:  ATTENDEE.AttendeeStatus  ATTENDEE.TimeAttended |
| reading\_terminal\_read | Grants GET access to:  CLIENT.Username CLIENT.Password CLIENT.ClientType ATTENDEE.AttendeeStatus ATTENDEE.TimeAttended ATTENDEE.AppointmentId ATTENDEE.TagId APPOINTMENT.AppointmentId APPOINTMENT.AppointmentName APPOINTMENT.Date APPOINTMENT.StartTime APPOINTMENT.EndTime APPOINTMENT.IsObligatory |

As illustrated above, the reading terminal only has access to PATCH requests on two attributes in the Attendee table. Nothing else in the database can be altered from a device with the reading\_terminal client definition. This also allows the system to expand to other public devices in the future. If a mobile application for viewing appointments and courses an attendee has applied for was added as a system module, a “mobile” client could be added with the “mobile\_read” role that only exposed GET access to appointments and courses.

The CheckPoint RESTful API is deployed on its own separate azure server.

# Reading Terminal

In order to register attendance in the CheckPoint Attendance System, a reading terminal is required. This device will be responsible for verifying that the attendee is accepted for the appointment and log the date and time of his attendance.

There are several RFID reading modules available on the market, but these are restricted by the transfer protocols embedded in the device and may also have restrictions regarding linking several devices together in a network.

Because of these restrictions, it was decided to use an Arduino microcontroller with a separate RFID reading module and a custom transfer protocol.

## Hardware Configuration and Circuit Diagram

The RFID reading device consists of an Arduino UNO microcontroller and an MFRC522 RFID reader configured according to *Figure 7‑1: Reading Terminal Circuit Diagram*.

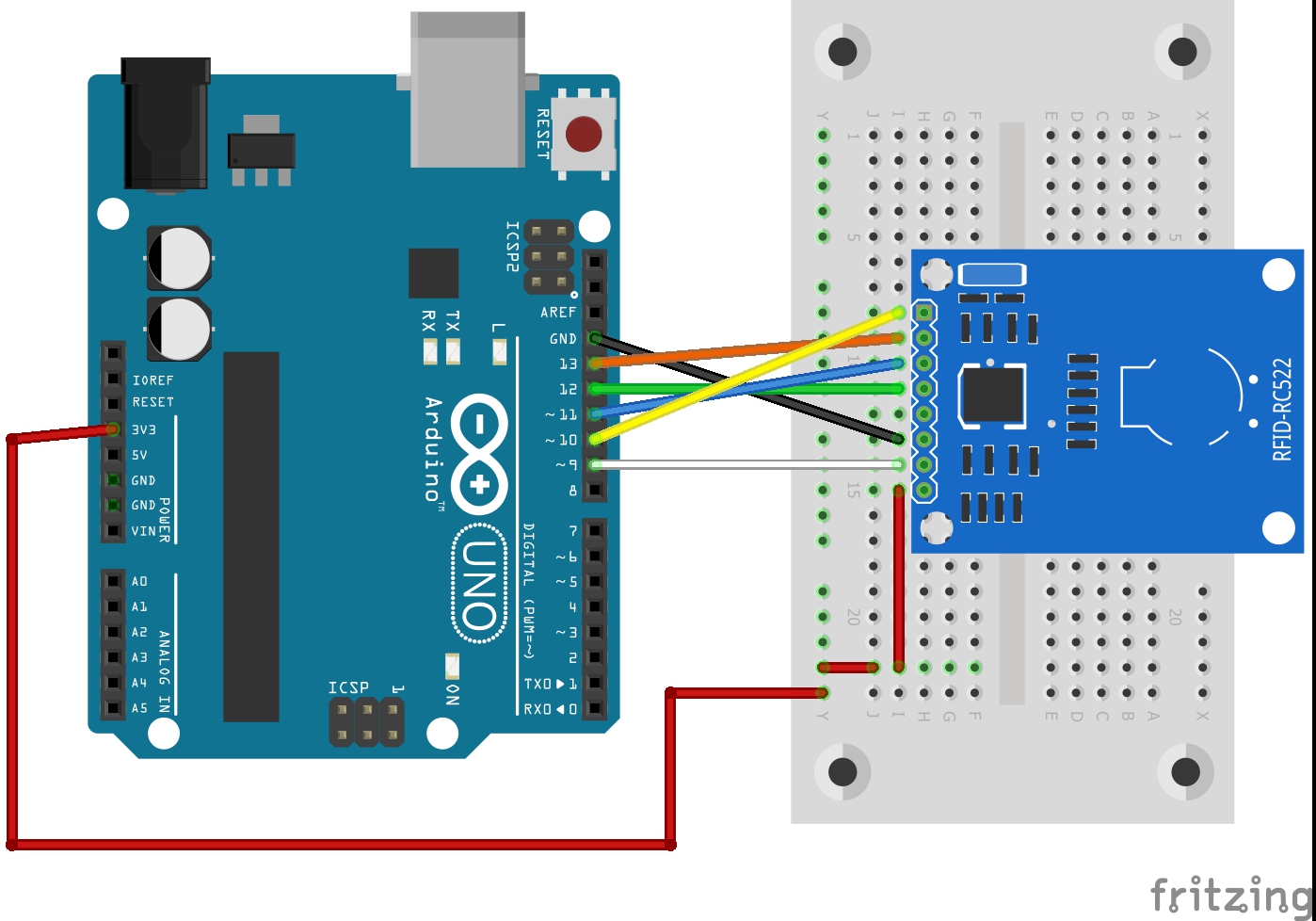


Figure ‑: Reading Terminal Circuit Diagram

## Transfer Protocol

The default transfer of data between an Arduino and a computer involves sending data as a «string» type by using “Serial.Print”. This is not very efficient as it creates a lot of overhead which results in more bytes to transfer for the same amount of information.

In order to avoid an unnecessary amount of bytes being transferred and secure consistent transfer of data from the RFID reading device, a custom transfer protocol was created for the RFID reading device.

# UML

This section contains the UML documentation used in the design and implementation of the software. It provides both an overview of the general functionality of the software in the use case diagram, but also provides a greater level of detail in the use case documents and class diagrams. Because of the agile nature of the development process, these documents are dynamic and susceptible to change due to changing functionality requirements.

## Use Case Diagram

The use case diagram represents an overview of the main functionality of the CheckPoint Attendance System described in *3.1 Functional Requirements* and can be viewed in *Figure 8‑1: Main use case diagram for CheckPoint Attendance System*.

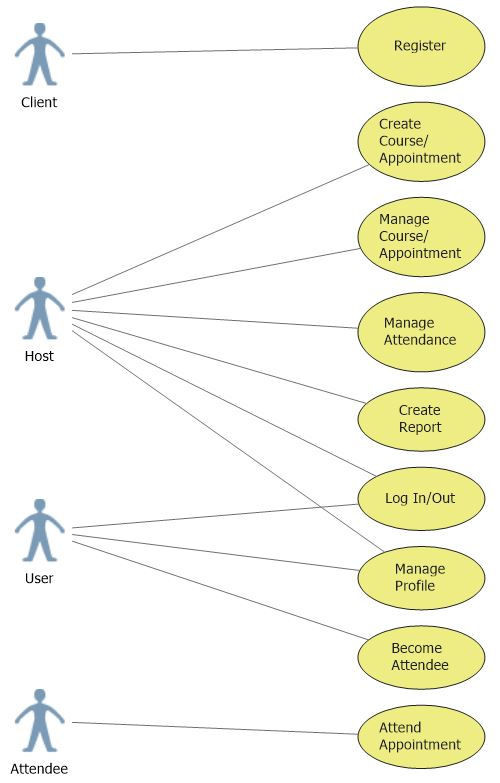


Figure 8‑1: Main use case diagram for CheckPoint Attendance System

## Use Case Documents

The use case documents provide a more detailed explanation of the different use cases found in *8.1 Use Case Diagram*.

### Register

**Scope:** CheckPoint Attendance System; Web application module

**Goal:** Register a new user or host and create an associated account.

**Brief:** A client accesses the CheckPoint Attendance System web page through a browser and wants to become a user or host by creating an account. He enters his account information and becomes a user or host. He is then redirected to his personal homepage.

**Primary actor:** Client

**Stakeholders and Interests:**

Client - wants to become either a registered user or host and gain access to the system’s functionality.

**Preconditions:** CheckPoint Attendance System webpage is loaded and showing in browser.

**Success Guarantees:** The client successfully registers and creates a user or host account, and is redirected to his homepage.

**Main success scenario:**

1. The client clicks the button for registration on the homepage and is redirected to the registration page.
2. The system prompts the client for new account information:

Username

Password

First name

Last name

Email

Phone number

Address

Postcode

1. Client enters new account information
2. System validates new account information
3. The new user/host is created and populated with the account information.
4. The user/host is added to the database.
5. The new user/host is logged in.
6. The new user/host is redirected to his homepage.

**Extensions:**

4.a Username already exists

1. System displays an appropriate error message.
2. The use case continues at step 2.

4.b Client details are in the wrong format

1. System displays an appropriate error message.
2. The use case continues at step 2.

6.a Unable to connect to the database

1. System displays an appropriate error message,
2. The use case continuous at step 2.

**Frequency of occurrence:** Every time a client attempts to register a new account.

### Log in

**Scope:** CheckPoint Attendance System; Web application module

**Goal:** The user or host logs in to his personal homepage.

**Brief:** A user or host accesses the login web page through a browser, enters his/her username and password and is redirected to his/her personal home page.

**Primary actor:** User, Host

**Stakeholders and Interests:**

User – Wants to gain access to his homepage in order to view his current appointments, look for new appointments, apply to appointments and change his personal details.

Host – Wants to gain access to his homepage in order to view attendance for the appointments he/she is hosting, make new appointments, accept applied users as attendees and change his personal details.

**Preconditions:** Login webpage is loaded in a browser.

**Success Guarantees:** The client successfully logs in and is redirected to his/her personal homepage.

**Main success scenario:**

1. The client clicks the button for logging in on the homepage and is redirected to the log in page.
2. The system prompts the client for account information:

Username

Password

1. Client enters account information
2. System validates account information
3. The user/host is logged in.
4. The user/host is redirected to his homepage.

**Extensions:**

4.a Account does not exist

1. System displays an appropriate error message.
2. The use case continues at step 2.

4.b Account information in the wrong format

1. System displays an appropriate error message.
2. The use case continues at step 2.

**Frequency of occurrence:** Everytime a user or host attempts to log in.

### Become Attendee

**Scope:** CheckPoint Attendance System; Web application module

**Goal:** The user has requested to attend a course or an appointment.

**Brief:** A user searches for courses or appointments on his homepage and finds a course or an appointment he would like to attend. He selects the appointment and requests to be an attendee for it.

**Primary actor:** User

**Stakeholders and Interests:**

User – Wants to attend a course or one or several appointments.

**Preconditions:** User is logged in and on the “Find courses/appointments” user tab on his homepage.

**Success Guarantees:** The user selects a course or appointment and requests to attend, becoming an attendee for that course/appointment.

**Main success scenario:**

1. User clicks the “Search for courses/appointments” button.
2. System displays available public courses and appointments.
3. User selects the course or appointment he wishes to request attendance for.
4. Course/appointment details are shown in a panel below the gridview.
5. User clicks the button for requesting attendance.

**Extensions:** N/A

**Frequency of occurrence:** Everytime a user requests to attend a course or an appointment.

### Manage Profile

**Scope:** CheckPoint Attendance System; Web application module

**Goal:** The user or host wants to change the personal information related to his account.

**Brief:** A user or host wants to edit his personal details and accesses his

**Primary actor:** User, Host

**Stakeholders and Interests:**

User – Wants to edit/update the personal details linked to his account.

Host – Wants to edit/update the personal details linked to his account.

**Preconditions:** User/host is logged in and on the “Personal Details” tab.

**Success Guarantees:** The new user details are saved and displayed on the “Personal Details” tab.

**Main success scenario:**

1. The system prompts the user/host to change account information, with the current information displayed in the respective field:

First name

Last name

Password

Email

Phone number

Address

Postal code

1. User/host changes the desired fields.
2. User/host clicks button to save changes.
3. System validates new account information
4. User/host is redirected to his personal homepage.

**Extensions:**

2.a. User/host leaves “Personal Details” tab before saving changes

1. The old account information is kept and no changes are made
2. User/host redirected to other page.

4.a. New account information is in incorrect format

1. System displays an appropriate error message
2. The field(s) with incorrect format is highlighted.
3. The use case continues at step 1.

**Frequency of occurrence:** Everytime a user or host opens the “Personal Details” tab.

### Create Appointment

**Scope:** CheckPoint Attendance System; Web application module

**Goal:** Create a new appointment as a host.

**Brief:** A host fills in the required details for the appointment and creates it. He can choose to have it publicly available to all users or private.

**Primary actor:** Host

**Stakeholders and Interests:**

Host – Wants to create a new appointment.

**Preconditions:** Host is logged in and on the “Create Appointment” tab.

**Success Guarantees:** A new appointment is created.

**Main success scenario:**

1. The system prompts the host to enter new appointment information:

Appointment name

Address

Date

Start time

End time

Description

Postal Code

Course Id

Availability

1. Host enters new appointment information
2. Host clicks button to create appointment.
3. System validates new appointment information
4. New appointment is created and saved.
5. Host is redirected to “My Appointments” tab.

**Extensions:**

2.a. Host leaves “Create Appointment” tab before creating appointment

1. The fields are cleared.
2. User/host redirected to other page.

4.b. Course Id does not exist.

1. System enquires host if he wants to create a new course with the provided course id.
   1. Host wishes to create a new course
      1. A new course is created with the provided Course Id field value and the appointment is added to the course.
   2. Host does not wish to create a new course
      1. Course Id field remains empty.
2. Use case continues at step 3.

**Frequency of occurrence:** Everytime a host loads the “Create Appointment” tab.

### Manage Appointment

**Scope:** CheckPoint Attendance System; Web application module

**Goal:** Update/edit an appointment’s information.

**Brief:** A host chooses one of his created appointments and updates/edits its information. The changes are saved and the host and the appointment’s attendees can see the new information on the “My Appointments” tab.

**Primary actor:** Host

**Stakeholders and Interests:**

Host – Wants to edit/update the personal details linked to his account.

Attendee – Wants to know what information has been changed about the original appointment.

**Preconditions:** Host is logged in and on the “My Appointments” tab.

**Success Guarantees:** The new appointment information is saved and displayed to the host and all it’s attendees.

**Main success scenario:**

1. Host clicks on the appointment he wants to update/edit from the gridview.
2. Appointment information is shown below the gridview.
3. Host clicks the “Edit appointment” button.
4. The system prompts the host to change appointment information, with the current information displayed in the respective fields:

Appointment name

Address

Date

Start time

End time

Description

Postal Code

Course Id

Availability

1. Host updates the desired fields.
2. Host clicks button to save changes.
3. System validates new account information
4. Appointments’ information is saved to the database.
5. Host is redirected to his personal homepage.

**Extensions:**

5.a. Host leaves “My Appointments” tab before saving changes

1. The old appointment information is kept and no changes are made
2. User/host redirected to other page.

7.a. New appointment information is in incorrect format

1. System displays an appropriate error message
2. The field(s) with incorrect format is highlighted.
3. The use case continues at step 1.

8.a. Unable to connect to the database

1. System displays appropriate error message.
2. Use case continues at step 4.

**Frequency of occurrence:** Everytime a host wants to edit an appointment.

### Create Course

**Scope:** CheckPoint Attendance System; Web application module

**Goal:** Create a collection of appointments and group them into a course.

**Brief:** A host wants to create a course which consists of several appointments on different dates and times. He creates a new course and adds the number of appointments he needs and sets each individual appointment’s information.

**Primary actor:** Host

**Stakeholders and Interests:**

Host – Wants to group his appointments to ensure that attendees are signed up for all of the appointments.

User – Wants to sign up for all the related appointments in a course, in order to make sure they

**Preconditions:** Host is logged in and on the page for creating courses.

**Success Guarantees:** The desired number of appointments are added to the course, and the course is saved to the database.

**Main success scenario:**

1. System prompts host for course information:

Course name

1. Host adds the desired number of appointments to the course.
2. Host enters new appointment information for each added appointment
3. System validates course and appointment information.
4. Host clicks the button for creating a new course.
5. New course and appointments are created and saved to the database.
6. Host is redirected to his personal homepage.

**Extensions:**

3.a. Host leaves the page for creating courses before saving changes

1. All information is cleared and added appointments are deleted.
2. User/host redirected to other page.

4.a. New course and/or appointment information is in incorrect format

1. System displays an appropriate error message
2. The field(s) with incorrect format is highlighted.
3. The use case continues at step 3.

6.a. Unable to connect to the database

1. System displays appropriate error message.
2. Use case continues at step 1.

**Frequency of occurrence:** Everytime a host wants create a course.

### Manage Course

**Scope:** CheckPoint Attendance System; Web application module

**Goal:** Edit one or more appointments belonging to a course.

**Brief:** A host wants to edit/update one or more appointments group in a course. He chooses the specific appointments, updates their information and saves the changes. These changes now are visible to all attendees and users.

**Primary actor:** Host

**Stakeholders and Interests:**

Host – Wants to edit/update one or more appointments he administers in a course.

Attendee – Wants to know about changes being made to appointments he is signed up for.

**Preconditions:** Host is logged in and on the page for managing courses.

**Success Guarantees:** The desired number of appointments’ information is updated and the changes are saved. Users and attendees can view the changes that has been made.

**Main success scenario:**

1. Host chooses the appointment(s) he wishes to update.
2. The system prompts the host to change appointment information, with the current information displayed in the respective fields:

Appointment name

Address

Date

Start time

End time

Description

Postal Code

Course Id

Availability

1. Host enters new appointment information for each added appointment
2. System validates course and appointment information.
3. Host clicks the button for creating a new course.
4. New course and appointments are created and saved to the database.
5. Host is redirected to his personal homepage.

**Extensions:**

3.a. Host leaves the page for managing courses before saving changes

1. No information is updated/changed.
2. User/host redirected to other page.

4.a. Updated course and/or appointment information is in incorrect format

1. System displays an appropriate error message
2. The field(s) with incorrect format is highlighted.
3. The use case continues at step 3.

6.a. Unable to connect to the database

1. System displays appropriate error message.
2. Use case continues at step 1.

**Frequency of occurrence:** Everytime a host update/edit a course.

### Manage Attendance

**Scope:** CheckPoint Attendance System; Web application module

**Goal:** Accept or deny attendee requests for courses/appointments

**Brief:** A host wants to decide which attendees to accept to his courses and appointments. He selects the attendee requests for his courses and appointments and either accepts or rejects them.

**Primary actor:** Host

**Stakeholders and Interests:**

Host – Wants to filter the attendance requests for his courses and appointments.

Attendee – Wants feedback on whether his attendee request was accepted or not.

**Preconditions:** Host is logged in and on the manage attendance page

**Success Guarantees:** Host selects the attendees who have applied to his courses and appointments and either accepts or rejects the requests.

**Main success scenario:**

1. Host selects the course or appointment with active attendee requests.
2. A list of the attendees with active attendee requests are shown in a gridview.
3. Host selects the attendee he wants to accept or reject
4. Attendee is accepted/rejected
5. Attendee is notified on request status on his own homepage.

**Extensions:**

**Frequency of occurrence:** Every time a host wants to manage attendee requests

### Attend Appointment

**Scope:** CheckPoint Attendance System; Reading Terminal module

**Goal:** Register and log an attendee’s attendance to the database.

**Brief:** An attendee swipes his RFID tag on reading terminal’s RFID reader and registers his attendance. The tag Id is transferred to the reading terminal and checked against the database. If the tag id exists in the database and is connected to the associated appointment, the attendance is logged with a timestamp.

**Primary actor:** Attendee

**Stakeholders and Interests:**

Attendee – Wants to register and log his attendance.

Host – Wants to gain an overview over who is attending his appointment.

**Preconditions:** User has received his RFID tag after registering. The user’s request to attend the appointment has been approved by the host, and the user is an attendee for the appointment.

**Success Guarantees:** The attendance has been registered and the database has been updated with the timestamp.

**Main success scenario:**

1. Reading terminal prompts attendee to swipe his RFID tag.
2. Attendee swipes his RFID tag on the reading terminal’s RFID reader.
3. Swipe is recognized by the reading terminal.
4. Attendee is found in the list of valid appointment attendees.
5. Attendee’s attendance is registered and logged to the database,
6. Reading terminal displays message to attendee confirming successful registration.

**Extensions:**

3.a. RFID reader fails to transfer tag id to the reading terminal

1. Reading terminal displays appropriate error message
2. Use case continues at step 1.

4.a. Attendee is not found in the list of valid appointment attendees.

1. Reading terminal displays appropriate error message.
2. Use case continues at step 1.

5.a Unable to connect to the database

1. Attendance information is serialized and stored locally on the reading terminal
2. Use case continues at step 6.

**Frequency of occurrence:** Everytime an attendee swipes his RFID tag to attend and appointment.

### Create Report

**Scope:** CheckPoint Attendance System; Web application module

**Goal:** Create a report with information about courses, appointments and attendance.

**Brief:** A host wants an overview of the attendance records for his courses and appointments. He chooses the courses and appointments he wants to include in his report, adds the desired filtering and gets a pdf report file.

**Primary actor:** Host

**Stakeholders and Interests:**

Host – Wants a report of attendance information related to his courses and appointments.

**Preconditions:** Host is logged in and on create report page.

**Success Guarantees:** The report is created and made available for the host to download as a pdf file.

**Main success scenario:**

1. Host chooses the courses and appointments he wants to include in the report
2. The system prompts the host for report filename.
3. Host enters report filename.
4. System prompts host for filtering options.
5. Host enters filtering options.
6. Host clicks the button for creating a new report.
7. System validates report filename.
8. New report is created and made available for downloading.
9. Host downloads the report.

**Extensions:**

1.a. Host does not add any courses or appointments to include in the report

1. System displays an appropriate error message
2. Use case continues at step 1.

5.a. Host does not choose any filtering options.

1. Report is filtered with the default settings.
2. The use case continues at step 6.

7.a. Filename is blank

1. System displays appropriate error message.
2. Use case continues at step 3.

8.a. Host created report but does not download it before leaving page.

1. Created report is deleted.
2. Host is redirected to the new page.

**Frequency of occurrence:** Every time a host loads the page for creating a report.

## Sequence Diagrams

The purpose of a sequence diagram is to understand the interactions between objects in the use case specific parts of the software. This provides an informative look at program flow and is a key part in the implementation.

The size of the sequence diagrams would render them unreadable in this document, and they have therefore been made available from the project homepage by going to <http://tfweb.hit.no/2017/Checkpoint>, navigating to the “Documents” tab and downloading the “CheckPoint Sequence Diagrams” file.

# Class Diagram and Structure Maps

Class diagrams are normally a well-established part of the UML documentation, and for the CheckPoint System this is no different. However, because of the fairly large amount of classes and the architectural complexity of the software, the class diagram has been assigned its own chapter. This chapter also explains several design choices and implementations for each of the systems three architectural layers presented in *4.1 Three-Tier Multi-layer Structure.*

## Class Diagram: High-Level Overview

The Class Diagram shown on the following page in *Figure 9‑1: High Level Class Diagram* represents a very general overview of the class structure of the CheckPoint application. The class blocks are contained within folders and these folders represent individual projects within a single solution, often but not always, a folder is indicative of a logical layer within the application.

The green blocks represent interfaces and the blue blocks are classes. The high-level view that this diagram presents is unable to communicate the specific functional relationships between the classes and so this will be covered in more detail in the subsequent subchapters. What this overview does make visible is the general organization of the application and the way in which interfaces have been used across the solution to enforce contractual agreements and assign interclass dependencies. Again, the precise nature of these contracts between “client” classes and “service” classes will be examined in the following subchapters.

Dashed lines represent the implementation of an interface by a class. Solid lines represent class inheritance. The larger arrowheads point to the classes being implemented or inherited. The smaller arrowheads indicate that a class is referenced.

The folders named “CheckPointViews” and “CheckPointPresenters” both belong to the presentation layer of the application. The “CheckPointModels” and “CheckPointDataTables” folders belong to the business layer of the application while the “”CheckPointDataAccess” folder resides in the data access layer. The “CheckPointInterfaces” and “CheckPointBootstrap” folders are crosscutting concerns that transverse each of the three logical layers.



Figure 9‑1: High Level Class Diagram

## Presentation Layer Class Diagram: Model View Presenter

The below class diagram, *Figure 9‑2: Presentation Layer Class Diagram,* is an overview of the basic structure of the CheckPoint presentation layer. This layer has been designed according to the Model View Presenter (MVP) architectural pattern.



Figure 9‑2: Presentation Layer Class Diagram

The primary goal of the MVP pattern is loose coupling between the user interface, the presentation logic, and the business logic. This is achieved by using methods of abstraction across a three-layer structure that resides mostly within the presentation layer of the application.

* The View layer contains classes that concern the user interface (UI).
* The Presenter layer contains the classes that act as a mediator between the View and the Model layers.
* The Model layer contains the classes that hold the business logic along with the data that is to be displayed in the View.

Loose coupling between the layers, especially the View and the Model, leads to more flexible, maintainable code and a more easily testable application. The use of interfaces is the foundation of the pattern providing a way for the Presenter layer to communicate with the other layers through abstraction instead of using concrete implementations of the classes and methods it needs to perform its tasks. The View remains completely unaware of the Model and the business logic. The View only knows about an abstract version of the Presenter.

Similarly, the Presenter is only aware of an abstract version of the Model and the Model is completely unaware of both the Presenter and the View. This separation of concerns is of great benefit when changes are made to the code. When a part of the code in one layer is changed, there will be minimum impact in the other layers. This reduces the amount of changes that will need to be made in the rest of the code. If a program is very simple, this might not make such a big difference. However, for any application that has some degree of complexity, failing to adequately separate concerns and handle dependencies can quickly result in code that is very difficult or sometimes even impossible to maintain.

There are several variants of the MVP pattern, the two most popular being known as “Supervisory Presenter” and “Passive View”. CheckPoint is structured around the Passive View type whereby the View is reduced to being as inactive as possible with all display related logic moved into the Presenter class. The Presenter is responsible for controlling what the View should display and how it should be displayed. The View is prohibited from direct communication with the Presenter. Any interaction is performed indirectly using events published by the View. To ensure maximum flexibility it is important to make sure that the Presenter only sends primitive data types to the View rather than platform specific types.

The following code snippets will attempt to explain the class relationships of the CheckPoint application found in the presentation layer. The focus will be how interfaces have been used to achieve the desired de-coupling.

We begin with a very basic interface that all of our views will implement and extend as shown in *Figure 9‑3: IView Interface.* Any properties or methods in this interface will be available globally to all Views of the application. In the code snippet below a single property is described which simply returns the current HTTP context URL of the ASP.NET Webform.

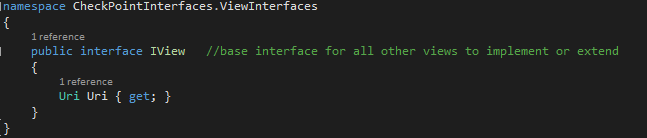


Figure 9‑3: IView Interface

In a similar way, we define an abstract base class for all of our Presenters to inherit as shown in *Figure 9‑4: Base Presenter Class.* Inside the abstract class, we define three virtual methods that will be overridden by the concrete Presenter classes. These three methods are used to control the creation and implementation of a Presenter at the appropriate time during a Webform lifecycle. This is necessary due to the way that ASP.NET controls the loading of WebForm pages. These base classes assist us in reusing code and avoiding unnecessary duplication of common methods and attributes.

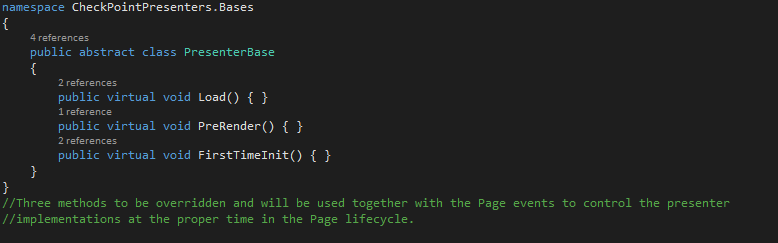


Figure 9‑4: Base Presenter Class

Next, we proceed to define an abstract base class for all of our views to inherit as shown in *Figure 9‑5: Base View Class*. Firstly, this base class implements our basic View interface. Secondly, this base class requires and accepts any generic Presenter class so long as that Presenter inherits form the base Presenter class that we defined. This base view class itself inherits from the “Page” class of an ASP.NET WebForm and in doing so lets us hook in to the important lifecycle events that take place when a Page is loaded. We override these events and use them to trigger a call to our base Presenters virtual methods.

Whenever we redirect to a View and it fires the “OnLoad ()” event, a Presenter will be created and the appropriate method will be called on the Presenter according to whether it is the first time that the View is loaded or a Postback. The specific details of the “CreatePresenter()” method we see below will be explained in a later chapter “SturctureMap Inversion of Control” and not shown here for the sake of simplicity. The important point is that we are able to call a method that can instantiate a Presenter at the correct time during a Page load.

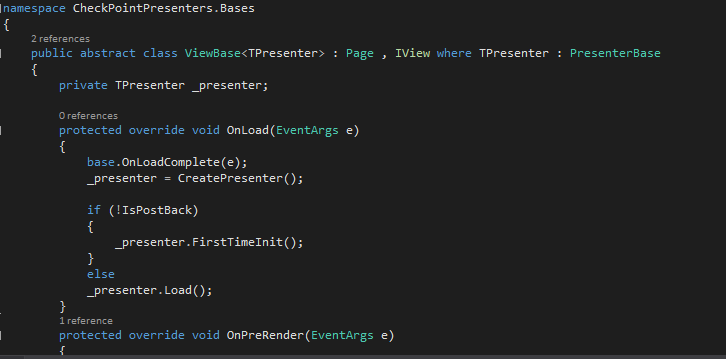


Figure 9‑5: Base View Class

Now the basic infrastructure is in place that we can begin to create our more specific View and Presenter classes.

An interface for a homepage is defined with two get and set string properties, as shown in Figure *Figure 9‑6: IHomeView Interface*. This is kept as simple as possible to provide an example of how data will be passed to the display.

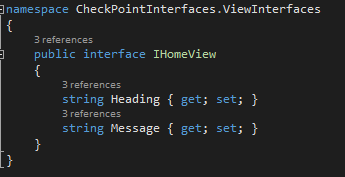


Figure 9‑6: IHomeView Interface

Next a concrete home View is defined as shown in *Figure 9‑7: Home View Class*. This View inherits from both the abstract base View and the home View interface. The abstract base View automatically endows the home View with all of the Page class attributes as well as the basic View interface attributes. The abstract class also obligates the home View to require a Presenter type and therefore exposes the important methods needed for a View to create its associated Presenter when it loads. The properties of the home View interface are now defined and assigned to two textboxes. These properties will be exposed to and controlled by the Presenter.

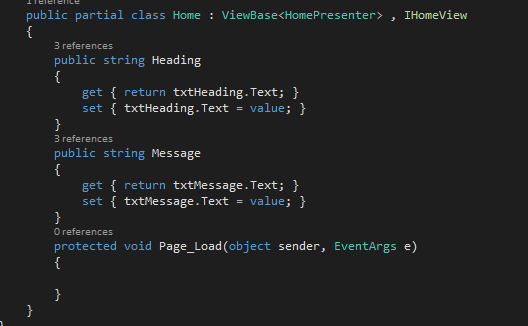
.

Figure 9‑7: Home View Class

A Model interface and its respective class is defined in the same way as the View as shown in *Figure 9‑8: IModel Interface* and *Figure 9‑9: Model Class*. Once more in the interest of simplicity, the Model has just two properties that reflect the data that the View will display. In reality, the Model will contain the objects and machinery that collects the required data from a persistent source and processes it before passing it to the Presenter. Here again the interface defines the function and the concrete class provides the detail of how the function will be accomplished.

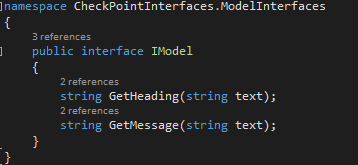


Figure ‑: IModel Interface

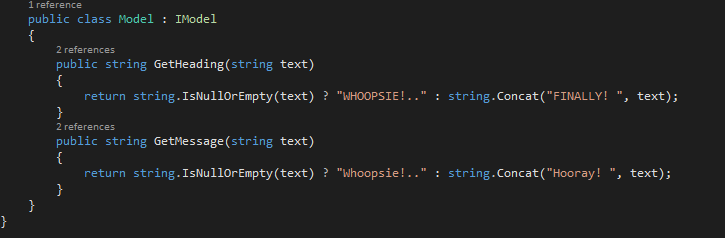


Figure 9‑9: Model Class

Finally, as shown in *Figure 9‑10: Home Presenter Class*, a home page Presenter class is defined with a constructor that accepts a View interface and a Model interface as arguments. This allows us to compose a Presenter that can access all the methods and properties of the View and Model without needing to instantiate the concrete View or Model. The Presenter does not depend on the details of the objects it will use. It depends only on the abstractions.

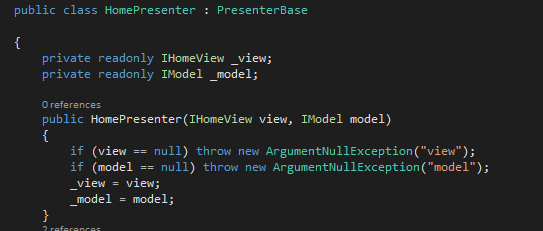


Figure 9‑10: Home Presenter Class

The presentation layer pattern is complete and the Presenter can now set the View properties with data from the Model when the View loads. *Figure 9‑11: Presenter Load Method* shows the Presenter class setting the ‘Heading’ and ‘Message’ properties of the View from within the ‘Load()’ method. The string data to set the ‘Heading’ is acquired from the Model using the ‘GetHeading()’ method.

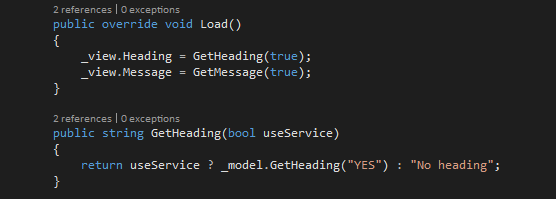


Figure 9‑11: Presenter Load Method

## CheckPoint Business Layer & Data Access Layer Class Diagram: Repository Pattern

The class diagram below, shown in *Figure 9‑12: Data Access and Business Layer Class Diagram*, represents an overview of the data access layer along with some specific classes from the business logic layer. Similar to the way that the CheckPoint presentation layer has been organized, the data access layer employs two patterns in order to achieve loose coupling between the core business logic and the data access layer



Figure 9‑12: Data Access and Business Layer Class Diagram

The patterns used are known as the “Repository” pattern and the “Unit of Work” pattern. These two patterns work in combination to separate the business layer from the data access layer and invert the direction of dependencies such that the data access layer depends on the business layer and not vice versa.

Often applications are written in a way that ties them to a specific vendor’s database, storage system or third party object relational mapping tools. Even if later it is desirable to change to another provider this can prove to be too difficult to do without rewriting major parts of the source code. The goal of these patterns is to make it possible to change the persistence layer if desired without a major rewrite. Once more, this is achieved by abstracting the functionality out to interfaces.

The following code snippets will attempt to show how the Repository pattern and Unit of Work pattern are implemented to communicate with a database. We begin with a completely generic interface that provides the most general queries common to all table queries as shown in *Figure 9‑13: Generic Repository Interface*. Every Repository class will inherit from this interface allowing us to avoid duplicating these basic methods throughout the code.

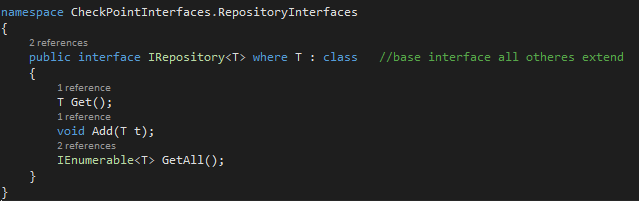


Figure 9‑13: Generic Repository Interface

Next, a generic Repository class is defined that inherits form the basic interface and accepts a “DbContext” object in its constructor as shown in *Figure 9‑14: Generic Base Repository Class*. The Entity framework DbContext object provides a way to access a database with a predefined connection string and make queries using convenient “LINQ” expressions. All Repository classes can extend this class through inheritance to provide specific functions for each table as desired.

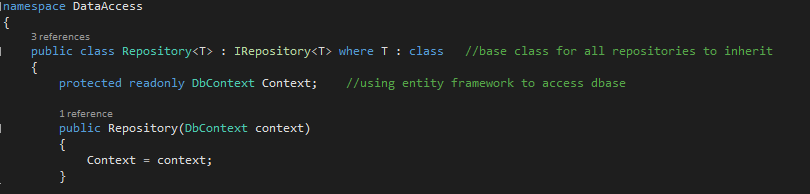


Figure 9‑14: Generic Base Repository Class

An interface is now defined for a specific type of table that will return “User” type object data from its queries as shown in *Figure 9‑15: IUser Repository Interface.* The “User” class is a model of the user table that exists in the database. This interface provides access to the Repository functions and methods through abstraction rather than needing a specific implementation to call upon or create inside a client class.

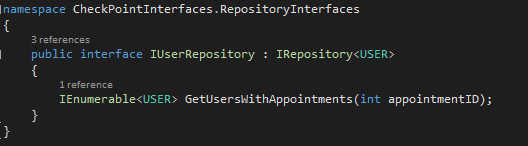


Figure 9‑15: IUser Repository Interface

A concrete specific “User” Repository is defined that describes the specific details of the functions that the Repository interface provides as shown in *Figure 9‑16: User Repository Class*. This is how the methods of the interface will actually be carried out yet the client class that calls the methods will not need to have any knowledge of this class itself – it will merely reference the “User” Repository interface instead. Using composition, we define the class constructor to accept a Context object.

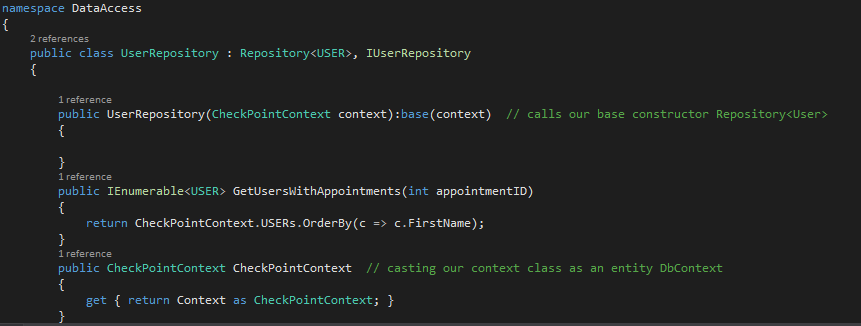


Figure 9‑16: User Repository Class

The “CheckPointContext” object, as shown in *Figure 9‑17: CheckPointContext Class*, is a class we define ourselves so that we have the ability to swap Entity framework with another ORM if that was desired or became necessary in the future.

CheckPointContext inherits form the Entity Framework DbContext class but it could easily be modified to inherit from another context object from another provider. The class properties are the object type representation of our database tables that we use to manipulate the data without directly accessing the database tables. Should we use another OR/M framework this, and the basic Repository class, is where the modifications would need to be made rather than throughout the source code. This isolates the framework dependency to the data access layer only.

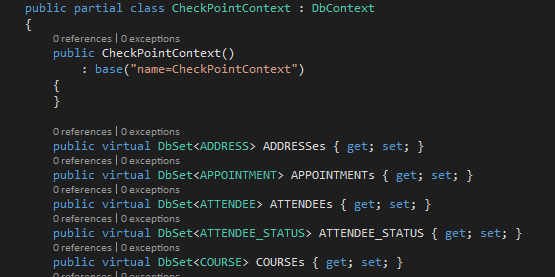


Figure 9‑17: CheckPointContext Class

Now two things are necessary to complete the pattern. A way to ensure that we do not break any rules of concurrency when saving changes to the database and a way to access the functionality provided by all of the different types of Repositories that we will create in one centralized place. For this, we will create a “Unit of Work” interface, as shown in *Figure 9‑18: IUnitOfWork Interface*, where we will add each type of specific Repository interface that we want to use as properties of the Unit of Work class.

A “Complete()” method will need to be called at the end of any type of query that will make changes to the data in the database. Since the Unit of Work will use a single DbContext object to carry out its work we are sure that any changes are performed as a single simple transaction. The Unit of Work keeps track of data that we manipulate as objects in the application and then handles the database update as a single transaction at the end during which the database will lock so that two users cannot make changes at the same time.

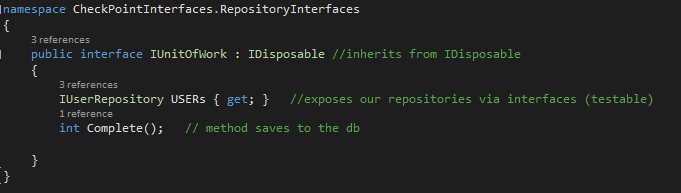


Figure 9‑18: IUnitOfWork Interface

A concrete Unit of Work class, as shown in *Figure 9‑19: UnitOfWork Class*, provides the details for the implementation and holds properties that allow access to any Repository methods. The constructor accepts our own Context object that inherits from DbContext. We are now able to call any Repositories and their methods through the Unit of Work interface rather than needing this concrete class to be instantiated. This de-couples the client from its service and lets us maintain clean separation between the layers of our architecture.

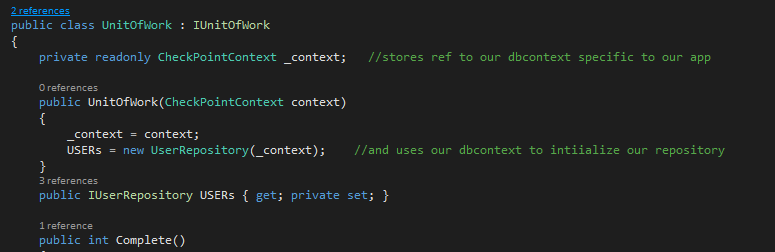


Figure 9‑19: UnitOfWork Class

It is now possible to inject our Unit of Work interface via the constructor into our Presenter and gain access to any Repository queries we wish to use using composition. The constructor injection is shown below in *Figure 9‑20: Presenter Constructor*.

An example of how the Presenter can call a method belonging to the ‘User’ Repository via the UnitOfWork is shown in *Figure 9‑21: Call to UnitOfWork Method*.

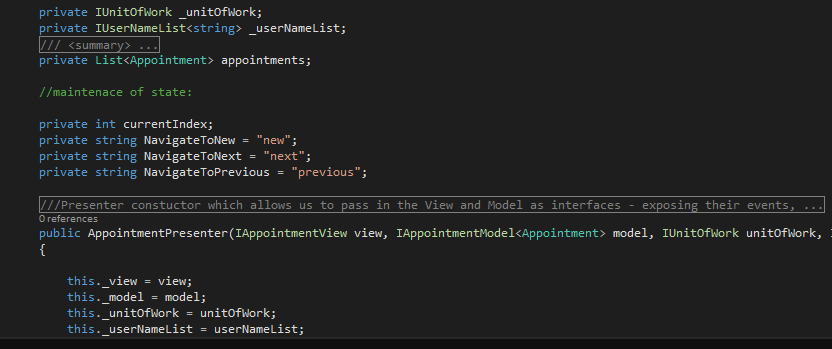


Figure ‑: Presenter Constructor



Figure 9‑21: Call to UnitOfWork Method

If the event that the OR/M or the type of database were swapped out, we would need to make a new concrete Repository classes to reflect the new Context objects or query syntax but we could avoid changing the code that exists in the core Business Logic layer of our application.

## StructureMap Inversion of Control: Inversion of Control Container



Figure 9‑22: Inversion of Control Class Diagram

The above class diagram, shown in *Figure 9‑22: Inversion of Control Class Diagram*, represents the Inversion of Control mechanism used by CheckPoint to perform the necessary dependency injections.

A vital part of the Checkpoint application architecture is the use of dependency injection. It is one of the main features of the Repository pattern, Model View Presenter pattern, and Unit of Work pattern. Indeed, anywhere where loose coupling is the goal it is highly likely that dependency injection and inversion of control will be employed to make it happen.

Dependency injection is nothing more than a technique for providing a class with its instance variables. If one class, the client, depends upon another class, the service, to accomplish some objective we say that the client depends on the service and the two classes are tightly coupled. If we seek to de-couple these classes then we use inversion of control to abstract the service’s functionality into an interface that the service will implement. This frees the client class from depending on the specific class and the details of how the function is performed. Instead, the client depends only on the interface abstraction. Using dependency injection, we can provide the client with any service which implements that interface. This is normally done by declaring the interface in the client class constructor but can be set as a property or a method too.

The injection of these interfaces is typically performed when the application first executes or during run-time by code outside of the main application. This is easily achieved in a Winforms solution by simply creating a new project from which we create all the objects and their dependencies and pass them into the main project when the project loads. Due to the nature of the ASP.NET Webforms Page Lifecycle, this is not so easy to do. A Webforms Page is renewed every time an event is fired or a redirection occurs. The developer has little control over this process. What is needed is a way to hook in to the Page loading events and a way to create the dependencies at the right time when a Page loads. This is where an Inversion of Control container like Structure Map is useful. StructureMap is a framework that can automatically perform the dependency injection. We begin by adding an entry to the Web.config file as shown below in *Figure 9‑23: Registering the Initializer in the Web.config file*.



Figure 9‑23: Registering the Initializer in the Web.config file

The Web.config command points to the directory of the class called “CheckPoint.Bootstrap.Initializer” which is located in the CheckPoint.Bootstrap folder.

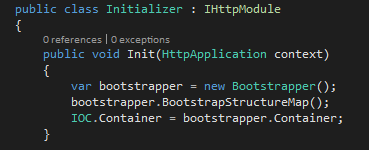


Figure 9‑24: Initializer Class

The Initializer class inherits from the IHttpModule class as shown in *Figure 9‑24: Initializer Class*. This causes the Initializer class to run “Init()” on application start up. A new Bootstrapper object is instantiated and the method “BootstrapStructureMap” is called. The Bootstrapper class is shown below in *Figure 9‑25: Bootstrapper Class*.

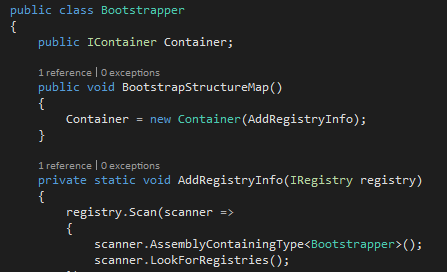


Figure 9‑25: Bootstrapper Class

This method creates a new container that will hold a registry of all the classes and objects that we need to satisfy our dependencies. StructureMap will scan through our projects designated directories looking for any classes that inherit from the StructureMap “Registry” class.

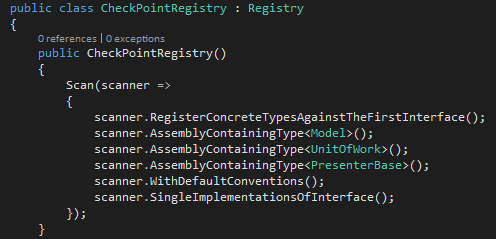


Figure 9‑26: CheckPointRegistry Class

In this case, it will discover the CheckPointRegistry class, as shown in *Figure 9‑26: CheckPointRegistry Class*, and begin scanning according to the criteria found inside the “Scan” method. This means that the container will search any directory in the solution where the “Model”, “UnitOfWork”, and “PresenterBase” classes reside and register the interfaces that are implemented by those classes. StructureMap will now know which implementations to provide for a class that is dependent on the interfaces held in the container.

When our View loads it calls a method to “CreatePresenter()” as shown in *Figure 9‑27: Base View Class*.

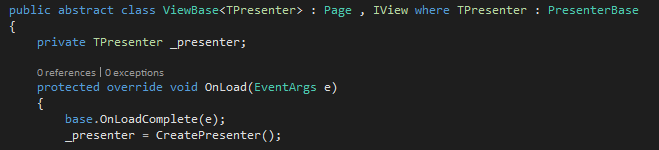


Figure 9‑27: Base View Class

The ‘CreatePresenter()’ method in turn calls the “GetPresenter<TPresenter>(this)” method in the IOC class, as shown in *Figure 9‑28: CreatePresenter Method*, and the view inserts itself as an argument to the method.

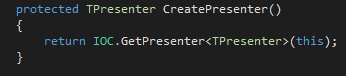


Figure 9‑28: CreatePresenter Method

Finally, when the “GetPresenter<TPresenter>(object view)” method inside the IOC class is called it returns a Presenter object with all of its dependencies satisfied, one of them being the View that initiated the method call in the first place. The ‘GetPresenter()’ method is shown below in *Figure 9‑29: GetPresenter Method*.

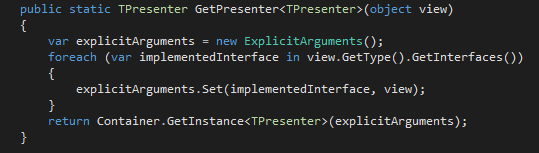


Figure 9‑29: GetPresenter Method

Each time a View is loaded this procedure will run to ensure that its appropriate Presenter will be supplied and that the Presenter will have been “injected” with all of the objects it depends on.