[illegible]

CO550
Web Applications
Coursework 3

Build a Simple Web Database Application

Alex Paulo da Costa

21611431

James Harris

21606555

19th January 2018

Computing & Web Development
Buckinghamshire New University

Summary

Background & Rationale

A Enterprise Model

A clerk in the head office (or operational manager) sets the routes and times they want for the bus lines.

Drivers are assigned to the routes, reducing the total time of the route to their contracted hours. The closer it is to zero (hours left to allocate) the better.

Customers can look at the timetable for the line, without knowing who the driver is, and choose the bus they want to take.

The customers board the bus, pay the driver, or show them a previously bought and still valid ticket, and take a seat.

B Functional Requirements

- Enable users to login and register
  - Normal (unprivileged users) can register anytime
  - Privileged users must be added by an admin
- Enable users and visitors to see bus timetables
- Enable registered users to save their favourite routes
- Allow admins and authorised staff to change timetables and rotas
  - Create bus lines and routes
  - Allocate staff to lines
- Allow staff to see their rota
  - Staff assigned to particular routes
  - Staff dashboard allows to print weekly rota as a pdf
- The users should be able to search routes by terms, e.g route code, town names or bus stop names

Out of these requirements, we were able to implement all but one, that being the one that would allow the staff to see their rota, since we were left with less time to implement the project due to our approach and technologies chosen, that we had to learn in order to implement the project.

C Outline of the Project

1. Review functional requirements against the business goals
2. Develop a user management system for customers, drivers and managers
   (a) Implement a user login system
   (b) Implement registration for customers
   (c) Implement a timetable for drivers (dashboard)
   (d) Restrict data modification to staff with the required role (e.g. managers)
3. Develop a timetabling service for buses
   (a) Implement bus lines
   (b) Implement bus routes for each line
   (c) Implement bus stops and allow associating routes with these stops
   (d) Build a front-end for route information
   (e) Allow customers to favourite the routes and lines
4. Check the implemented system satisfies the functional requirements outlined by the business goals
   (a) Make any required changes to meet the requirements

Application Design

A ERDs

A.1 Initial ERD

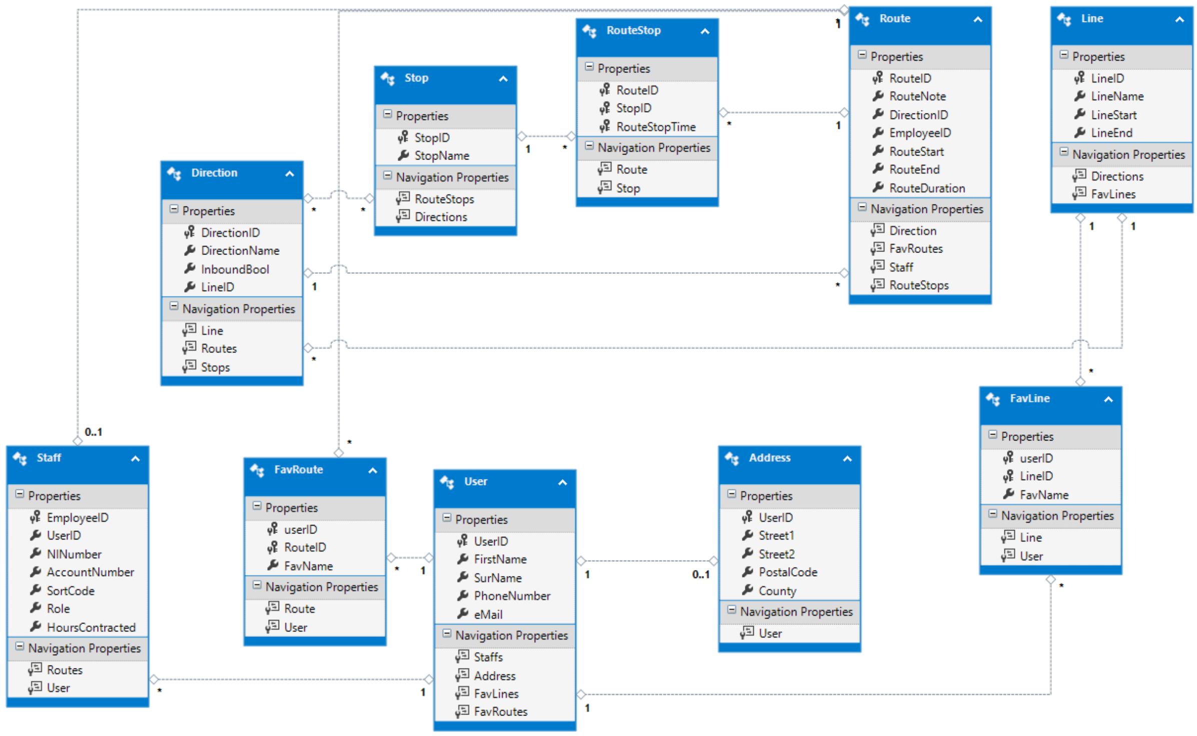


Figure 2.1: Initial ERD, corresponding to what we thought the database would look like

A.2 Final ERD

In order to obtain the final ERD, we started by basing ourselves on what we had previously done for the class diagrams, since we decided to base ourselves in a code-first approach, and started building our models very similarly to what we had in our class diagram from the first submission. After testing and seeing what worked and what didn't with the technology we utilized, our ERD and database design approached and finally settled into the shape represented in the diagram.

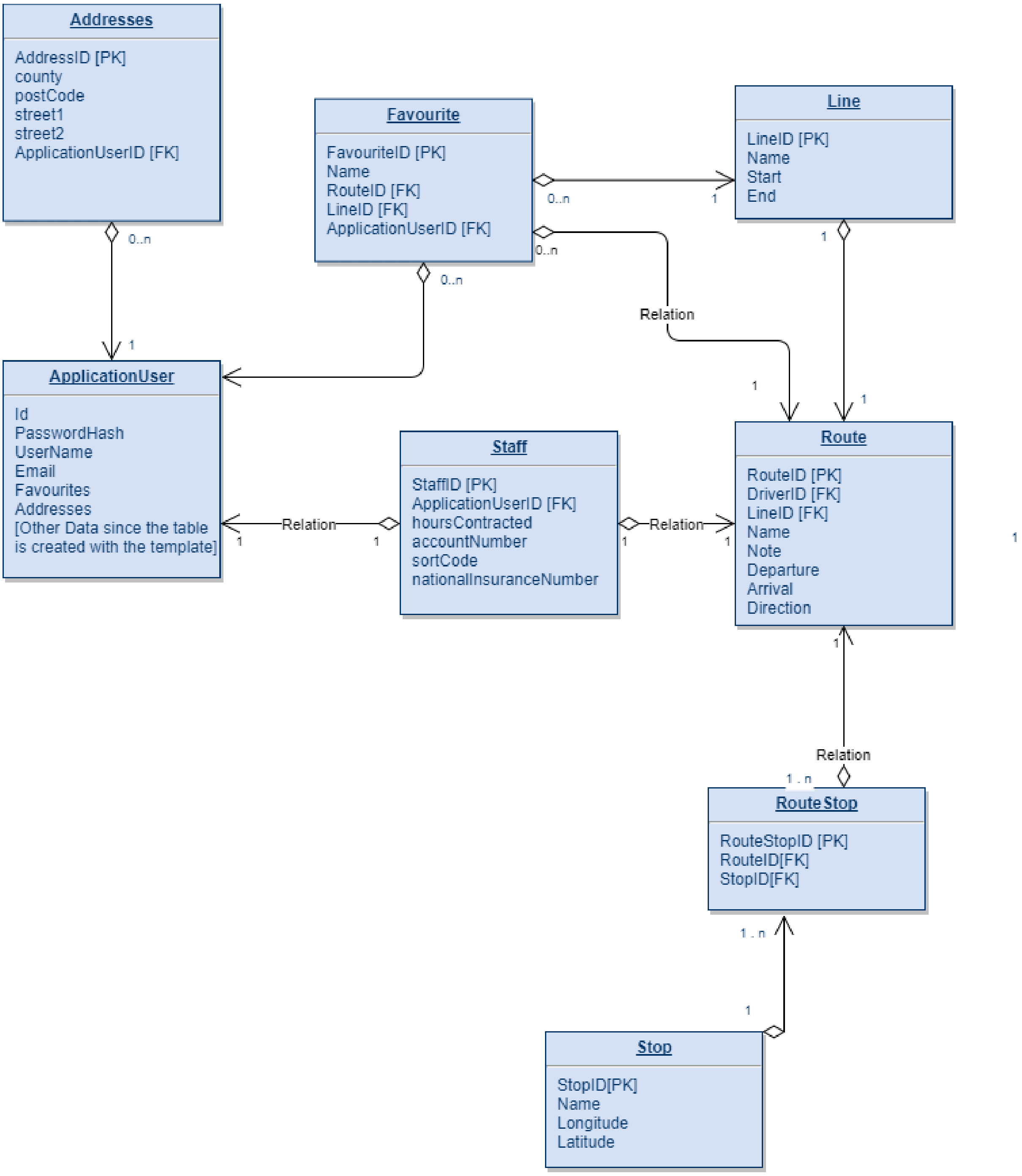


Figure 2.2: Final ERD, obtained whilst making the Models and creating the database

B Class diagrams

B.1 Initial class diagram

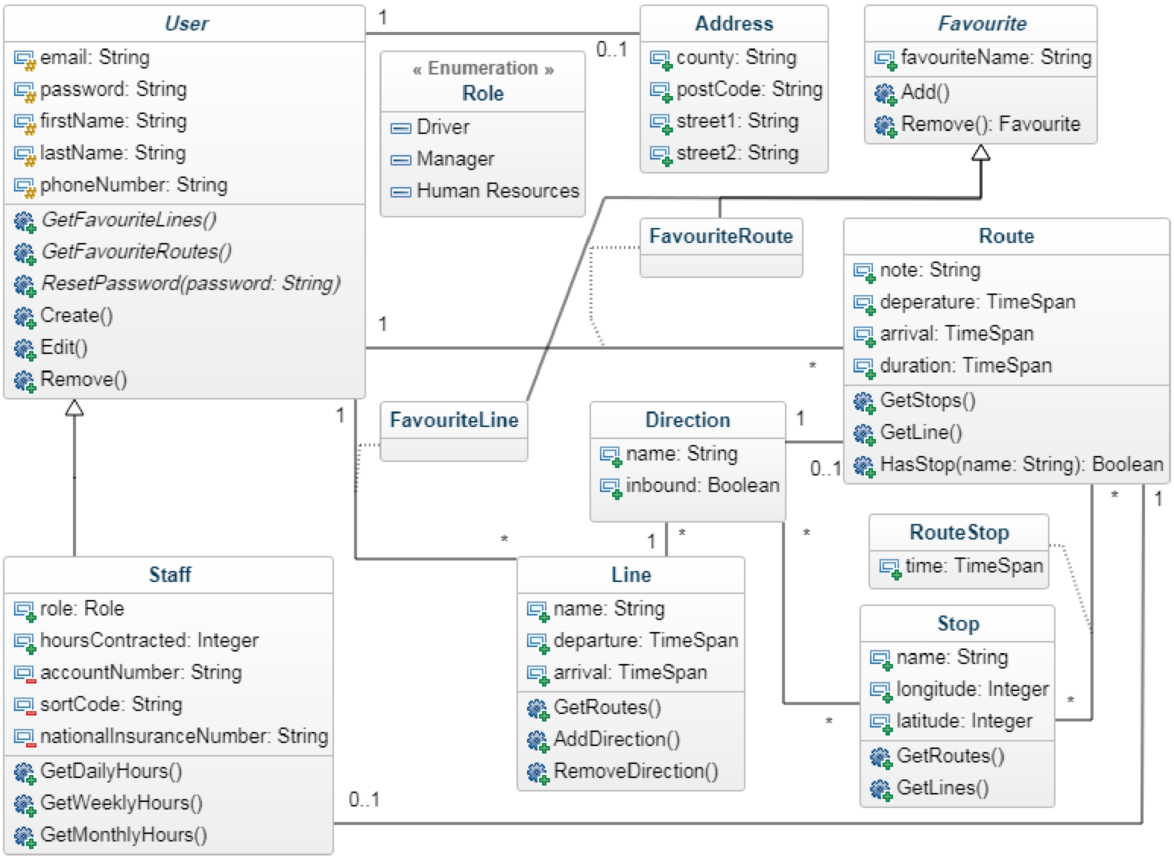


Figure 2.3: Initial class diagram, corresponding to what we thought the classes would look like based on the requirements analysis

B.2 Final class diagram

In order to obtain this diagram, we started by using the initial data previewed in the initial class diagram to create the models that would form our database. After that, we scaffolded the views and controllers, giving us the methods for the classes. The Admin and RoleAdmin controllers were custom developed from scratch in a series of trial and error until we managed to create the functionality that was necessary to us. The attributes that these two classes contain are just instances of classes and interfaces that allow to manipulate the users and roles tables in the database. In reality, that means that these controllers do not have any models associated nor entries in the database.

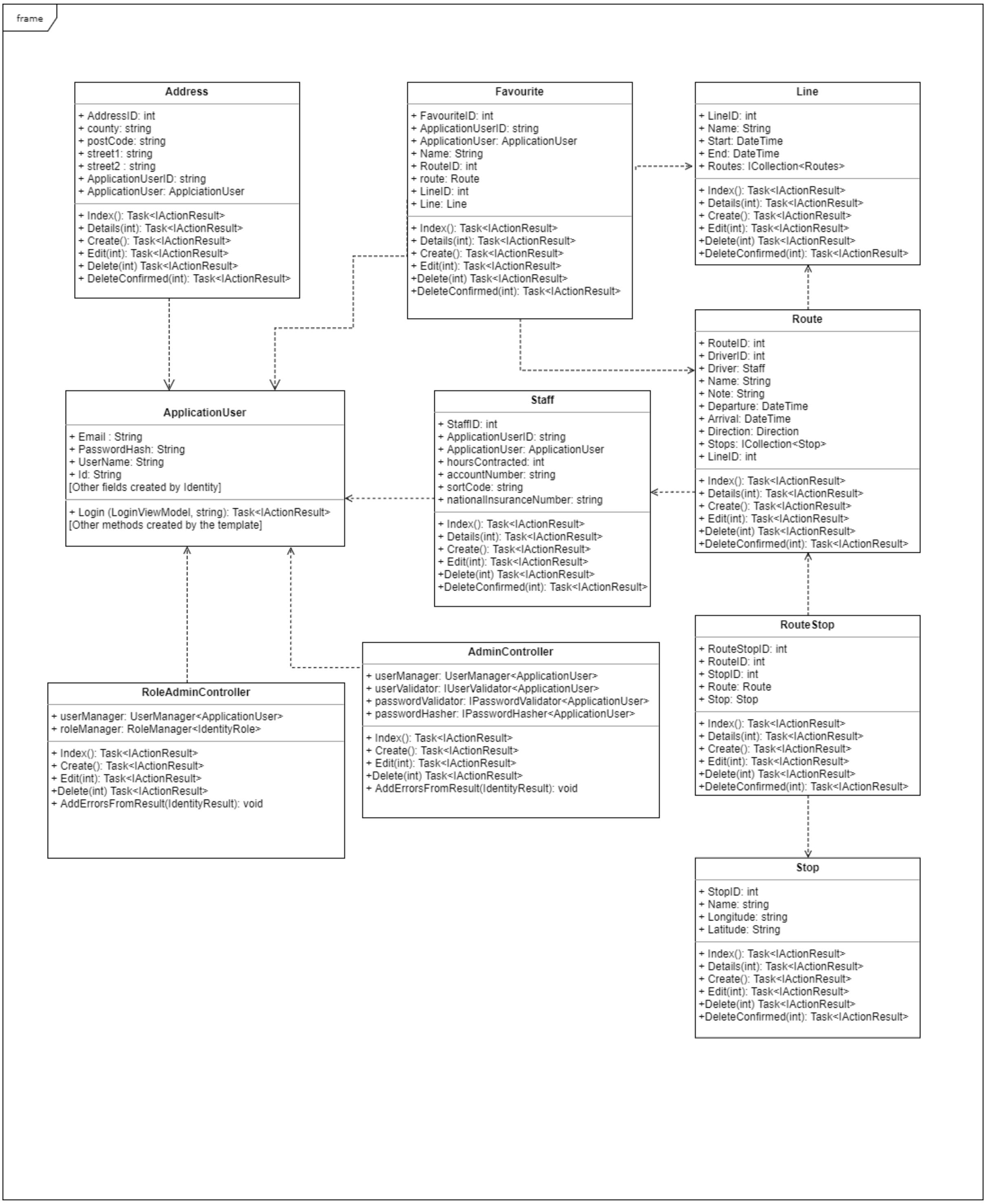

frame
Address
+ AddressID: int
+ county: string
+ postCode: string
+ street1: string
+ street2 : string
+ ApplicationUserID: string
+ ApplicationUser: ApplciationUser
+ Index(): Task<IActionResult>
+ Details(int): Task<IActionResult>
+ Create(): Task<IActionResult>
+ Edit(int): Task<IActionResult>
+ Delete(int) Task<IActionResult>
+ DeleteConfirmed(int): Task<IActionResult>
Favourite
+ FavouriteID: int
+ ApplicationUserID: string
+ ApplicationUser: ApplicationUser
+ Name: String
+ RouteID: int
+ route: Route
+ LineID: int
+ Line: Line
+ Index(): Task<IActionResult>
+ Details(int): Task<IActionResult>
+ Create(): Task<IActionResult>
+ Edit(int): Task<IActionResult>
+Delete(int) Task<IActionResult>
+DeleteConfirmed(int): Task<IActionResult>
Line
+ LineID: int
+ Name: String
+ Start: DateTime
+ End: DateTime
+ Routes: ICollection<Routes>
+ Index(): Task<IActionResult>
+ Details(int): Task<IActionResult>
+ Create(): Task<IActionResult>
+ Edit(int): Task<IActionResult>
+Delete(int) Task<IActionResult>
+DeleteConfirmed(int): Task<IActionResult>
Route
+ RouteID: int
+ DriverID: int
+ Driver: Staff
+ Name: String
+ Note: String
+ Departure: DateTime
+ Arrival: DateTime
+ Direction: Direction
+ Stops: ICollection<Stop>
+ LineID: int
+ Index(): Task<IActionResult>
+ Details(int): Task<IActionResult>
+ Create(): Task<IActionResult>
+ Edit(int): Task<IActionResult>
+Delete(int) Task<IActionResult>
+DeleteConfirmed(int): Task<IActionResult>
ApplicationUser
+ Email : String
+ PasswordHash: String
+ UserName: String
+ Id: String
[Other fields created by Identity]
+ Login (LoginViewModel, string): Task<IActionResult>
[Other methods created by the template]
Staff
+ StaffID: int
+ ApplicationUserID: string
+ ApplicationUser: ApplicationUser
+ hoursContracted: int
+ accountNumber: string
+ sortCode: string
+ nationalInsuranceNumber: string
+ Index(): Task<IActionResult>
+ Details(int): Task<IActionResult>
+ Create(): Task<IActionResult>
+ Edit(int): Task<IActionResult>
+Delete(int) Task<IActionResult>
+DeleteConfirmed(int): Task<IActionResult>
RouteStop
+ RouteStopID: int
+ RouteID: int
+ StopID: int
+ Route: Route
+ Stop: Stop
+ Index(): Task<IActionResult>
+ Details(int): Task<IActionResult>
+ Create(): Task<IActionResult>
+ Edit(int): Task<IActionResult>
+Delete(int) Task<IActionResult>
+DeleteConfirmed(int): Task<IActionResult>
AdminController
+ userManager: UserManager<ApplicationUser>
+ userValidator: IUserValidator<ApplicationUser>
+ passwordValidator: IPasswordValidator<ApplicationUser>
+ passwordHasher: IPasswordHasher<ApplicationUser>
+ Index(): Task<IActionResult>
+ Create(): Task<IActionResult>
+ Edit(int): Task<IActionResult>
+Delete(int) Task<IActionResult>
+ AddErrorsFromResult(IdentityResult): void
RoleAdminController
+ userManager: UserManager<ApplicationUser>
+ roleManager: RoleManager<IdentityRole>
+ Index(): Task<IActionResult>
+ Create(): Task<IActionResult>
+ Edit(int): Task<IActionResult>
+Delete(int) Task<IActionResult>
+ AddErrorsFromResult(IdentityResult): void
Stop
+ StopID: int
+ Name: string
+ Longitude: string
+ Latitude: String
+ Index(): Task<IActionResult>
+ Details(int): Task<IActionResult>
+ Create(): Task<IActionResult>
+ Edit(int): Task<IActionResult>
+Delete(int) Task<IActionResult>
+DeleteConfirmed(int): Task<IActionResult>

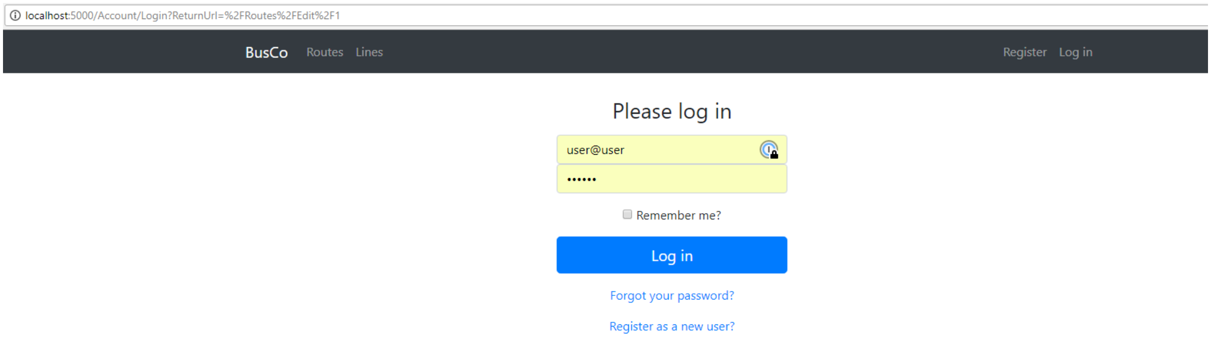
Figure 2.4: Final class diagram, obtained after scaffolding the views and developing the admin controllers

Technical Report

GitHub repository: https://github.com/alexcosta97/web-apps-project

For this project we used ASP.NET Core 2 and an SQLite3 database. This is interacted with the Entity Framework. We also opted for a code-first approach as it was the most intuitive to add user authentication to.

For our user authentication system we used ASP.NET Core Identity. With this system we were able to create a unified user database and add roles to the registered users. These roles were then used to restrict access to specific methods - for example guests can only get the route index and details page, while managers and admins can create, edit and delete routes. If guests try to access these pages they will get redirected to the login page. Normal users will receive an $Unauthorised$ error.



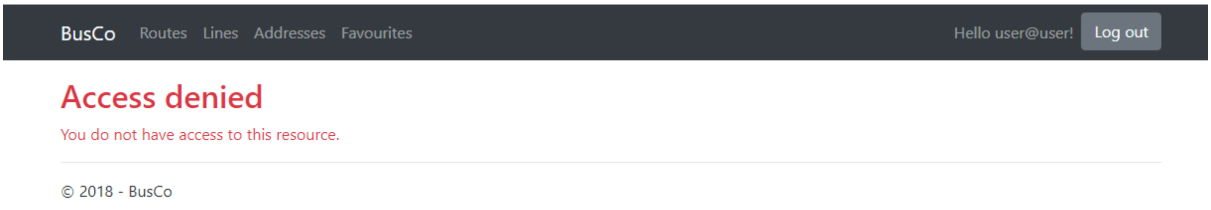
localhost:5000/Account/Login?ReturnUrl=%2FRoutes%2FEdit%2F1

Please log in

user@user

••••••

Figure 3.1: Guest accessing $/Routes/Edit/1$, redirected to login page

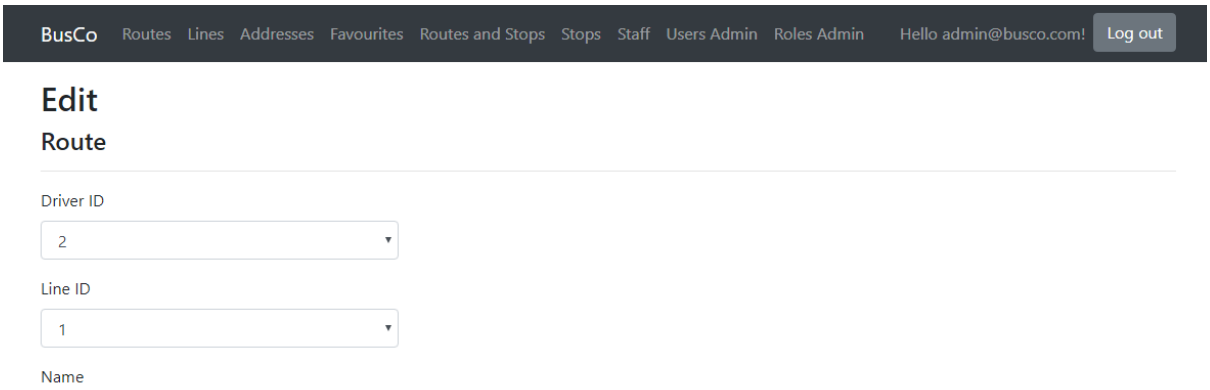


Access denied

You do not have access to this resource.

© 2018 - BusCo

Figure 3.2: User accessing $/Routes/Edit/1$ showing unauthorised error



Edit

Route

Driver ID

2

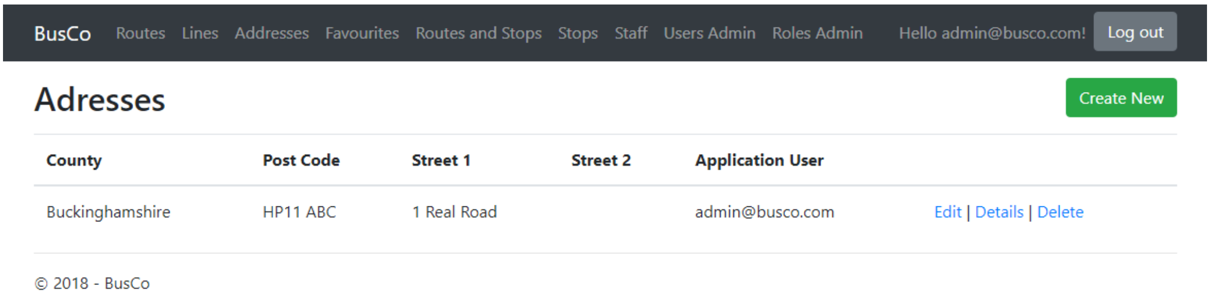
Line ID

1

Name

Figure 3.3: Manager accessing $/Routes/Edit/1$ and getting the edit page displayed

Other functions of the site, such as favourites and addresses are also restricted to those associated with the currently logged in user.

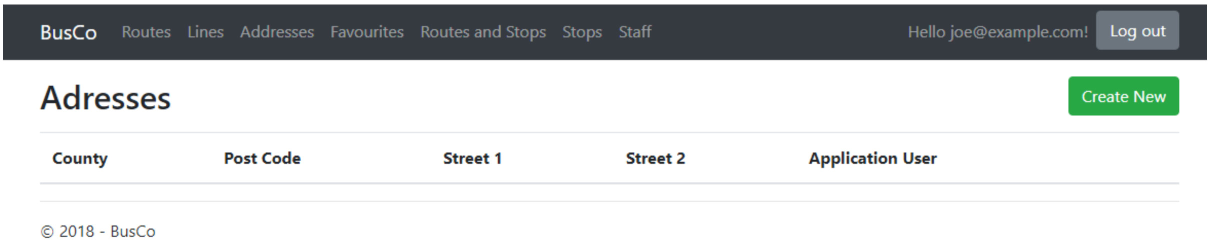


Adresses

County	Post Code	Street 1	Street 2	Application User	
Buckinghamshire	HP11 ABC	1 Real Road		admin@busco.com	Edit | Details | Delete

© 2018 - BusCo

Figure 3.4: User account with an address created by the user



Adresses

County	Post Code	Street 1	Street 2	Application User

© 2018 - BusCo

Figure 3.5: Another user account with no addresses displayed

In order to allow the system administrator manage users and roles in the application, we had to implement our own custom controllers and views. It made for a better customized and saved also the amount of scaffolded code we had to review.

The rest of the views and controllers were scaffolded after we had built the models based on our initial class diagram. After scaffolding the controllers and views, we had to test the components that were built to make sure that the relationships between the elements were defined like we wanted them to and that the data that we wanted to be displayed was doing so correctly.

For this project, we spent most of the time around the authorization processes, making sure that only the right users had access to the right data and functionality, but also around customizing the views and controllers so that names instead of IDs were displayed to the users, when an element from a relationship had to be selected (for example the staff member that will be the driver for a route).

Evidence of Testing

In order to test our application, we made sure that only the right links on the top navigation were available to the right type of users. That means that anonymous users can only access the index and details of the routes and lines, that registered users, on top of that, can also access all the information of addresses and favourites, but only the ones that concern them; that the managers can also access the create, edit and delete functions for routes and lines, as well as assign stops to routes, add stops to the system and add staff to the system. The admins, on top of all that functionality, can also access all the functionality of the Users Admin and Roles Admin sections.

In order to test that, we tried to login with various different types of users and see if the links were displayed to us in the first place, and then try to open those views by manually entering the name of the controller and action that we wanted to access. In all attempts to bypass the system, the system would either ask us to login (in the case of the user being anonymous) or give an access denied message.

The second type of tests that we had to do related to the display of names instead of IDs when the information related to a member of another model than the one being shown in the current view. In order to do that, we opened the different actions of our application for all the controllers and confirmed that the right information was being displayed. If the name was being displayed correctly, that also meant that the relationship in the database was well created, especially inside the Index, Details and Delete controllers, where the whole object that was referenced by the model was sent to the view.