Greenwich Community Theatre - Prototype System

Group 1 Submission Report

|  |  |
| --- | --- |
| Student | Student Banner ID |
| Submitter: |  |
| Colleague: |  |
| Colleague: |  |
| Colleague: |  |
| Colleague: |  |

Table of Contents

[1. Introduction to GCT Online Services 3](#_Toc5024626)

[2. Design Artefacts 4](#_Toc5024627)

[a. UML Use Case Diagram 4](#_Toc5024628)

[b. Sequence Diagrams 5](#_Toc5024629)

[c. State chart Diagrams 8](#_Toc5024630)

[d. UML Design Class Diagram 9](#_Toc5024631)

[3. Design Patterns 10](#_Toc5024632)

[a. Model-View-Controller (MVC) 10](#_Toc5024633)

[Code Sample for the scenario from above: 11](#_Toc5024634)

[b. Singleton 12](#_Toc5024635)

[4. Design/ Implementation problems 13](#_Toc5024636)

[a. Database Design 13](#_Toc5024637)

[b. Components communication 13](#_Toc5024638)

[c. Data validation 13](#_Toc5024639)

[d. User interface design 14](#_Toc5024640)

[e. Implementing UML Class Diagram using an O/RD 14](#_Toc5024641)

[f. C# properties 14](#_Toc5024642)

[5. HCI Factors 15](#_Toc5024643)

[6. Work Breakdown 20](#_Toc5024644)

[Group/Team Name: Group 1 20](#_Toc5024645)

[Main responsibilities for each group member 20](#_Toc5024646)

[6. Appendix 20](#_Toc5024647)

[a. Code listings 20](#_Toc5024648)

[GCTContext- DB & Model 20](#_Toc5024649)

[HomeController, AccountController and UsersController 24](#_Toc5024650)

[Views 28](#_Toc5024651)

[b. Meetings Minutes 30](#_Toc5024652)

[Thursday 17th January 2019 30](#_Toc5024653)

[Wednesday 23rd January 2019 30](#_Toc5024654)

[Wednesday 6th February 2019 30](#_Toc5024655)

[Wednesday 20th February 2019 31](#_Toc5024656)

[Wednesday 6th March 2019 31](#_Toc5024657)

[Wednesday 20th March 2019 31](#_Toc5024658)

[Wednesday 27th March 2019 32](#_Toc5024659)

[Thursday 28th March 2019 32](#_Toc5024660)

[Saturday 30th March 2019 32](#_Toc5024661)

[Monday 1st April 2019 32](#_Toc5024662)

[7. Individual report 33](#_Toc5024663)

[a. Review: Part 1 33](#_Toc5024664)

[b. Review: Part 2 33](#_Toc5024665)

# Introduction to GCT Online Services

The product that we have delivered represents a prototype of a system that could be implemented in your company. It is designed to address the issue that theatres or cinemas are confronting when trying to move from the manual booking to an online system.

The website that we have designed is addressed to PC users, the website can still be accessed on mobile devices however some functionality is not supported. The website presents the user with a modern looking graphical user interface, which is secure, fast and reliable. The website allows the user to register as either an individual customer or, as an agency or social clubs. If the user registers as an agency or social club they will need to be accepted by a manager of the system. Once their account has been registered, they will be able to benefit from discounted tickets.

The login process is simple, no matter who the user is, all users login from the same page and once their identity has been recognised they are directed to the correct layout of the website. For example, if a manager logs into the website it will give them additional functionality, as they have administrative permissions that standard users do not have access to.

The interface that the user sees is quite intuitive and walks them through the site from the beginning to end of a given task that requires multiple steps to complete, this task might be adding a ticket of a selected performance to the basket. This process first requires the user to select a play, then choose a date from the list provided, followed by selecting a seat (or more) and finally, adding the selected seats into the basket where they are displayed as individual tickets.

It uses SQL Server as a database system which can easily handle thousands of users, thus the system is scalable as it can support a lot of users. The passwords are not stored directly in the database, first a hash function is run, and a password hash is generated, therefore it is extremely difficult to hack. In order to hack the system someone would first need the hash algorithm and then computational power. Additionally, payments that are made are very secure since the card details never touch the backend of the system, this is because Stripe API services are used. If a customer was to save their card details it is only their token that is stored in our data, which is useless to a hacker if they do not possess our private encryption key.

The website is easy for the managers and sales staff to manage as the naming conventions used are relatively intuitive, they relate to the actions that they take when it comes to managing users, printing tickets and verifying order details.

# Design Artefacts

## UML Use Case Diagram

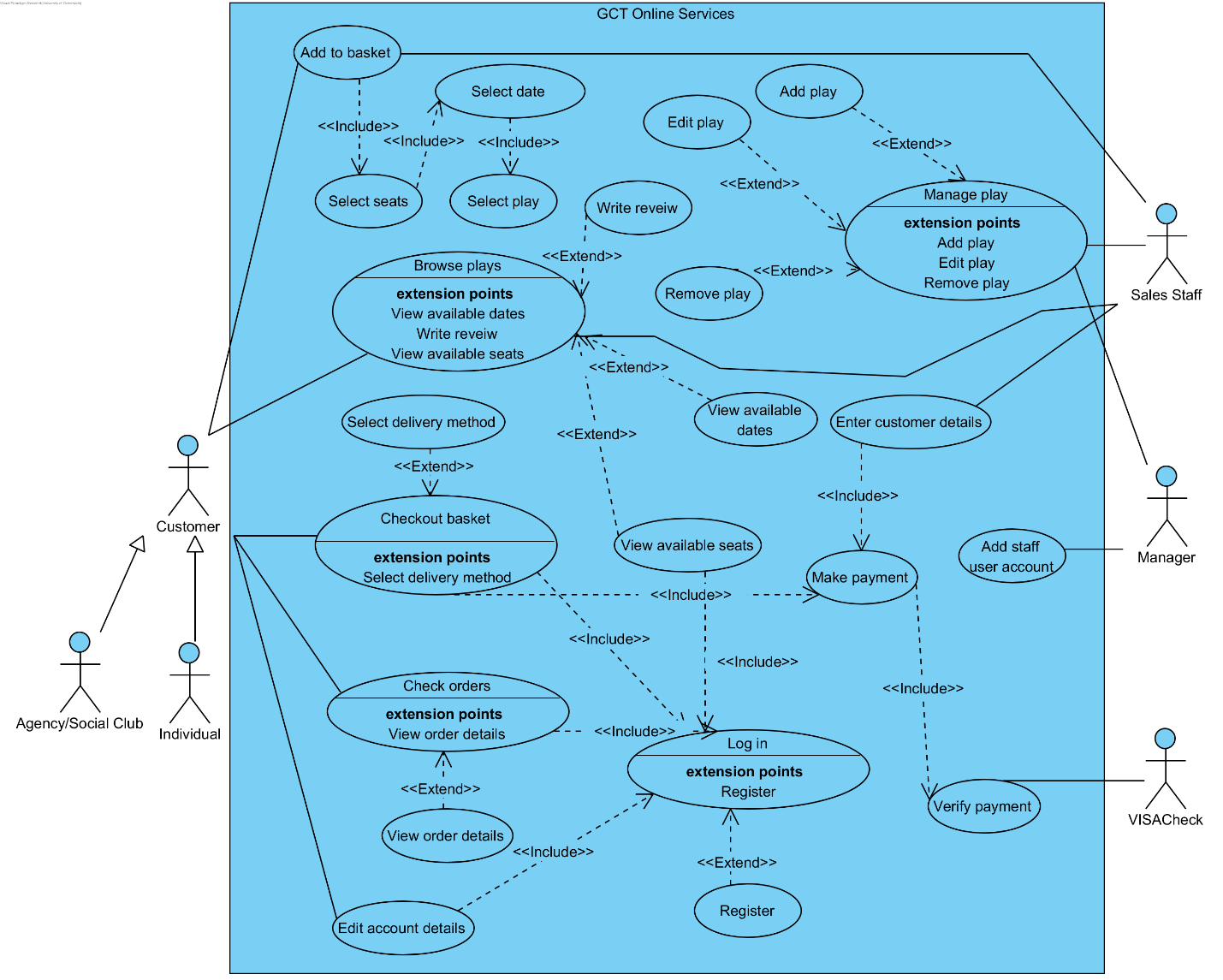


Figure 1 Use case diagram

## Sequence Diagrams

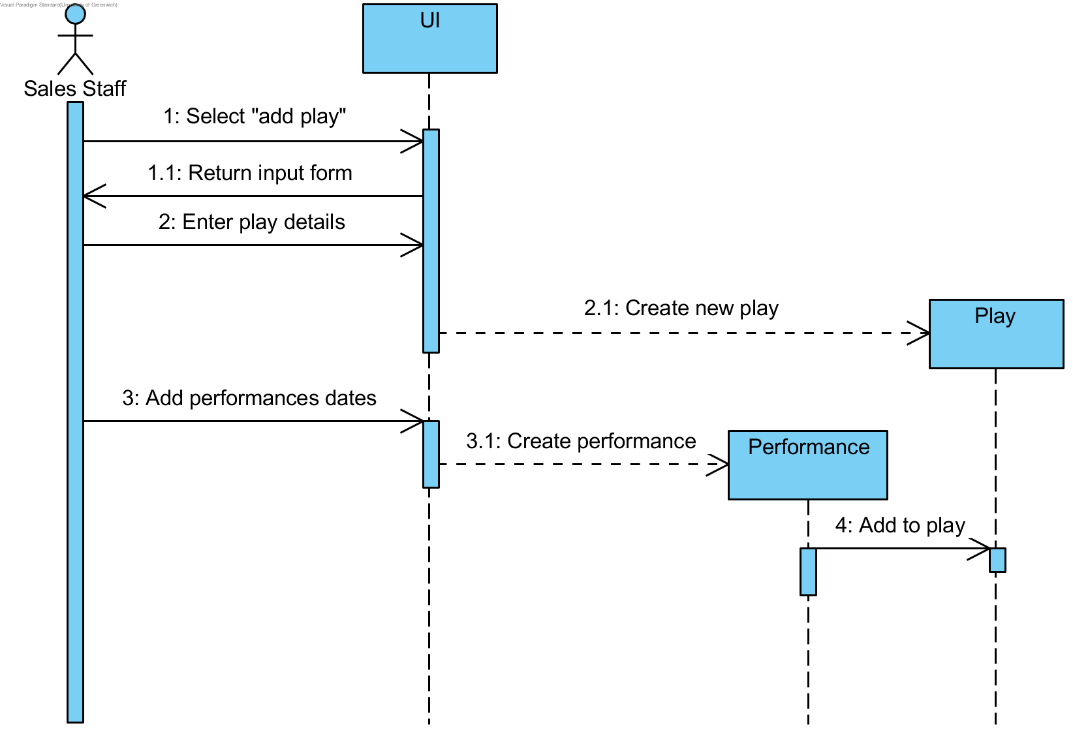


Figure 2 Sequence: Add play

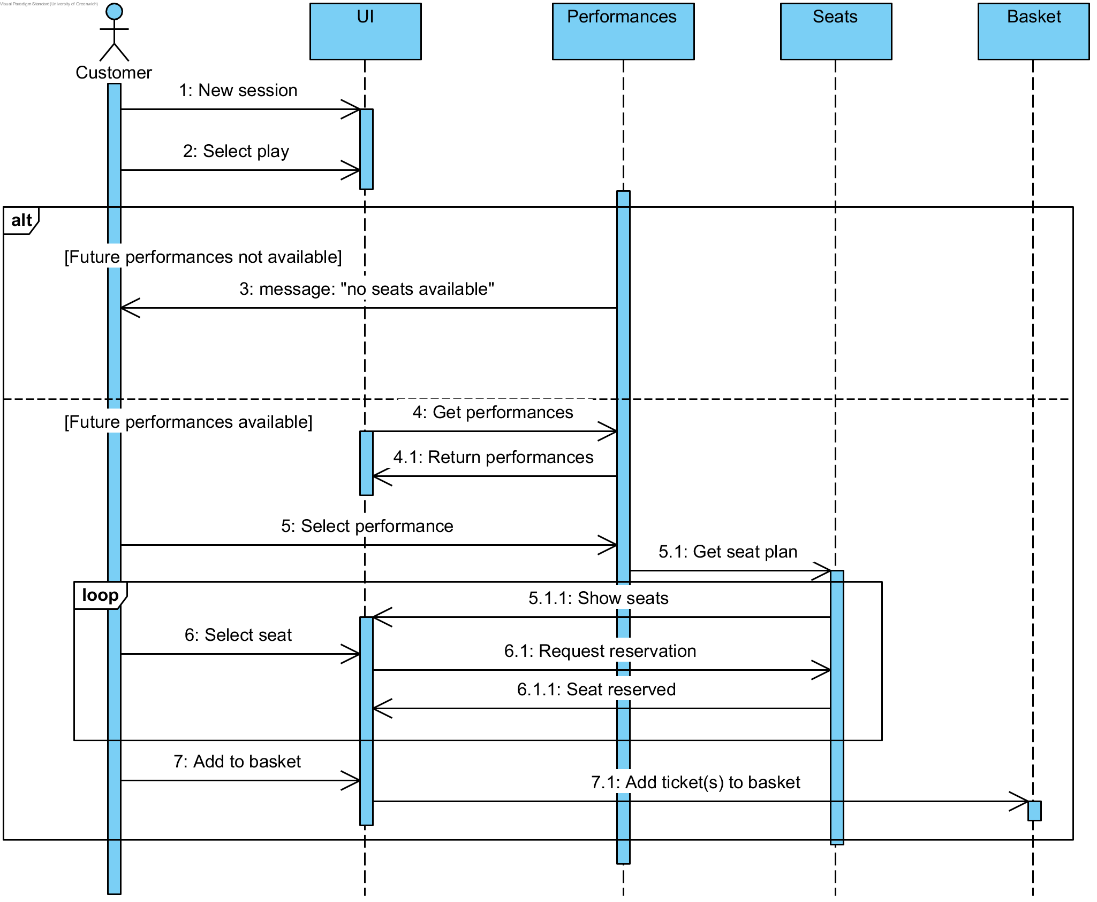


Figure 3 Sequence: Add to basket

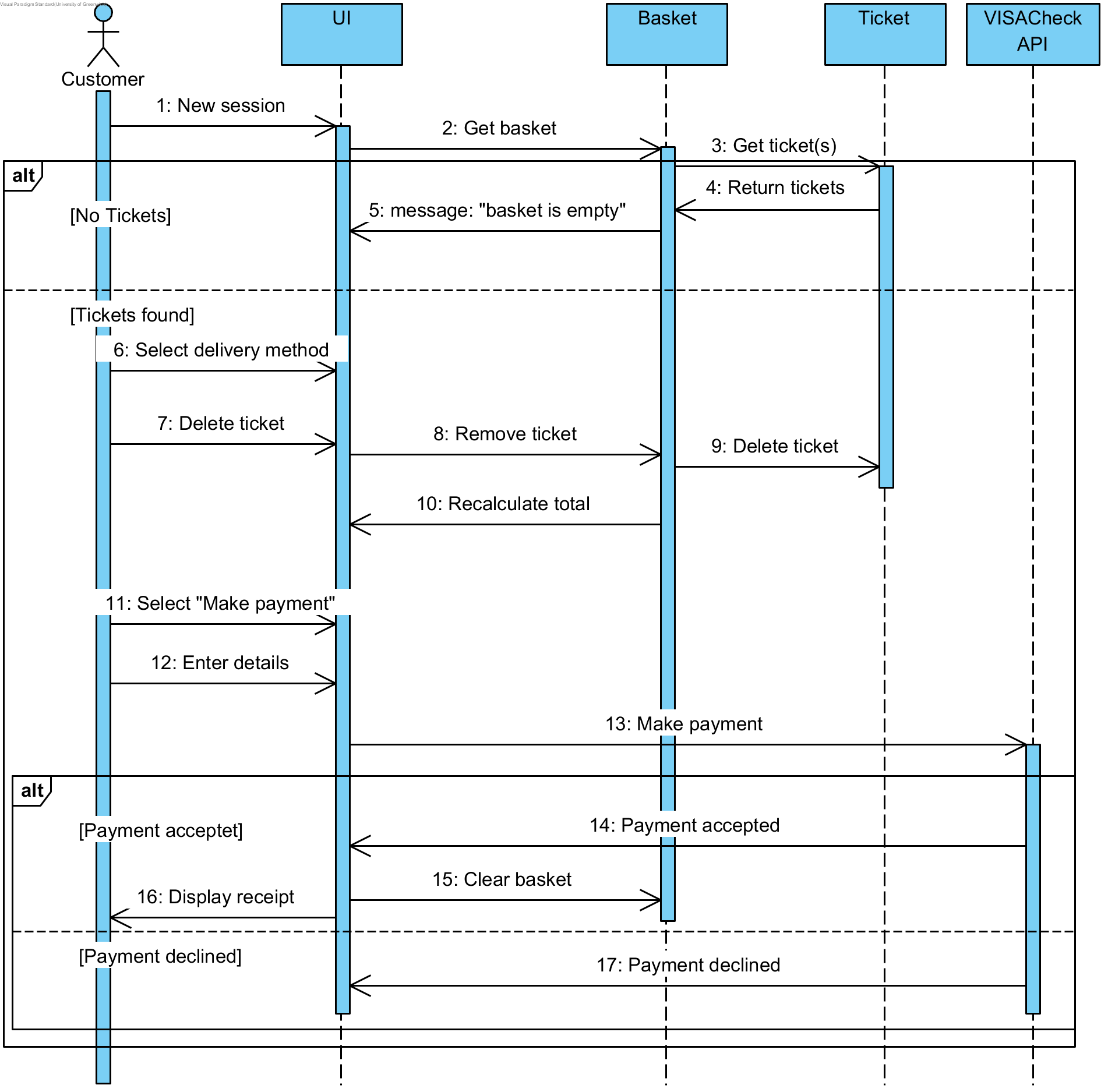


Figure 4 Sequence: Checkout

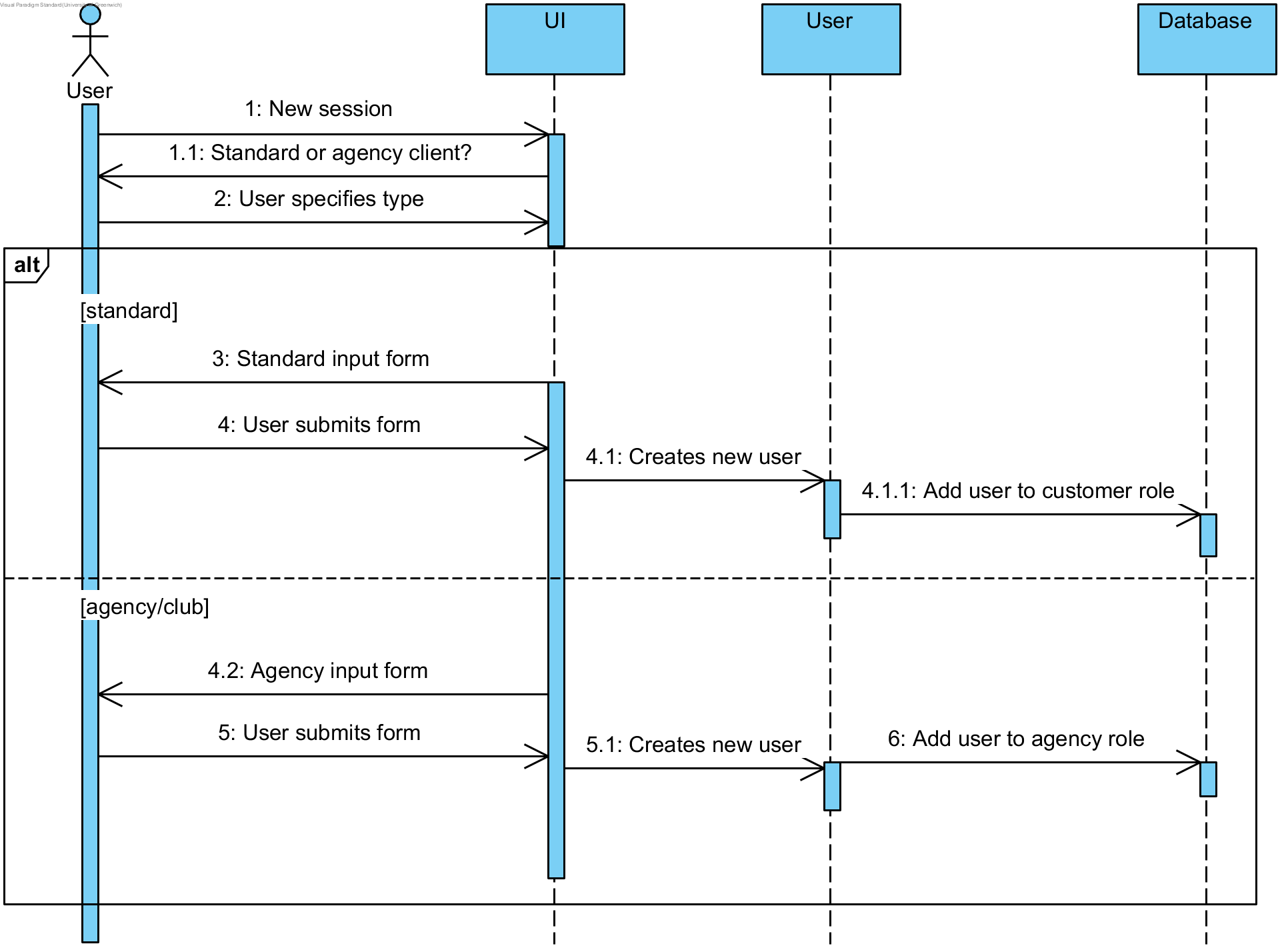


Figure 5 Sequence: Register

## State chart Diagrams

|  |  |
| --- | --- |
| Figure 6 State diagram: Basket | Figure 7 State diagram: Seat |

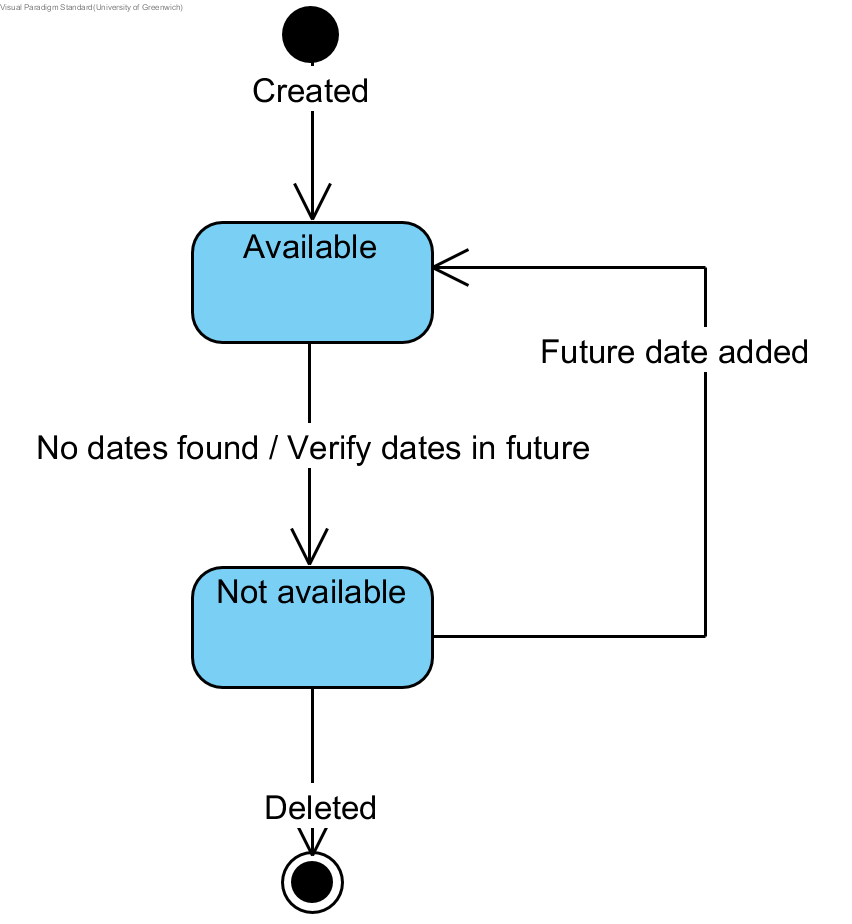
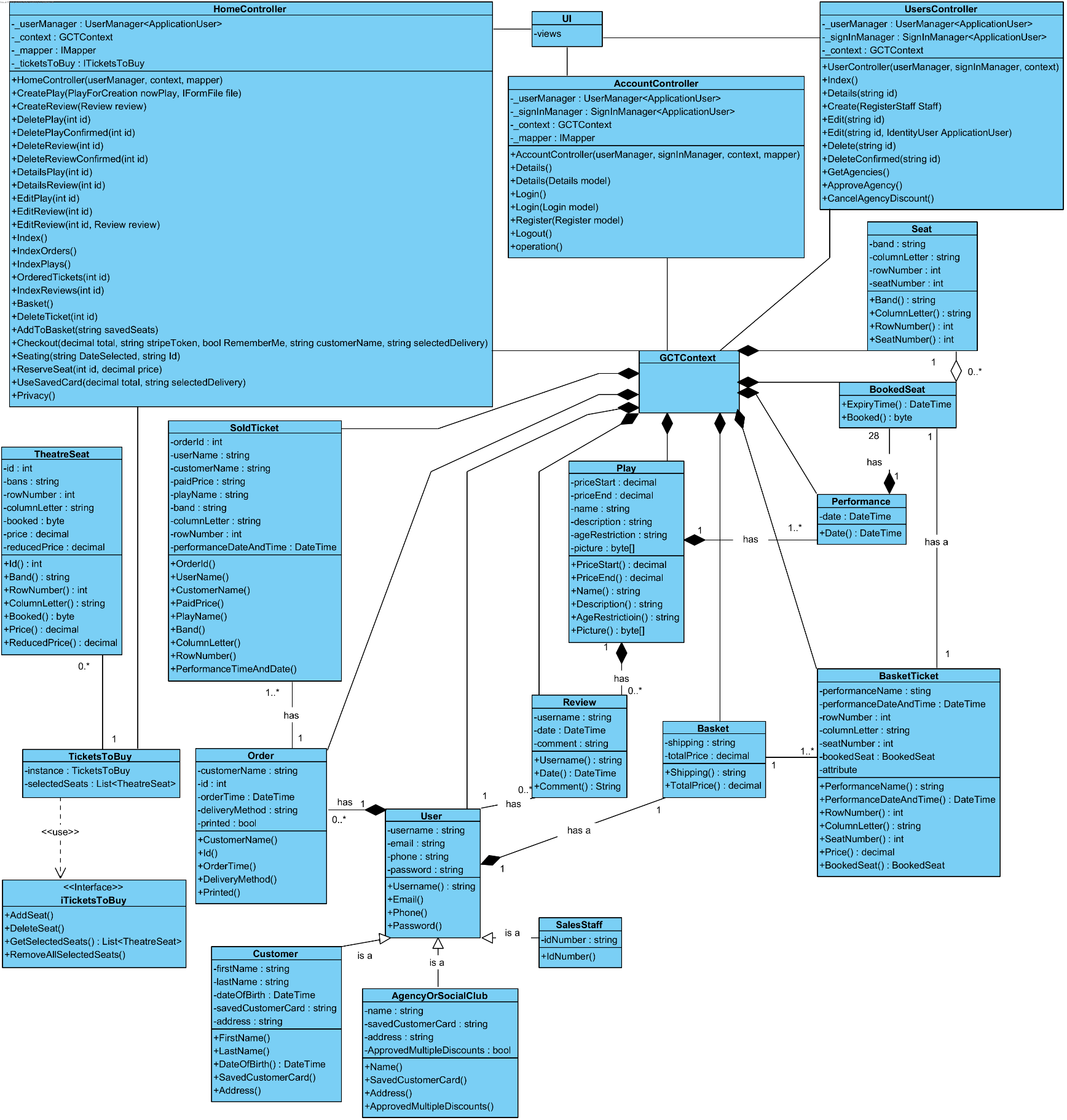


Figure 8 State diagram: Play

## UML Design Class Diagram

This class diagram is representational of the program code. For the sake of clarity, getter and setter methods have been omitted for model classes. This reflects the way getters and setters are implemented in C# (example in Section 4, part f).

# Design Patterns

This application makes a strong use of MVC design pattern as it is an ASP.Net Core web application with a prebuild MVC design. Additionally, we attempted to enrich the application with other design patterns such as Singleton pattern.

## Model-View-Controller (MVC)

A simple example of using MVC is the following:

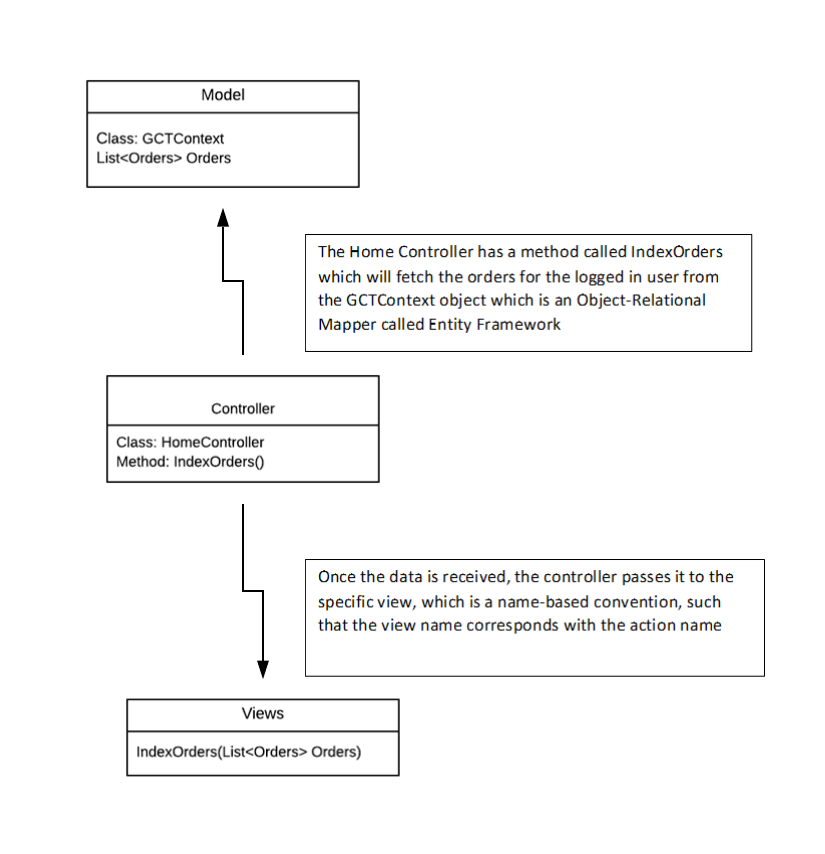


Figure 9 MVC Example

### Code Sample for the scenario from above:

#### Model class that contains data:

public class GCTContext : IdentityDbContext<ApplicationUser>

{

public GCTContext(DbContextOptions<GCTContext> options)

: base(options)

{

}

...

public virtual DbSet<Order> Orders { get; set; }

...

}

#### Controller Class that fetches data:

public class HomeController : Controller

{

...

// get all the orders for a specific client

public async Task<IActionResult> IndexOrders()

{

var user = await \_userManager.GetUserAsync(User);

var orders = await \_context.Orders.Where(x => x.UserId == user.Id).ToListAsync();

return View(orders);

}

...

}

#### View receives the data from controller and displays

As it can be noticed the called View has same name as the action in the controller

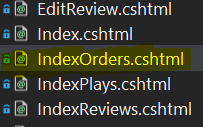


Figure 10 IndexOrders View

Code inside the view:

@model IEnumerable<Order>

@{

ViewData["Title"] = "Orders";

}

...

## Singleton

The Singleton pattern is implemented in the class called TicketsToBuy in such a way that we are taking advantage of the fact that it can be only one instance of it thus an efficient way of using it is for storing information. This class is used to keep track of the seats selected by the user when he/she selects a seat from the graphical interface, instead of adding the seats directly to the basket.

[HttpGet]

public async Task<ActionResult> ReserveSeat(int id, decimal price)

{

...

List<TheatreSeat> savedSeats = \_ticketsToBuy.GetSelectedSeats();

...

// look if seat already added, if yes remove it

foreach (TheatreSeat savedSeat in savedSeats)

{

if (savedSeat.Id == id)

{

\_ticketsToBuy.DeleteSeat(id);

bookSeatResult.Result = "unsuccessful";

return Json(bookSeatResult);

}

}

...

Also, in the code implementation one could notice that the constructor is private, there is a static method to return the instance and the methods are synchronized:

private static TicketsToBuy instance = null;

private List<TheatreSeat> selectedSeats;

private TicketsToBuy()

{

selectedSeats = new List<TheatreSeat>();

}

// get the instance if already created

[MethodImpl(MethodImplOptions.Synchronized)]

public static TicketsToBuy GetInstance()

{

if (instance == null)

{

instance = new TicketsToBuy();

}

return instance;

}

// add a seat

[MethodImpl(MethodImplOptions.Synchronized)]

public void AddSeat(TheatreSeat seat)

{

selectedSeats.Add(seat);

}

...

# Design/ Implementation problems

There were multiple issues that occurred when we started to implement the design we had such as: database configuration, components communication, data validation, user interface design, C# properties and implementation of the UML Class Diagram using an Object-relational mapper (O/RD) such as EntityFramework. We attempted to add a solution that will allow user to dynamically change the content of the current webpage through JavaScript and CSS formatting.

## Database Design

None of us had much prior knowledge to working with databases which represented a big issue since the project is based on persistent storage. So, we used Visual Studio which has an integrated UI Designer for the tables, the additional code produced when adding foreign key constraints and actions that will take place on deletion of the created entities where still unknown. This impediment was overcome through research and practice, fortunately Microsoft provides a lot of tutorials and videos that came in handy in the learning process. It required some time to get a good degree of understanding of SQL Server database system.

## Components communication

Another challenging factor was handling the communication between various components. It did not take long to realise that returning data from a view does not always work as expected, especially if a model was passed previously. There are few ways of handling communication from controller to view and vice-versa, however when using dictionary of objects such as ViewData and ViewBag since their values only exists for sending data attempting to return their content will result in errors, also misspelling their actual name will not show errors so no data will be display if mistyped.

## Data validation

Despite the Don’t Repeat Yourself (DRY) approach available in ASP.NET, there were still issues that would pop up from time to time when having to deal with input forms such as having to change the input fields dynamically based on user’s selection. Therefore, required field for one side of the form will not always be the same as the ones from the other side, consequently resulting in a invalid model state, although on the front side everything seemed filled in and correct. One of the approaches to overcome this impediment was to remove the validation from the model to the UI through JavaScript, the other solution was to do the check inside the controller and sending feedback to the View. According to more experienced programmers, custom validation shall be carried in the front end, thus reducing the server requests.

## User interface design

Not having much experience with CSS and JavaScript proved troublesome when we decided to build a web application, one of the most difficult challenges consisted in building the seating plan representation, luckily, we discovered Bootstrap which is very helpful in implementing basic design for this kind of projects, there were plenty of examples available online and W3Schools helped a lot with JavaScript learning. Another challenge was to allow the user to reserve a seat without needing to reload the page, this was achieved by using Ajax and jQuery to edit the selected seats and make a call to the controller to reserve a seat for ten minutes, giving the user enough time to finish the booking.

## Implementing UML Class Diagram using an O/RD

Having to do queries to the database using the structured query language can be a pain, thus we decided to make use of some of the latest technologies available for web application developers such as EntityFramework which transforms queries into simple object-oriented classes where there is a class that maps all the tables from the database into their copies written in code.

## C# properties

Another factor that is confusing first it is the properties which come as part of C# programming language where a variable does not need to be declared first, instead a property can be created to do the get and set methods that the variable will need. Therefore, confusion was raised for designing the UML diagram with C# properties. For example:

Private int attribute;

Public int getAttribute()

{

Return attribute;

}

Public void setAttribute(int attribute)

{

This.attribute = attribute;

}

All the code from above becomes a one-line code which extremely simplifies the work the programmer has to do.

Public int Attribute {get; set;}

# HCI Factors

The user interface has been designed in accordance with modern UX principles. At any given time, only the information required to perform an action is displayed to avoid the user becoming overwhelmed.

From the early stages of the system design process, the way users would interact with the system was a key priority and influenced the way the underlying code was written. For example, the order in which functionality is presented to the user is performed by the views. The aim was to create interfaces and procedures that balance simplicity with speed and functionality, with the understanding that the way a system “looks and feels” can greatly affect the reputation of the company it represents.

The system was designed to be familiar to users who navigate websites frequently. The homepage consists of a menu bar at the top which is segregated from the body in order to make it easily identifiable. The menu contains links which vary depending on the user’s permission level. Every user always sees a link to the homepage and privacy policy (which is a placeholder in this prototype system). Guest users (those who are not logged in) can only see the option to log in or register; customers can access their basket, manage their account and log out; and sales staff can additionally manage performances.

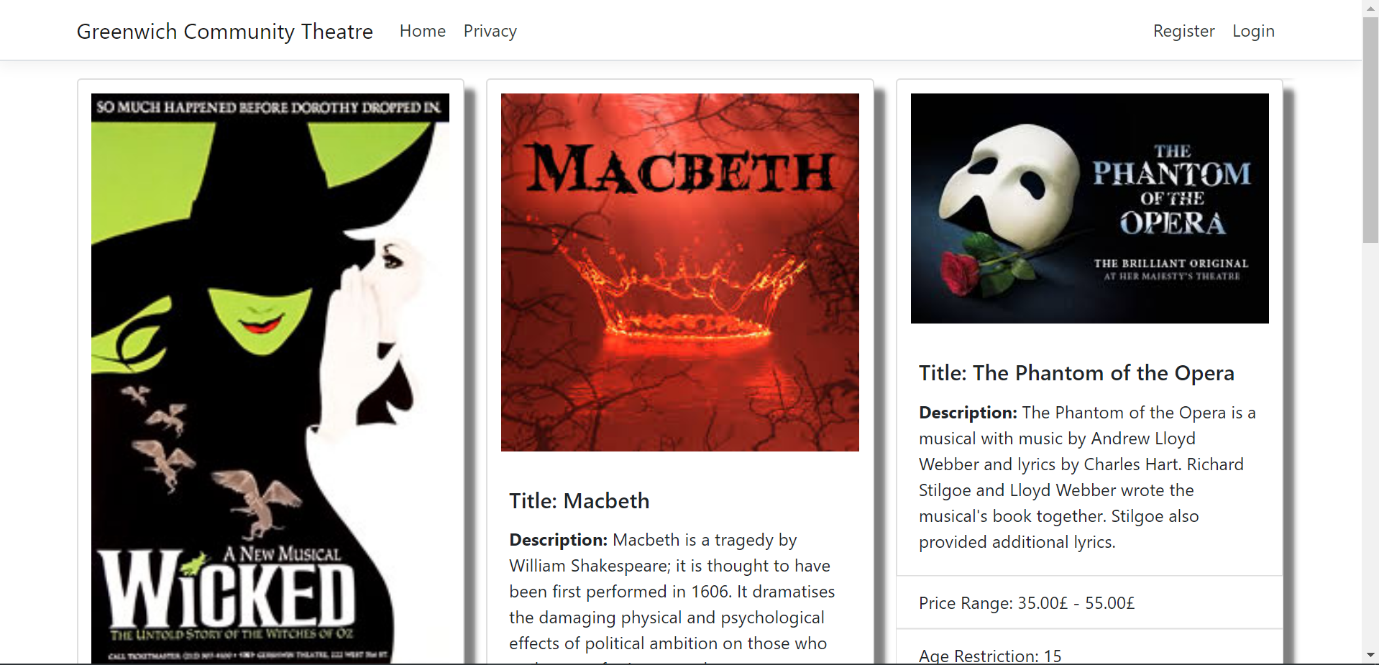


Figure 11 Home page view for a user who is not logged in

The homepage shows upcoming performances in a grid layout, with each play displayed with its artwork, title, description and age rating. Clicking on any of these grid items brings up an expanded view of the play, which also shows which dates are available to see that play. Users can select a date and press “Book tickets”. This redirects them to a seating plan with seat availability displayed dynamically via a colour coding scheme.

The booking interface displays an approximation of the layout of seats in the theatre, with each of the three price band boundaries indicated with black lines. Every seat is colour-coded: available seats are white and unavailable seats are red. The red colouring follows the common traffic light convention that is ubiquitously recognised among cultures. When a user selects an available seat, its colour is changed to blue to reflect this. The combined total of all selected seats is displayed at the bottom of the screen (next to the “add to basket” button) for the convenience of the user.

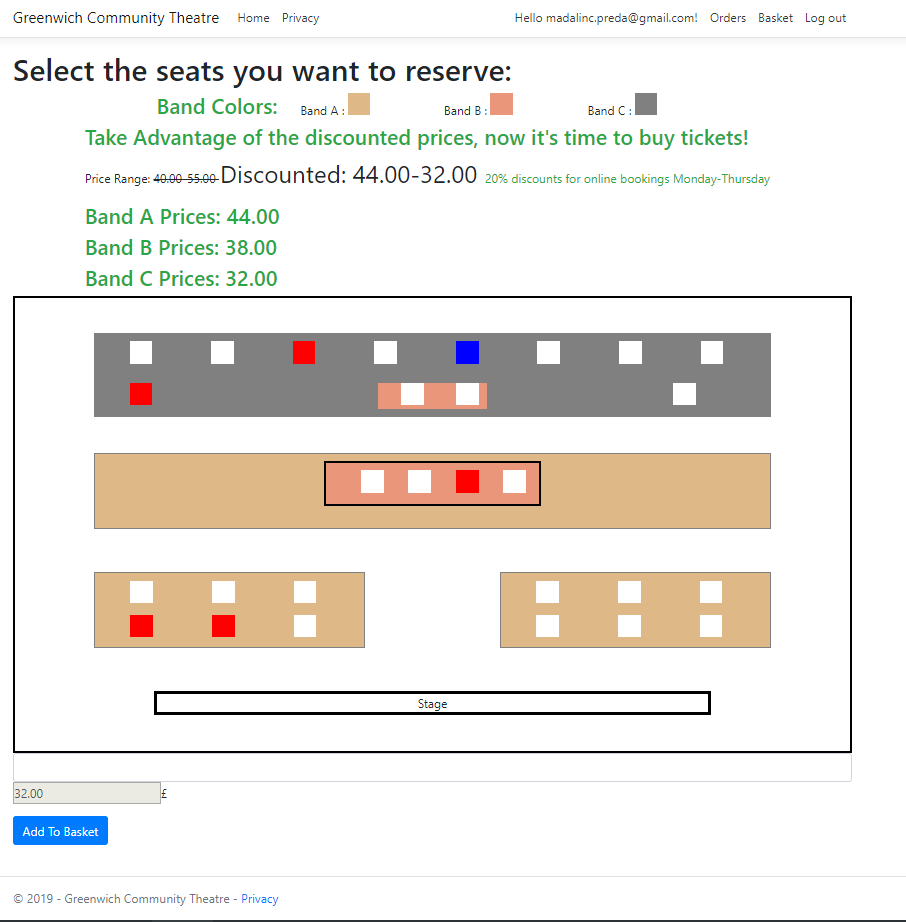


Figure 12 Seat selection interface

In order to make a booking, a customer must first be logged in. To log in, they must simply enter their email address (which serves as their username) and password and click “log in”. This creates basket to which tickets can be added. They are then able to browse shows and their performance dates, select dates from drop down lists and book seats for them.

At any time while logged in, users can view their basket, with details of any tickets reserved inside it.

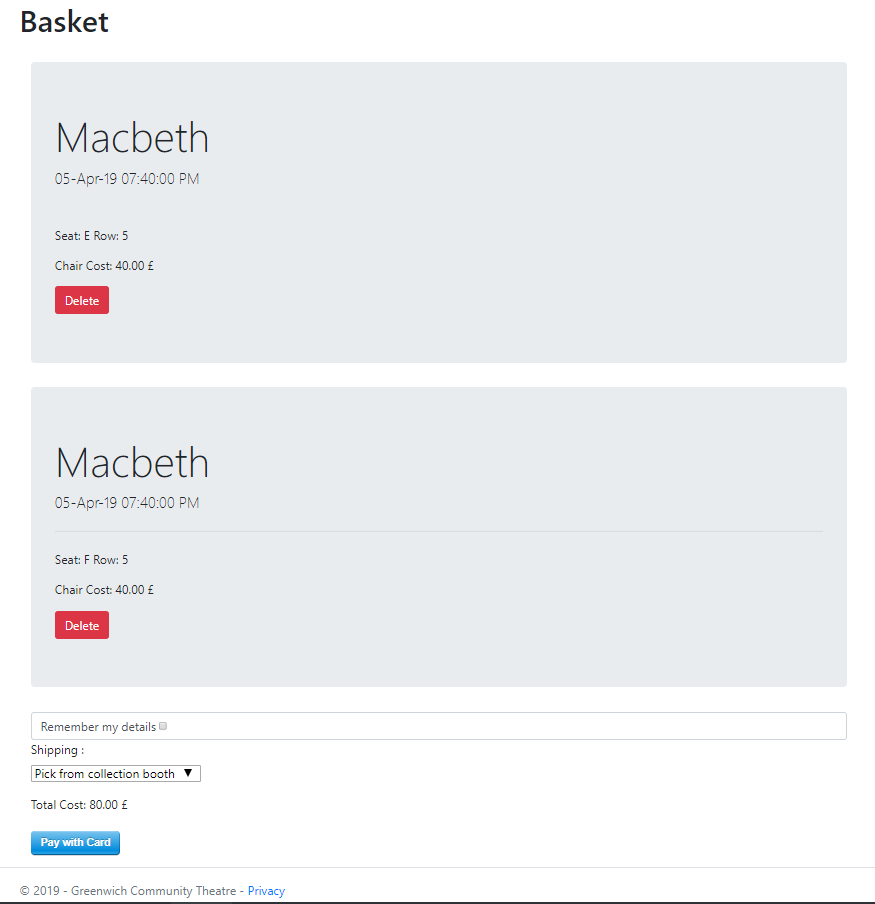


Figure 13 The customer basket showing two reserved tickets

New users must register before being able to purchase tickets using the system. The “Register” button is displayed on every page while a user is not logged in. Pressing this results in a form being displayed which the user must fill in to create an account. At this stage, the system collects information including the user’s name, email address and postal address (for billing and mailing purposes). Collecting as much information as possible at the time of registration simplifies future booking processes and avoids the user having to repeatedly enter the same information.

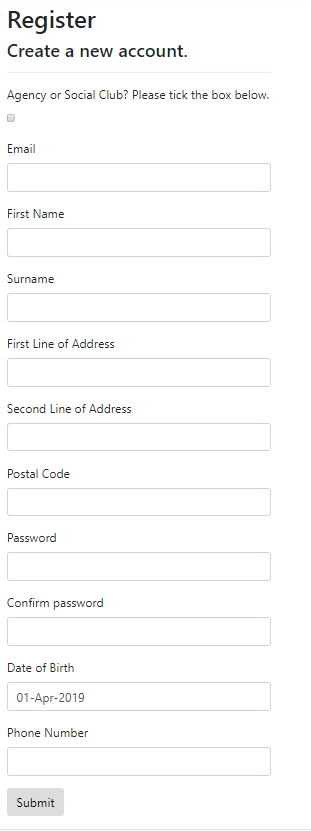


Figure 14 The customer registration form

The decision to use the email address as a username adds an extra layer of simplicity to the customer registration process, requiring less information from the user than requiring a separate username and email address.

The staff features follow similar HCI patterns. CSS styles and layout concepts are shared between the customer and staff user types. Site functionality is modularised in a task-orientated fashion. For example, when a staff member wishes to add, remove or edit performance details, only information and functionality (apart from macro site navigation) relating to performances is displayed to them.

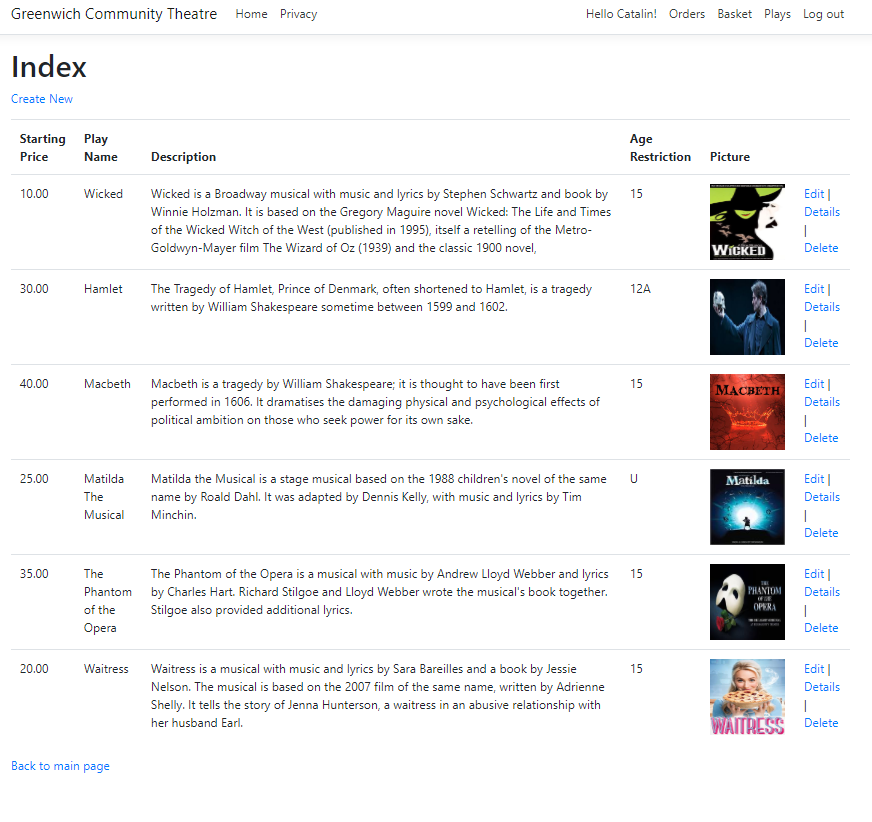


Figure 15 The staff play management interface

It is our opinion that the HCI patterns in the prototype are efficient and provide a good user experience. Because the processes are intuitive, it eliminates the requirement for a user to first study a user guide in order to use the software.

# Work Breakdown

## Group/Team Name: Group 1

|  |  |  |  |
| --- | --- | --- | --- |
| Team member name | Student ID | individual overall work contribution (%) | Signature |
| Joshua Bennett | 000983229 | 20 | J.B. |
| Matthew Peter Dredge | 000935649 | 20 | M.P.D. |
| Wayne Edward Harvey-Pilot | 000933367 | 20 | W.E.H-P |
| Madalin-Cristian Preda | 000937119 | 20 | M-C.P. |
| Thomas Lee Stoyles | 000990057 | 20 | T.L.S. |
| **Total 100%** | | |  |

## 

## Main responsibilities for each group member

|  |  |
| --- | --- |
| Madalin-Cristian Preda | Team leader  Programmer  Tester |
| Joshua Bennett | Coordinator  Tester  Design class diagram |
| Matthew Dredge | Use case diagram |
| Wayne Harvey-Pilot | State chart diagrams |
| Thomas Stoyles | Sequence diagrams |

# 

# Appendix

## Code listings

### GCTContext- DB & Model

public class GCTContext : IdentityDbContext<ApplicationUser>

{

public GCTContext(DbContextOptions<GCTContext> options)

: base(options)

{

}

public virtual DbSet<ApplicationUser> ApplicationUsers { get; set; }

public virtual DbSet<Basket> Basket { get; set; }

public virtual DbSet<BasketTicket> BasketTickets { get; set; }

public virtual DbSet<BookedSeat> BookedSeats { get; set; }

public virtual DbSet<Order> Orders { get; set; }

public virtual DbSet<Play> Plays { get; set; }

public virtual DbSet<Performance> Performances { get; set; }

public virtual DbSet<Review> Reviews { get; set; }

public virtual DbSet<Seat> Seats { get; set; }

public virtual DbSet<SoldTicket> SoldTickets { get; set; }

}

#### Play

public partial class Play

{

public Play()

{

Performances = new HashSet<Performance>();

Reviews = new HashSet<Review>();

}

public int Id { get; set; }

[Display(Name = "Starting Price")]

public decimal PriceStart { get; set; }

[Display(Name = "Last Price")]

public decimal PriceEnd { get; set; }

[Display(Name = "Play Name")]

public string Name { get; set; }

public string Description { get; set; }

[Display(Name = "Age Restriction")]

public string AgeRestriction { get; set; }

public byte[] Picture { get; set; }

public virtual ICollection<Performance> Performances { get; set; }

public virtual ICollection<Review> Reviews { get; set; }

}

#### Performance

public partial class Performance

{

public Performance()

{

BookedSeats = new HashSet<BookedSeat>();

}

public int Id { get; set; }

public int PlayId { get; set; }

public DateTime Date { get; set; }

public virtual Play Play { get; set; }

public virtual ICollection<BookedSeat> BookedSeats { get; set; }

}

#### ApplicationUser

public class ApplicationUser : IdentityUser

{

public string IdNumber { get; set; }

public string AgencyOrClubName { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public DateTime DateOfBirth { get; set; }

public string SavedCustomerCard { get; set; }

public string Address { get; set; }

public bool? ApprovedMultipleDiscounts { get; set; }

public virtual Basket Basket { get; set; }

public virtual ICollection<Order> Orders { get; set; }

public virtual ICollection<Review> Reviews { get; set; }

}

#### Basket

public partial class Basket

{

[Key]

[ForeignKey("User")]

public string UserId { get; set; }

public string ShippingMethod { get; set; }

public decimal TotalPrice { get; set; }

public virtual ApplicationUser User { get; set; }

public virtual ICollection<BasketTicket> Tickets { get; set; }

}

#### BasketTickets

public partial class BasketTicket

{

public int Id { get; set; }

public string BasketId { get; set; }

public int PerformanceId { get; set; }

public int BookedSeatId { get; set; }

public decimal Price { get; set; }

public virtual BookedSeat BookedSeat { get; set; }

public virtual Performance Performance { get; set; }

public virtual Basket Basket { get; set; }

}

#### BookedSeat

public partial class BookedSeat

{

public BookedSeat()

{

BasketTickets = new HashSet<BasketTicket>();

}

public int Id { get; set; }

public int PerformanceId { get; set; }

public int SeatId { get; set; }

public byte Booked { get; set; }

public DateTime? ExpiryTime { get; set; }

public virtual Performance Performance { get; set; }

public virtual Seat Seat { get; set; }

public virtual ICollection<BasketTicket> BasketTickets { get; set; }

}

#### Order

public partial class Order

{

public Order()

{

SoldTickets = new HashSet<SoldTicket>();

}

[Display(Name = "Order Number")]

public int Id { get; set; }

public string UserId { get; set; }

[Display(Name = "Customer Name")]

public string ClientName { get; set; }

[Display(Name = "Date and Time Placed")]

public DateTime OrderTime { get; set; }

[Display(Name = "Shipping Chosen")]

public string DeliveryMethod { get; set; }

[Display(Name = "Printed")]

public bool IsPrinted{ get; set; }

public virtual ApplicationUser User { get; set; }

public virtual ICollection<SoldTicket> SoldTickets { get; set; }

}

#### Review

public partial class Review

{

public int Id { get; set; }

public int PlayId { get; set; }

public string UserId { get; set; }

public string UserName { get; set; }

[StringLength(400, ErrorMessage = "The {0} must be at least {2} and at max {1} characters long.", MinimumLength = 6)]

public string Comment { get; set; }

public DateTime Date { get; set; }

public virtual Play Play { get; set; }

public virtual ApplicationUser User { get; set; }

}

#### Seat

public partial class Seat

{

public Seat()

{

BookedSeats = new HashSet<BookedSeat>();

}

public int Id { get; set; }

public string Band { get; set; }

public string ColumnLetter { get; set; }

public int RowNumber { get; set; }

public int SeatNumber { get; set; }

public virtual ICollection<BookedSeat> BookedSeats { get; set; }

}

#### SoldTicket

public partial class SoldTicket

{

public int Id { get; set; }

public int OrderId { get; set; }

public string UserId { get; set; }

public string CustomerName { get; set; }

public decimal PaidPrice { get; set; }

public string PlayName { get; set; }

public string Band { get; set; }

public string ColumnLetter { get; set; }

public int RowNumber { get; set; }

public DateTime PerformanceTimeAndDate { get; set; }

public virtual Order Order { get; set; }

public virtual ApplicationUser User { get; set; }

}

### HomeController, AccountController and UsersController

Due to the length of the code in those three classes, only UsersController is provided in this section as the ZIP file with code is attached to the final report submission as well. Attaching HomeController class means, alone, around 50 pages

// controller class that only manager can access

[Authorize(Roles = "Manager")]

public class UsersController : Controller

{

private readonly UserManager<ApplicationUser> \_userManager;

private readonly SignInManager<ApplicationUser> \_signInManager;

private readonly GCTContext \_context;

public UsersController(UserManager<ApplicationUser> userManager,

SignInManager<ApplicationUser> signInManager,

GCTContext context)

{

\_userManager = userManager;

\_signInManager = signInManager;

\_context = context;

}

// return all the users

public async Task<IActionResult> Index()

{

return View(await \_context.Users.ToListAsync());

}

//return all agencies

public async Task<IActionResult> GetAgencies()

{

var agencies = await \_context.ApplicationUsers.Where(x => x.ApprovedMultipleDiscounts != null).ToListAsync();

return View(agencies);

}

// approve agency discount

public async Task<IActionResult> ApproveAgency(string id)

{

var agency = await \_context.ApplicationUsers.FirstOrDefaultAsync(x => x.Id == id);

if (agency == null)

{

return RedirectToAction(nameof(GetAgencies));

}

agency.ApprovedMultipleDiscounts = true;

\_context.ApplicationUsers.Update(agency);

await \_context.SaveChangesAsync();

return RedirectToAction(nameof(GetAgencies));

}

// cancel discount for agency

public async Task<IActionResult> CancelAgencyDiscount(string id)

{

var agency = await \_context.ApplicationUsers.FirstOrDefaultAsync(x => x.Id == id);

if (agency == null)

{

return RedirectToAction(nameof(GetAgencies));

}

agency.ApprovedMultipleDiscounts = false;

\_context.ApplicationUsers.Update(agency);

await \_context.SaveChangesAsync();

return RedirectToAction(nameof(GetAgencies));

}

// view details of users

public async Task<IActionResult> Details(string id)

{

if (id == null)

{

return NotFound();

}

var ApplicationUser = await \_context.Users

.FirstOrDefaultAsync(m => m.Id == id);

if (ApplicationUser == null)

{

return NotFound();

}

return View(ApplicationUser);

}

// view to create a staff user

public IActionResult Create()

{

return View();

}

// submit the details of the new member of staff

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Create([Bind("Email,Number,Name,Password," +

"ConfirmPassword")]RegisterStaff Staff)

{

if (ModelState.IsValid)

{

var user = new ApplicationUser() { Email = Staff.Email, UserName = Staff.Name, IdNumber = Staff.Number };

var result = await \_userManager.CreateAsync(user, Staff.Password);

if (result.Succeeded)

{

var registeredUser = await \_userManager.FindByEmailAsync(Staff.Email);

await \_userManager.AddToRoleAsync(user, "SalesStaff");

\_context.Basket.Add(new Basket() { ShippingMethod = "Collection Booth", UserId = user.Id, TotalPrice = 0 });

}

await \_context.SaveChangesAsync();

return RedirectToAction(nameof(Index));

}

TempData["UserNotifier"] = new UserNotifier()

{

CssFormat = "alert-danger",

MessageType = "Error!",

Content = "User wa not created."

};

return View(Staff);

}

// edit the details of a user

public async Task<IActionResult> Edit(string id)

{

if (id == null)

{

return NotFound();

}

var ApplicationUser = await \_context.Users.FindAsync(id);

if (ApplicationUser == null)

{

return NotFound();

}

return View(ApplicationUser);

}

// submit all the changes to user details

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Edit(string id, [Bind("UserName,Email,PhoneNumber")] ApplicationUser ApplicationUser)

{

if (id != ApplicationUser.Id)

{

return NotFound();

}

if (ModelState.IsValid)

{

try

{

\_context.Update(ApplicationUser);

await \_context.SaveChangesAsync();

}

catch (DbUpdateConcurrencyException)

{

if (!ApplicationUserExists(ApplicationUser.Id))

{

return NotFound();

}

else

{

throw;

}

}

return RedirectToAction(nameof(Index));

}

return View(ApplicationUser);

}

// delete a user

public async Task<IActionResult> Delete(string id)

{

if (id == null)

{

return NotFound();

}

var ApplicationUser = await \_context.Users

.FirstOrDefaultAsync(m => m.Id == id);

if (ApplicationUser == null)

{

return NotFound();

}

return View(ApplicationUser);

}

// confirm user deleted

[HttpPost, ActionName("Delete")]

[ValidateAntiForgeryToken]

public async Task<IActionResult> DeleteConfirmed(string id)

{

var ApplicationUser = await \_context.Users.FindAsync(id);

\_context.Users.Remove(ApplicationUser);

await \_context.SaveChangesAsync();

return RedirectToAction(nameof(Index));

}

//check if user exists

private bool ApplicationUserExists(string id)

{

return \_context.Users.Any(e => e.Id == id);

}

}

### 

### Views

Again, since the project represents a website a view represents a webpage thus there are more than 20 web pages, each page having on average 90 lines of code, therefore only one view is attached which represents the main page of the application. It is a cshtml file which contains a mixture of .NET Html helpers, HTML, CSS and JavaScript.

#### Index (Main page)

@using Microsoft.Extensions.Options

@model IEnumerable<Play>

@{

ViewData["Title"] = "Home Page";

int index = (int)ViewData["Index"];

int noOfPerf = (int)ViewData["NumberOfPerformances"];

}

@section Styles

{

<style type="text/css">

div.card-link {

font-size: 2vw;

}

div.card {

display: inline-block;

}

</style>

}

<div class="card-columns">

@for (int j = 0; j < noOfPerf; j++)

{

<div class="card" style="box-shadow: 10px 10px 5px grey;">

<img class="card-img-top img-fluid" style="padding:1vw;"

src="@Url.Action("RenderPhoto", "Home", new { photoId = Model.ElementAt(j).Id })" alt="Card image cap">

<div class="card-body">

<h5 class="card-title">Title: @Html.DisplayFor(model => model.ElementAt(j).Name)</h5>

<p class="card-text">

<span class="font-weight-bold"> Description: </span> @Html.DisplayFor(model => model.ElementAt(j).Description)

</p>

</div>

<ul class="list-group list-group-flush">

<li class="list-group-item">

Price Range:

@Html.DisplayFor(model => model.ElementAt(j).PriceStart)<span>&#163;</span> - @Html.DisplayFor(model => model.ElementAt(j).PriceEnd)<span>&#163;</span>

</li>

<li class="list-group-item">Age Restriction: @Html.DisplayFor(model => model.ElementAt(j).AgeRestriction)</li>

</ul>

<div class="card-footer text-center">

@if (SignInManager.IsSignedIn(User))

{

@if (User.IsInRole("SalesStaff") || User.IsInRole("Manager"))

{

<**a** class="card-link" **asp-controller**="Home" **asp-action**="SelectDate" **asp-route-id**="@Model.ElementAt(j).Id" title="Book Tickets">Book Tickets</**a**>

}

else

{

<**a** class="card-link" **asp-controller**="Home" **asp-action**="SelectDate" **asp-route-id**="@Model.ElementAt(j).Id" title="Buy Tickets">Buy Tickets</**a**>

}

}

else

{

<**a** class="card-link" **asp-controller**="Account" **asp-action**="Login" title="Login">Login To Buy Tickets</**a**>

}

</div>

</div>

}

</div>

<script language="javascript" type="text/javascript">

$(document).ready(function () {

$("button").click(function () {

if ($("span").text().toString() != "Added");

$("span").text("Added");

});

});

</script>

## Meetings Minutes

### Thursday 17th January 2019

#### Attendees

* Joshua Bennett
* Matthew Dredge
* Thomas Stoyles
* Wayne Harvey-Pilot
* Madalin Preda

#### Discussion items

* Reviewed the work of each team member from term 1.
  + Madalin had begun programming the system over the Christmas holiday.
    - Reviewed Madalin’s existing system and agreed it was a good foundation.
    - It was agreed to use Madalin’s conceptual class structure (and pre-existing code) for this reason.
  + Agreed upon the best use cases from each use case diagram and created a rough sketch.
* Discussed roles of each team member.

### 

### Wednesday 23rd January 2019

#### Attendees

* Joshua Bennett
* Thomas Stoyles
* Wayne Harvey-Pilot
* Madalin Preda

#### Discussion items

* Quick roundup of progress during the week.
* Agreed to meet fortnightly on the day of SDP teaching in order to maximise relevancy of the meetings.
* Created a use case diagram.

### 

### Wednesday 6th February 2019

#### Attendees

* Joshua Bennett
* Matthew Dredge
* Wayne Harvey-Pilot
* Madalin Preda

#### Discussion items

* Reached consensus over the basic HCI design of the application:
  + Views presented to each user type.
* Prioritised functionality we would like to achieve.
  + Top priority: Browsing plays, buying tickets, generating tickets to print by staff, management of plays by staff, the ability to create user accounts.
  + Lower priority: Editing of user accounts, viewing a list of previous orders, special “manager” user role, mobile optimisation, editing existing plays.

### 

### Wednesday 20th February 2019

#### Attendees

* Joshua Bennett
* Matthew Dredge
* Thomas Stoyles
* Madalin Preda

#### Discussion item

* Created the design class diagram.

### 

### Wednesday 6th March 2019

#### Attendees

* Joshua Bennett
* Matthew Dredge
* Thomas Stoyles
* Wayne Harvey-Pilot
* Madalin Preda

#### Discussion items

* Created sequence diagrams.
* Created state chart diagrams.

### 

### Wednesday 20th March 2019

#### Attendees

* Joshua Bennett
* Matthew Dredge
* Thomas Stoyles
* Wayne Harvey-Pilot
* Madalin Preda

#### Discussion item

* Reviewed operation of the system.

### 

### Wednesday 27th March 2019

#### Attendees

* Joshua Bennett
* Matthew Dredge
* Thomas Stoyles
* Madalin Preda

#### Discussion items

* Authored HCI, design methodology and design/implementation problem discussions.

### 

### Thursday 28th March 2019

#### Attendees

* Joshua Bennett
* Madalin Preda

#### Discussion items

* Performed detailed system testing.
* Fixed bugs in the code.
* Began formally setting out the report with previously-created resources.

### Saturday 30th March 2019

#### Attendees

* Joshua Bennett
* Madalin Preda

#### Discussion items

* Performed detailed system testing.
* Fixed bugs in the code.
* Began formally setting out the report with previously-created resources.

### 

### Monday 1st April 2019

#### Attendees

* Joshua Bennett
* Matthew Dredge
* Thomas Stoyles
* Wayne Harvey-Pilot
* Madalin Preda

#### Discussion Item

* Finalised report.
* Planned prototype demonstration.

# Appendix B: Individual report

## Review: Part 1

## Review: Part 2