CSC3002 – Development Diary.

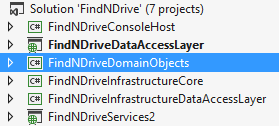
# 11th November – 25th November

## Development Environment

Prior to starting the development process, let me make a note of the development environment I’ll be using:

* Physical device (ASUS Memo Pad HD 7” Android tablet)
* Physical device (LG Nexus 5 Android Mobile Phone)
* Server (My personal computer hosting WCF services)
* Android development (Android Studio)
* WCF Development (Visual Studio 2012 Ultimate)

## Created Visual Studio solution.

To make a start on the project, I created a new Visual Studio 2012. I decided to invest some time into creating a well-defined/structured solution, in which each solution had a well-defined role within the application. As a result, I ended up with the following project structure.

## Created Domain Models:

To make a start on the development process, I started with creating the domain models for my project. The first two models I am going to create are the User and CarShare models since these are the two fundamental types of objects my application will be dealing with. Please see below. The decision to create domain models was influenced by the decision to use Entity Framework together with the Code First approach which is described in the next section.

/// <summary>

/// Represents User entity.

/// </summary>

public class User

{

/// <summary>

/// Gets or sets the user id.

/// </summary>

[DataMember]

public int UserId { get; set; }

/// <summary>

/// Gets or sets the user name.

/// </summary>

[Required]

[DataMember]

public string UserName { get; set; }

/// <summary>

/// Gets or sets the email address.

/// </summary>

[EmailAddress]

[Required(ErrorMessage = "You must provide valid email address.")]

[DataMember]

public string EmailAddress { get; set; }

/// <summary>

/// Gets or sets the first name.

/// </summary>

[DataMember]

public string FirstName { get; set; }

/// <summary>

/// Gets or sets the last name.

/// </summary>

[DataMember]

public string LastName { get; set; }

/// <summary>

/// Gets or sets the date of birth.

/// </summary>

[DataMember]

[DataType(DataType.Date)]

[Column(TypeName = "DateTime2")]

public virtual DateTime DateOfBirth { get; set; }

/// <summary>

/// Gets or sets the gender.

/// </summary>

[DataMember]

public virtual Gender Gender { get; set; }

/// <summary>

/// Gets or sets the role.

/// </summary>

[DataMember]

public virtual Roles Role { get; set; }

}

/// <summary>

/// The car share.

/// </summary>

public class CarShare

{

/// <summary>

/// Gets or sets the car share id.

/// </summary>

[DataMember]

public int CarShareId { get; set; }

/// <summary>

/// Gets or sets the user id.

/// </summary>F

[Required]

[DataMember]

public int DriverId { get; set; }

/// <summary>

/// Gets or sets the driver.

/// </summary>

[DataMember]

[ForeignKey("DriverId")]

public User Driver { get; set; }

/// <summary>

/// Gets or sets the departure city.

/// </summary>

[DataMember]

[Required(ErrorMessage = "Departure city cannot be empty.")]

public string DepartureCity { get; set; }

/// <summary>

/// Gets or sets the destination city.

/// </summary>

[DataMember]

[Required]

public string DestinationCity { get; set; }

/// <summary>

/// Gets or sets the time of departure.

/// </summary>

[Required(ErrorMessage = "You must specify the departure date and time.")]

[DataMember]

[DataType(DataType.Date)]

[Column(TypeName = "DateTime2")]

public DateTime DateAndTimeOfDeparture { get; set; }

/// <summary>

/// Gets or sets the description.

/// </summary>

[Required]

[DataMember]

public virtual string Description { get; set; }

/// <summary>

/// Gets or sets the fee.

/// </summary>

[Required]

[DataMember]

public double Fee { get; set; }

/// <summary>

/// Gets or sets a value indicating whether pets allowed.

/// </summary>

[Required]

[DataMember]

public bool PetsAllowed { get; set; }

/// <summary>

/// Gets or sets the available seats.

/// </summary>

[Required]

[DataMember]

public int AvailableSeats { get; set; }

/// <summary>

/// Gets or sets the participants.

/// </summary>

[DataMember]

public ICollection<User> Participants { get; set; }

/// <summary>

/// Gets or sets a value indicating whether smokers allowed.

/// </summary>

[Required]

[DataMember]

public bool SmokersAllowed { get; set; }

/// <summary>

/// Gets or sets a value indicating whether private.

/// </summary>

[Required]

[DataMember]

public bool Private { get; set; }

/// <summary>

/// Gets or sets a value indicating whether women only.

/// </summary>

[Required]

[DataMember]

public bool WomenOnly { get; set; }

/// <summary>

/// Gets or sets the vehicle type.

/// </summary>

[Required]

[DataMember]

public VehicleTypes VehicleType { get; set; }

}

**TODO: Add more models as the application grows.**

## WCF Services, REST, JSON – Communication established.

I have developed first version of the WCF service. There are currently two service contracts defined for the Android app client to interact with. First is the user service that’s responsible for registering new users as well as logging in existing ones. Please see the screenshot below for more information.

I decided to use REST as the underlying pattern for developing my WCF services. The reason being is that REST uses HTTP as its transport protocol together with HTTP verbs to establish communication between both ends of the system. Another factor which has also influenced my decision is the use of JSON style messages which is described below. WCF makes it easy to create RESTful services by specifying extra attributes and configuring endpoints as it can be seen on the screenshots below.

I also have managed to establish a two-way communication between the Java code that will be used to develop the client application and the WCF service. In order to ensure interoperability between the two different technologies, I decided that both the client and the WCF service will exchange messages in JSON format. JSON stands for JavaScript Object Notation and is an open standard format that uses human-readable text to transmit data objects consisting of attribute-value pairs.

Both WCF service and Android provide native support for serialization and deserialization of objects to/from JSON messages. I have however, decided to use a third party library to help with serialization/deserialization of JSON object on the client side. The library, called GSON has been developed by Google and will provide me with the ability to use Generics and save from writing a lot of repeated boiler-plate code, which I will write about in another section. Below is the order of events prior to sending a message.

Serialization to JSON using GSON

User Object

JSON String(User Object)

Transmit over HTTP

WCF Service

User Object

Deserialization from JSON

As an example of my domain objects being serialized into JSON messages, please take a look at the User object below:

{"DateOfBirth":"\/Date(721785600000+0000)\/","EmailAddress":"participant1@domain.com","FirstName":"John","Gender":0,"LastName":"Doe","Role":0,"UserId":2,"UserName":"JohnDoe"}

The current client application uses LogCat to display the details of the two-way communication taking place and details of objects being transferred over the wire. There is no User Interface developed at this point since my main concerns at this point was to establish the communication.



Figure 1 all operation contracts defined for the UserService

The second service is responsible managing user’s car shares, i.e. retrieving and adding new CarShares to the database.



Figure 2all operation contracts defined for the CarShareService

Example of a WCF service endpoint configured for REST.

<service behaviorConfiguration="Default" name="FindNDriveServices2.Services.UserService">

<endpoint address="" behaviorConfiguration="webHttpBehavior" binding="webHttpBinding" bindingConfiguration="webHttpTransportSecurity" contract="FindNDriveServices2.Contracts.IUserService" />

<host>

<baseAddresses>

<add baseAddress="https://findndrive.no-ip.co.uk/Services" />

</baseAddresses>

</host>

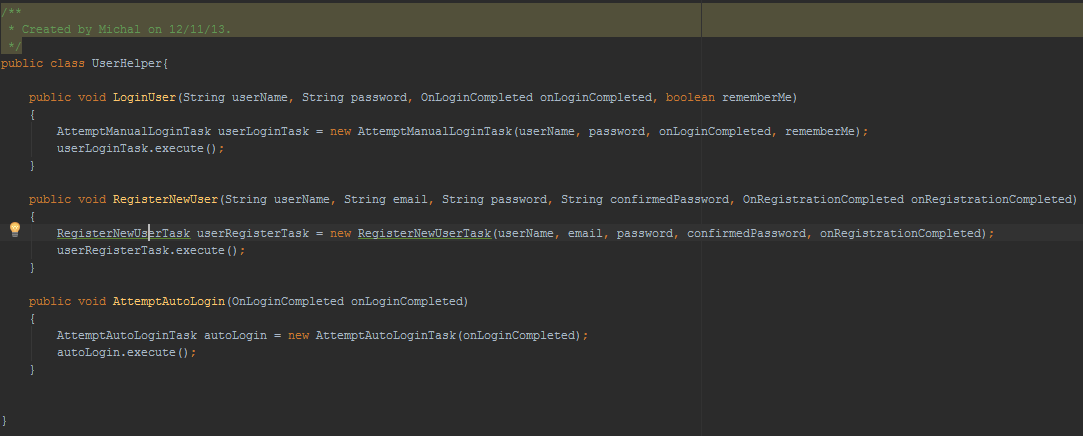
</service>

**TODO:** **Code is still to be properly refactored and commented. UserService and CarShare service methods to be partially redesigned and possibly relocated to another service to enable further separation of concerns.**

**TODO: Make it possible for the user to log in to the system using either their username or email address.**

**TODO: Create a basic User Interface for the client application.**

**ISSUE: I’m having major issues with converting dates and times to and from the WCF service. After a lot of research, I found that when sending a serialized Date & Time object, WCF will only accept it in the following epoch format: "/Date("+epochlong+")/" which means I’m going to have to write my own WCF date/time converter. Medium priority, will investigate later.**

Android Client App – Basic User Interface.The first version of the Android app supports registering and logging in of the user. The design is still very basic. My main focus was to get as much of the client’s architecture developed as possible. Since Android does not allow any networking activities to be performed on the main thread, every single interaction of the service had to be moved to a separate class which inherits from an async activity class which can be compared to a thread which performs its task in an async mode.

To make interaction with my WCF service easier, I have developed a system where a network activity is started in async mode and uses a call-back method to return back to the class which originally called it. Please see the diagram below for more information. Every activity which uses a network task implements an interface which is passed to the network task. The network task can use the instance of this interface to call a call-back method on the original activity which triggered it. This helps me to avoid passing an instance of the entire activity to the network task which would be very expensive and would show a bad design.



Network Task completed its job, call-back method is called on the Login Activity

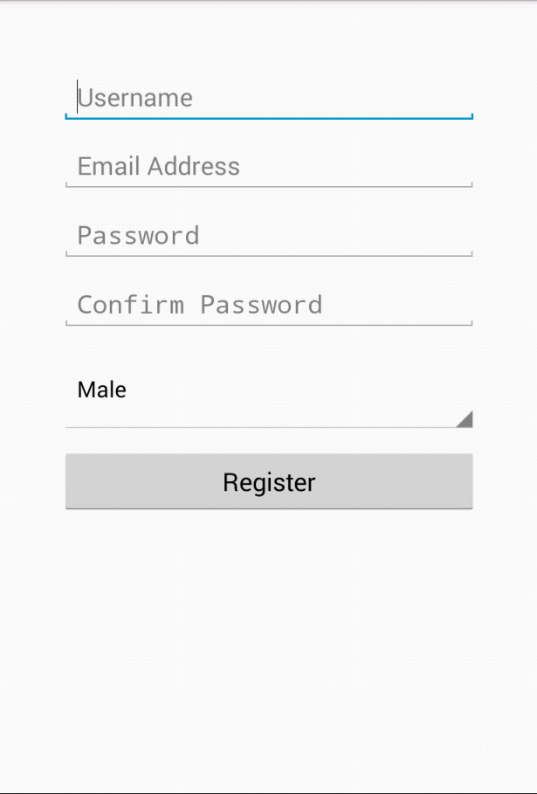
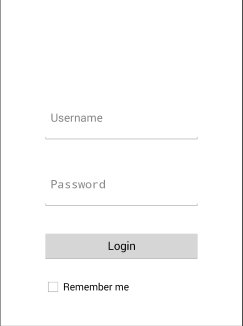
Login Network Task spawned – contacts WCF service in the background, independent of main UI thread.

Login Activity Created

Application starts

Main UI thread

In Android, every screen that the user interacts with is known as an Activity. So far, I have developed a UserHelper class which acts as a glue between activities and the Networking tasks. This means that the UserHelper calls whatever networking task an activity wants to perform and invokes it. How does the async task inform an activity about a task which it just finished? Every time an async task is called, it gets an implementation of a callback interface passed in. This allows the async task to call a method on the activity class once the task is finished without having to pass an entire activity to the async task class.



**TODO:** **Revise the layout to make sure it scales properly on screens of various sizes and in both orientations.**

**ISSUE: Each time the device is rotated and its orientation changes, Android tears down the entire activity and goes through the process of creating the activity from scratch again. This creates the need to develop a system whose responsibility will be to save the state of each activity before it’s torn down and reload its state after its recreated. Without such a system in place, I am currently experiencing nullreference exceptions when the Activity attempts to reload information which has not been saved or reinitialized and is therefore null. This crashes the application but is still a rather low priority issue which will be addressed later.**

## User Registration, Logging in & Security.

As you can see on the above screenshot, the user registration asks the user to input only the most necessary information required to identify each user and allow them to safely log in. The decision to make the user registration quick and simple comes from taking a look at the application from the user’s perspective. Most users are not willing to spend more than a minute on setting up a new account, for this reason the amount of information required to create a new account should be kept to a minimum. My intention is to provide an in-application facility to allow the users to fill in the rest of their profile information at a later date, such as date of birth, whether they are a smoker etc.

This was perhaps one of the most difficult tasks associated with this project and took me at least 4 days to implement properly. In order to provide a secure log in mechanism for users, I decided to integrate the WebMatrix.WebSecurity membership provider into my WCF service. This particular security mechanism uses my existing database and creates a separate set of tables for encrypted user passwords and user role information.

Integrating the WebMatrix.WebSecurity membership provider was quite a challenging task. It took a lot of research and experimenting with the web.config file to achieve the desired result. One of the main changes that I had to implement to the web.config file is the inclusion of WebMatrix.WebSecurity membership into the WCF service:



The system worked perfectly for user registration. Users were able to register properly and the WebMatrix.WebSecurity mechanism was able to store encrypted passwords successfully. Unfortunately not everything went according to plan. One of the main problems that I encountered during the implementation of this security mechanism into my WCF service was a nullreference exception each time I tried to validate user’s credentials.

**ISSUE: WebMatrix.WebSecurity throws nullreference exception while attempting to log user in. Blocker, must investigate immediately.**

After almost two days of debugging and researching the possible causes, I was not getting any closer to solving the problem. This was until I realised that my WCF service was still being hosted inside a local service host which is a console application. Since console application ignores web.config files and uses app.config instead to store its configuration data, I realised the WebMatrix.WebSecurity mechanism was not being initialised properly and this is where the nullreference exception was coming from.

The only solution I could possibly think of was to migrate all of my web services and host them inside IIS which understands the configuration data stored in the web.config file. The process of migrating from self-hosted service to an IIS host is described in one of the sections below.

After migrating all of my WCF services to IIS, the security mechanism was being initialised properly and the nullreference exception did not longer crash my WCF service. From that point on, I was able to successfully authenticate and log users in to the system by simply executing the following statement at the top of my service’s login method:

if (!validatedUser.IsValid || !WebSecurity.Login(login.UserName, login.Password))

{

validatedUser.ErrorMessages.Add("Invalid Username or Password.");

validatedUser.IsValid = false;

}

The WebMatrix.WebSecurity mechanism worked flawlessly from that point on. For illustration purposes, below is the block of code taken from the RegisterUser method responsible for adding new user’s encrypted password to the database:

WebSecurity.CreateUserAndAccount(register.User.UserName, register.Password);

## ServiceResponse, DTO objects, GSON.

In order to provide a consistent API throughout the numerous exposed service contracts, I decided to create a consistent and generic ServiceResponse. This Service Response object acts as a secure wrapper for data sent back from the WCF service to the client, which is used to indicate whether a particular call resulted in a success or failure. This proved to provide a far richer API than the alternative of throwing Exceptions. When a request is made to the WCF service, each response from the service is wrapped inside a Service Response objects which contains the following information:

- **ServiceResponseCode:** Indicates whether the service call resulted in success or failure. As an example, the user service might return a failure as a service response code if the user provides incorrect login credentials. These initial status codes code be expanded upon within future sprints. For instance the client could then use this information to potentially invoke an operation retry, or provide some sort of business-level compensation logic, for instance if a business-level transaction was being performed.

- **ErrorMessages:** An optional list of error messages that contain the useful information as to why an operation did not succeed (For instance validation reasons). Within the current architecture, this information was used within the presentation layer to provide the user with extra information as to why an action did not succeed.

- **Result:** This property was provided to operations which were not defined as ‘fire and forget’, that is to say an operation which provided a return value. The result was of type T, where the type of T depends on the WCF service called, ie. Project Service might return a result type of ‘Project’

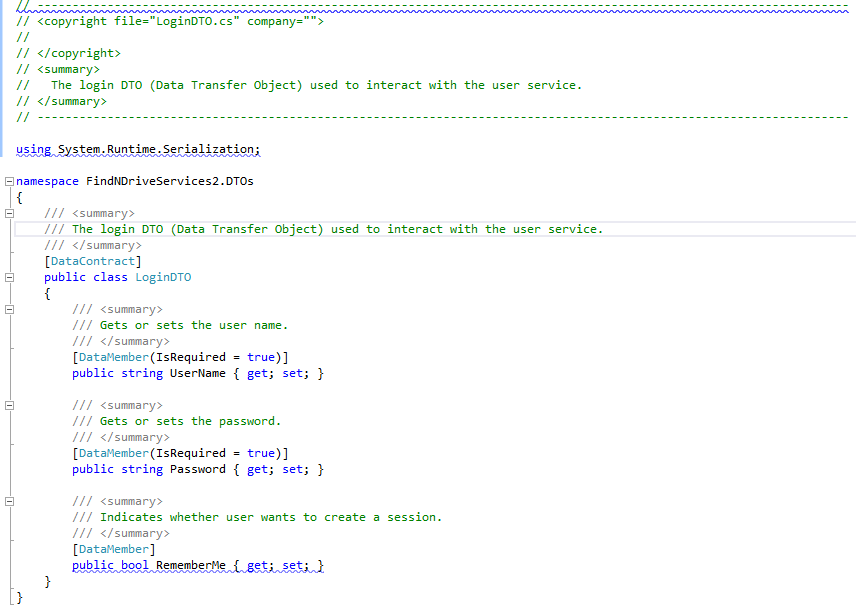
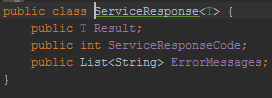
Consider the screenshot below which contains most of the code from the RegisterUser method in the UserService. It returns the ServiceResponse object back to the client to indicate whether user has registered successfully or not.

Figure 3 Piece of code from the RegisterUser method illustrating the usage of the ServiceResponse object.

With the ServiceResponse Result being a generic type, it provides a very convenient way of returning any type of object back to the Android client without having to create a separate ServiceResponse for User, CarShare etc.

The GSON library which I talked about in one of the earlier sections provides me with a very similar way of parsing the generic Result into a specialized object without creating a separate   
ServiceResponse object on the client side for User, CarShare etc.

This means that GSON allows me to create a single ServiceResponse object on the client side which looks as follows:



Here’s how it works. When calling the WCF User Service, I know that the service will return a   
ServiceResponse object with the Result of type User. I can then tell GSON that the Result of type T is going to be of User type. This is done by passing a new TypeToken of type User to the instance of the GSON object.



This will essentially allow me to re-use the generic ServiceResponse object for all the responses from the WCF service. It’s very important to note that this approach works for arrays, list as well as single objects. Additionally Data Transfer Objects (DTOs) were provided as a means of decoupling a complex

Domain model from a specific operation, as a domain object could potentially contain many irrelevant fields to the required operation. This design decision lead to improved testability, and aided in providing a service layer which could be shared potentially by numerous different projects, and under various contexts as the services were not domain-specific, due to the abstraction of DTOs. This decision also benefited a decoupling such that any future updates to domain models could be made without breaking existing API contracts, and therefore avoiding possible regression issues.

The example of the DTO object below shows login information in the form of username, password and the remember me Boolean passed in from the client application to the WCF service.

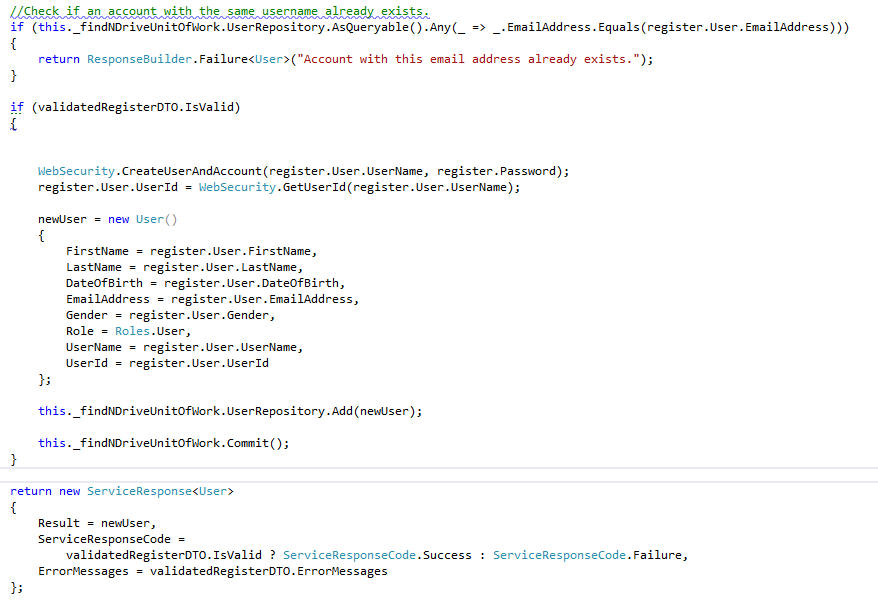
The WCF service then uses this information to validate user’s credentials and log the user in or display an “invalid username or password” message.

Figure 4LoginDTO containing informaion used by the WCF service to authenticate the user.

WCF Service Validation.As well as validating data on the client side, I have implemented logic to perform additional validation on the WCF side as well. This is serves two purposes:

* Ensuring no incorrect data is saved into the database.
* If there is another client developed by someone else attempting to call the service, we do not want to rely on their validation logic.

Validation is performed on data annotation provided by entity framework. This provides a reliable quick and easy way of checking whether the data passed in from the client meets our criteria. As an example, consider the below screenshot:



## Entity Framework and the repository patterns.

For my Data Access Layer (DAL), I decided to proceed with Entity Framework and the Code First approach.

Entity Framework is an object relation mapper that sits on top of the ADO.NET framework and provides developers with the ability to think of data stores in terms of entities and business objects. In conjunction with the generic repository and unit of work patterns that was employed within the project, the application features a robust and flexible data access layer. This not only follows the industry standards but also provides a level of abstraction that allows developers to concentrate on the application logic. This meant that developers did not need to worry about hard-coded SQL dependencies or a particular data engine/storage schema.

In fact, it makes the data access layer is so flexible that all of the queries to retrieve data from the database can be written using LINQ statements. These statements would otherwise be much more complex, error prone, and tightly coupled to an implementation if developed using standard SQL. Adding information to the database is just as easy as retrieving it. Using the ‘Add’ and ‘Commit’ methods exposed by the unit of work and repository classes allows developers to insert very complex objects into the database using no more than two lines of code. For example, when adding a new project to the database, we first had to add this project to the project repository by calling projectRepository.Add(newProject) and then commit the changes to the database by calling unitofwork.commit().

Using the code first approach provided by the Entity Framework allowed us to develop the datastore and relationships between entities gradually by means of writing classes that represent domain objects ie. User or Project entities. Using one simple command, we were able to generate and alter the schema of our database and populate it with initial data automatically.

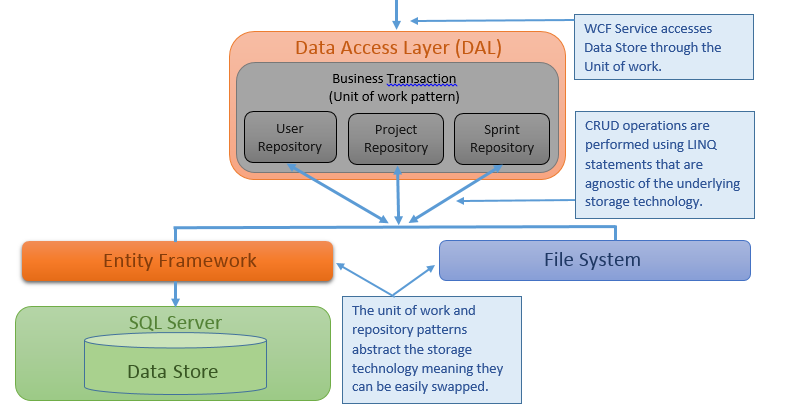
Another great feature of Entity Framework is the ability to make rapid changes to the database model without data loss simply by executing the “Update-database command” in the Package Manager Console in Visual Studio.

The Update-Database will regenerate the database from scratch if needed and repopulate it with seed data to allow for quicker testing. This means that at any time during the course of the project development, I can delete my database and regenerate it with a single command with all the seed data being automatically pushed to the database for me.

For my Data Access Layer (DAL), I considered using the repository and unit of work patterns. After a few hours of weighing the pros and cons, I concluded that the extra development time spent on implementing the above patterns will pay off in the future when the following scenarios are considered:

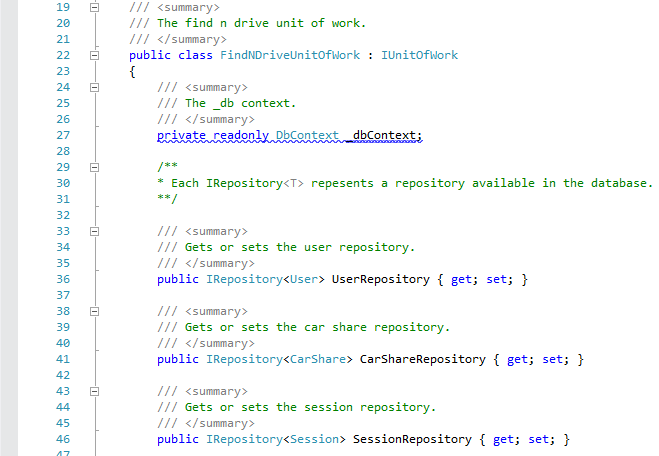
* If I ever decide to change the database technology, doing so will be extremely easy.
* Additional web services can re-use existing repositories.
* Repositories for new domain objects can be created easily.
* Unit of work provides the necessary CRUD operations.

Implementing the patterns will also provide an additional layer of abstraction between the Data Access Layer and the business logic layer. Below is a diagram that I created which illustrates how the two patterns fit with the Data Access Layer.

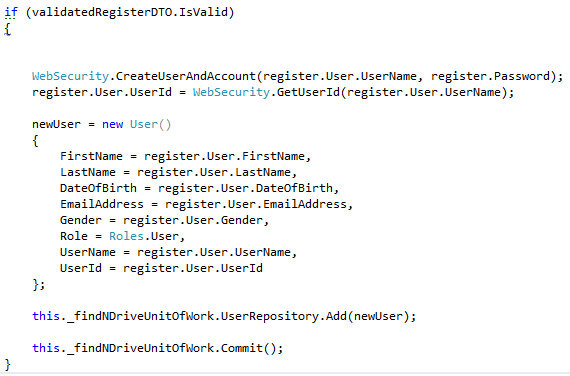


The Unit of work class as shown below exposes the repository objects for each domain object. Interaction with the database i.e. adding, updating and deleting records will be carried out in the following manner:

Unitofwork.Repository.Add(new object); <- which adds a new object to the database.



The below example shows the process of adding a new user to the database with the help of entity framework, repository and unit of work patterns.



**TODO:** **As more domain objects and WCF services are being created, more repositories will need to be added to the unitofwork class.**

## HTTPS, Data encryption, IIS.

Security was one of my paramount concerns from the very beginning. For this reason, I decided to implement end-to-end transport security in the very early stages of the development process to ensure all data transmitted over the wire is encrypted. The need for data encryption in this application comes from a lot of sensitive user data being transferred between the client application and the WCF service. The Hypertext Transfer Protocol Secure (HTTPS) protocol will provide all the necessary encryption for my application. The protocol simply lays the HTTP message on top of the SSL/TLS protocol, thus adding the security capabilities of SSL/TLS to standard HTTPS communications.

To enable HTTPS, the WCF service had to be reconfigured to use the HTTPS protocol.

The Web.config file had to be modified to include the following pieces of code:

1. Enable transport security in System.ServiceModel by including the below code:

<webHttpBinding>

<binding name="webHttpTransportSecurity">

<security mode="Transport">

<transport clientCredentialType="None" />

</security>

</binding>

</webHttpBinding>

1. Modify the default service behaviour to enable HTTPS and disable plain HTTP.

<serviceBehaviors>

<behavior name="Default">

<serviceMetadata httpGetEnabled="false" httpsGetEnabled="true" />

<serviceDebug includeExceptionDetailInFaults="true" />

</behavior>

</serviceBehaviors>

1. Change the base address of every service to inclide https

<service behaviorConfiguration="Default" name="FindNDriveServices2.Services.UserService">

<endpoint address="" behaviorConfiguration="webHttpBehavior" binding="webHttpBinding" bindingConfiguration="webHttpTransportSecurity" contract="FindNDriveServices2.Contracts.IUserService" />

<host>

<baseAddresses>

<add baseAddress="https://findndrive.no-ip.co.uk/Services" />

</baseAddresses>

</host>

## </service>

The next stage was to host the WCF service in IIS by creating a new website, creating self-signed SSL certificate and binding that certificate to the newly created site.

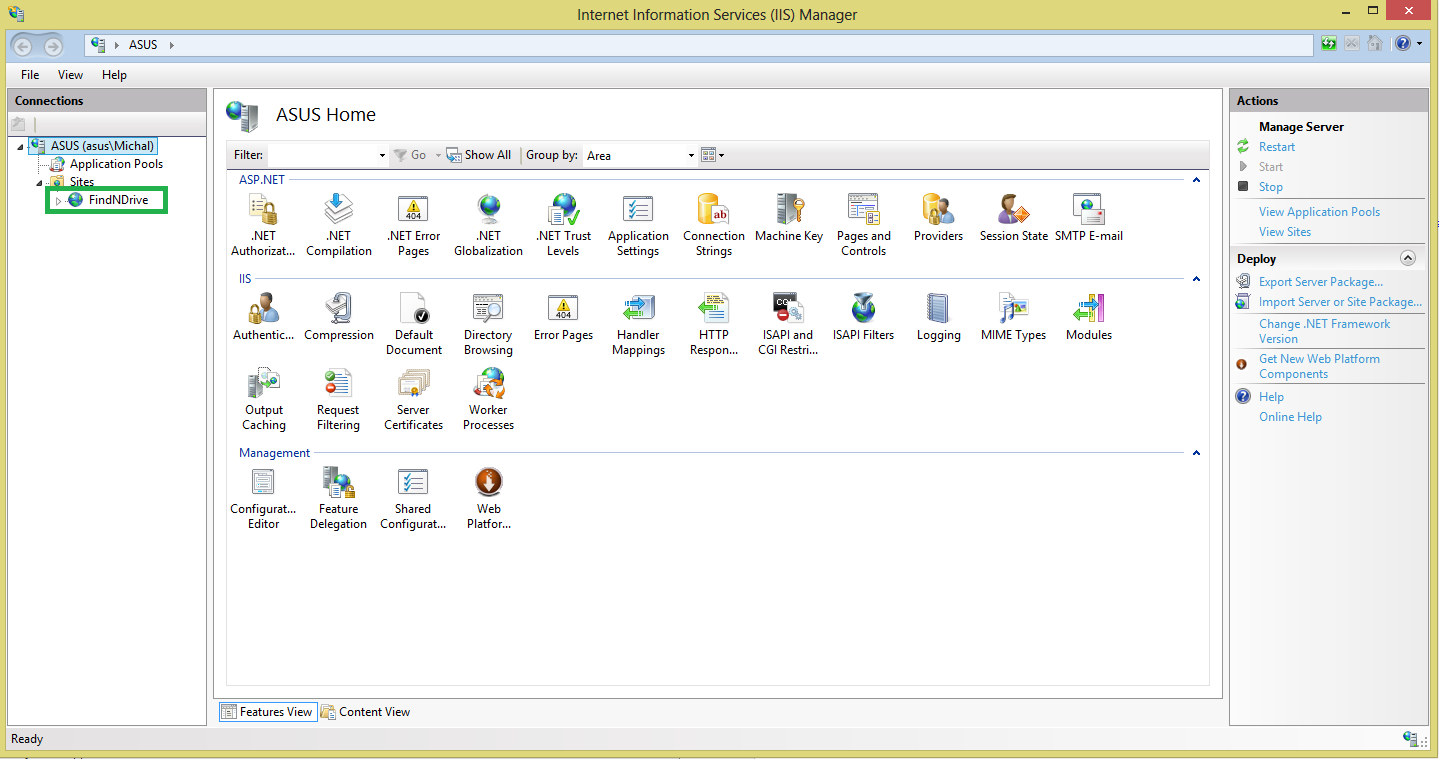


Figure 5FindNDrive website created and hosted in IIS. Please see the green rectangle in top left.

## 

Figure 6New SSL certificate created for the FindNDrive website.

Finally, all that was left was to bind the SSL certificate to the website.

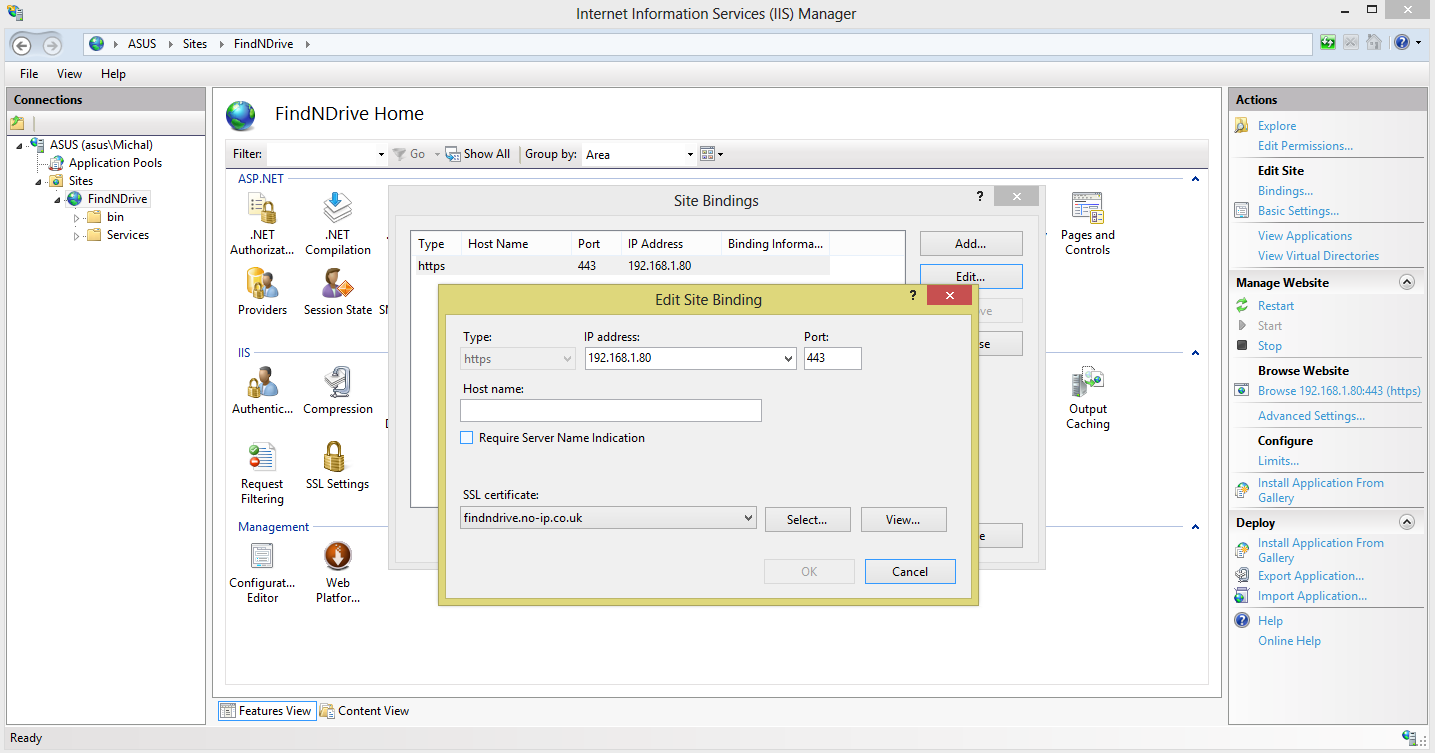


Figure 7Binding SSL certificate to FindNDrive website.

At this point, the FindNDrive WCF services hosted in IIS were HTTPS enabled while all incoming HTTP connections were refused.

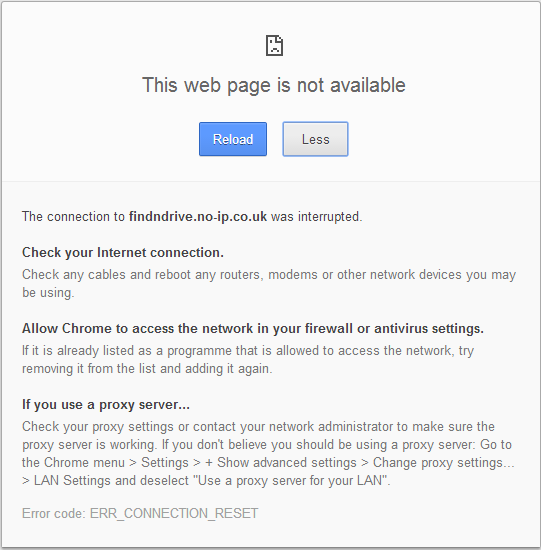


Figure 8FindNDrive now refuses all HTTP connection. HTTPS is the only allowed method of connecting to the service.

However, I encountered the following problem while attempting to connect the android client application to the service via the HTTPS protocol:

**SSLPeerUnverifiedException**

After a lot of research, I realised that this is a very common issue with Android and self-signed certificates created in IIS. The reason for this exception is that the SSL certificate which the android client receives from the WCF service is not trusted since it was not created by a known CA. This can be resolved in two ways, by purchasing an SSL certificate from known CA, or by forcing the Android client to accept all SSL certificates it receives whether trusted or not. Due to budget constraints, I chose the latter which is perfectly fine for development and testing purposes and can be easily replaced by proper implementation at a later date.

In order to force the Android client to accept all SSL certificates, I have written the SSLSocketFactory as it can be seen on the screenshot below:

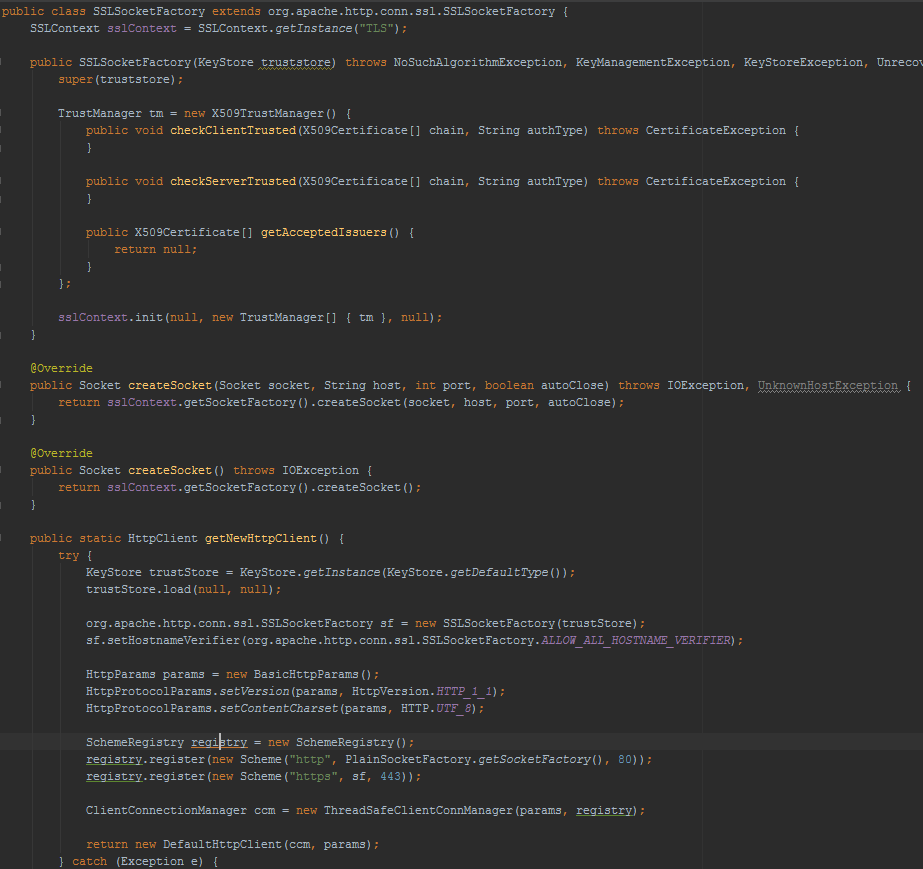


Figure 9 SSLSocketFactory, developed with the help from: http://madurangasblogs.blogspot.co.uk/2013/08/avoiding-javaxnetsslsslpeerunverifiedex.html

## DDNS, Port forwarding.

As I mentioned earlier, the development server has been set up on my personal computer. For the purposes of testing, I will be accessing the server through the client application from outside of my home network i.e. via my networks’ external IP address on many occasions. In order to make this possible, I had to solve a lot of problems such as port forwarding inside my home network, lack of static IP address to access the service from external network and lack of domain name. Below is a detailed description of how I addressed each of the issues I was facing:

* Port forwarding: In order to route the incoming HTTPS traffic to the server, I had to create a port forwarding rule on my router to direct all the incoming traffic on port 443 (default HTTPS port) to my home server with the IP address 192.168.1.80.
* To account for lack of static IP address, I decided to sign up for a Dynamic DNS (DDNS) service which uses a piece of software on my personal computer to contact the DDNS Company and provide them with my most recent address. The same DDNS has also provided me with a domain name [www.findndrive.no-ip.co.uk](http://www.findndrive.no-ip.co.uk) to allow the client application to contact the WCF service irrespective of any IP changes. This is because at the time of the client application making a request to the WCF service residing on my personal computer, the above domain name with the help of DNS servers of the DDNS Company is always resolved to the most recent and up-to-date IP address of my computer. This makes it virtually impossible for the client application to use an outdated address for a WCF service request.

## Session management – first version.

To provide the users with the ability to stay logged in for an extended period of time, I had to implement my own session management system which provides secure authentication using a session file and does not require username and password.

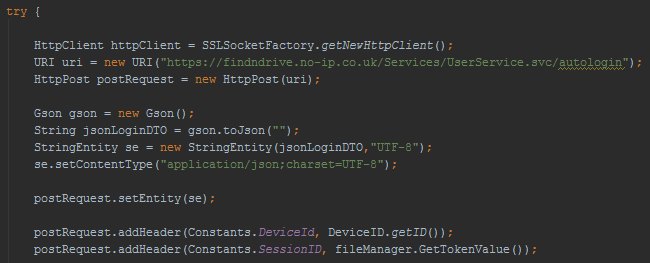
The decision to develop my own session management mechanism comes from the lack of other available solutions for the Android/WCF mixture. REST, being a stateless architectural style, it does not support proper session management. For this reason, I want to be able to create a virtual session between the client application and the WCF service and give users the illusion of the application keeping them logged in. While users think a session is running in the background, the application will sent session cookie file to the WCF service which the latter will then authenticate against information currently stored in the database.

The first, basic version of this system took about 7 days to build and test. It works for the following scenarios:

* User logs in without ticking the Remember Me box. Application establishes temporary session which is invalidated on application exit and expires after 30 minutes of inactivity.
* User logs in and ticks the Remember Me box. Application establishes permanent session set to last 14 days. Exiting the application does not invalidate the session. When the user start the application again, it auto-logs the user in using the session file created during the initial login.

Please refer to the session management diagram for a detailed breakdown of how the session management system functions. For session management on the WCF side, I developed my own session manager called SessionManager which is encapsulated inside each WCF service. SessionManager extracts session information from the HTTP header of an incoming request and validates it against records currently present in the database.

### Android client – session information injection.

Below is a piece of code showing how the client injects session information from a cookie file into the HTTP header.

### The SessionManager class.

Below is a piece of code responsible for validation of the current session.

/// <summary>

/// The validate session.

/// </summary>

/// <returns>

/// The <see cref="bool"/>.

/// </returns>

public bool ValidateSession()

{

if (WebOperationContext.Current != null)

{

var incomingSessionId = WebOperationContext.Current.IncomingRe quest.Headers[Constants.SessionId];

var incomingDeviceId = WebOperationContext.Current.IncomingRequest.Headers[Constants.DeviceId];

var randomId = WebOperationContext.Current.IncomingRequest.Headers[Constants.RandomId];

int userId = GetUserId(incomingSessionId);

if (userId == -1)

return false;

var savedSession = \_findNDriveUnitOfWork.SessionRepository.Find(GetUserId(incomingSessionId));

if (savedSession != null)

{

if (savedSession.SessionType == SessionTypes.Temporary)

{

if (randomId != savedSession.LastRandomId)

{

return false;

}

}

if (!incomingSessionId.Equals(savedSession.SessionId))

return false;

var encryptedId = EncryptValue(incomingDeviceId);

if (!savedSession.LastKnownId.Equals(encryptedId))

return false;

var result = DateTime.Compare(DateTime.Now, savedSession.ExpiresOn);

if (result > 0)

return false;

if(savedSession.SessionType == SessionTypes.Temporary)

RefreshSession(savedSession);

}

else

return false;

}

else

return false;

return true;

}

Below is the auto login operation contract from my user service showing how the user it authenticated using session file.

/// <summary>

/// The auto user login.

/// </summary>

/// <returns>

/// The <see cref="ServiceResponse"/>.

/// </returns>

public ServiceResponse<User> AutoUserLogin()

{

User loggedInUser = null;

if (this.\_sessionManager.ValidateSession())

{

var userId = this.\_sessionManager.GetUserId();

if (userId != -1)

{

loggedInUser = this.\_findNDriveUnitOfWork.UserRepository.Find(userId);

}

}

return new ServiceResponse<User>

{

Result = loggedInUser,

ServiceResponseCode = (loggedInUser == null) ? ServiceResponseCode.Failure : ServiceResponseCode.Success,

};

}

In order to keep the WCF operation contracts short and functionally cohesive, I decided to split the login mechanism into two separate contracts. First one being the auto login mechanism which is called when the client application attempts to log the user in automatically by providing the service with session file in the HTTP header. The second operation contract is the standard manual login method which validates user’s credentials against those stored in the database.

It’s also important to note that the content of the session file is encrypted since all messages are passed via HTTPS.

**TODO:** **Investigate using proper cookie files instead of a custom cookie-like file.**

**TODO: Streamline the process of adding information to the HTTP header on the client side.**

## Application File Manager

To help with the process of managing applications’ files such as the session file described above, I created my own file manager which the client application will use to perform all of its required file operations. At the moment, the functionality built into the file manager checks for the existence of the application folder in the device’s /sdcard directory and also checks for existence of the session cookie file. If application folder is not found, one is created. The file manager is also responsible for writing content to session cookie file and reading content from it.

# 25th November – 09th December

## Session Management – fixed bugs, more advanced.

The previous version of the session management did not account for one specific scenario. If a user is in the middle of a temporary session and the application is suddenly terminated by either the built in or a third party task manager, the client application has never had a chance to contact the server to invalidate the session. This resulted in the user being able to automatically re-login despite not having selected the Remember Me button before.

Below is a solution which I have developed to overcome this problem:

1. When the client application is started, a new random string is generated using JAVA’s UUID function. I’ll refer to it as UUID from now on. This UUID is kept in memory until user exists the application.
2. When the user logs in without selecting the Remember Me button, this UUID is added to the service call and is saved in the database.
3. Every subsequent call to the WCF service which uses a temporary session and requires authentication compares the UUID in the HTTP header against the one stored in the database.
4. When the user who has been logged in temporarily kills the client application the UUID is wiped from memory and a new one is created when the client application is restarted. This time, when the client application will attempt to auto-login the user, the two UUID’s will not match and the WCF service will return an error telling the client application to ask the user to log in manually.

## Fixed Date & Time conversion.

I finally managed to find a fix to the WCF date/time issue where the date & time for each CarShare object had to be serialized and deserialized in a very ugly fashion. I wrote my own WCFDateTimeHelper class which has two methods in it:

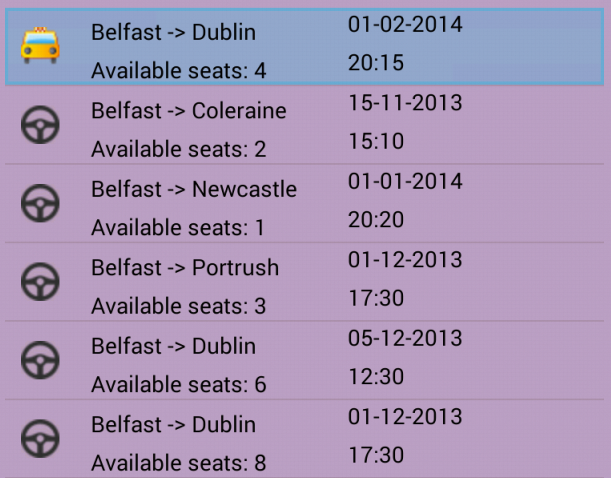
* public static Date parseWCFDateTimeString(String wcfDate) – takes a date & time string from WCF service and converts it to a JAVA Date object.
* public static String ConvertToWCFDateTime(Date date) – takes a JAVA date object and converts it to a WCF compatible data & time epoch string.

With such a solution in place, I can easily convert the data back and forth before sending it to the WCF service and after receiving it from the WCF service by using the static methods that the WCFDateTimeHelper exposes.

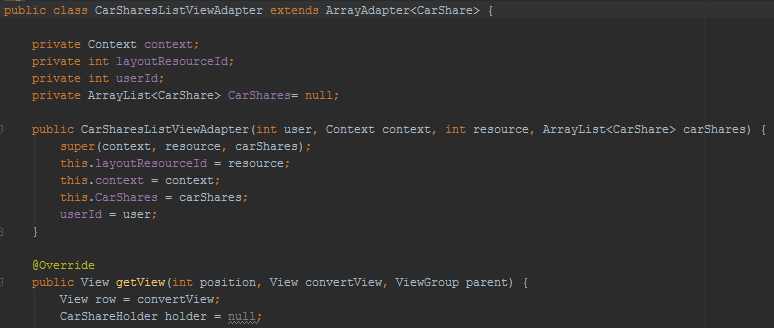
## My Car Shares List

Finally got the chance to develop some UI on the client side. Until now, the user home screen was empty and did not show any useful information to the user. I have now added a listview which uses a custom adapter to display the following information about a car share in a single row:

* Departure and Destination cities.
* Number of available seats.
* Date of departure.
* Time of departure.
* Image icon indicating whether the currently logged in user is the driver or participant of the car share.

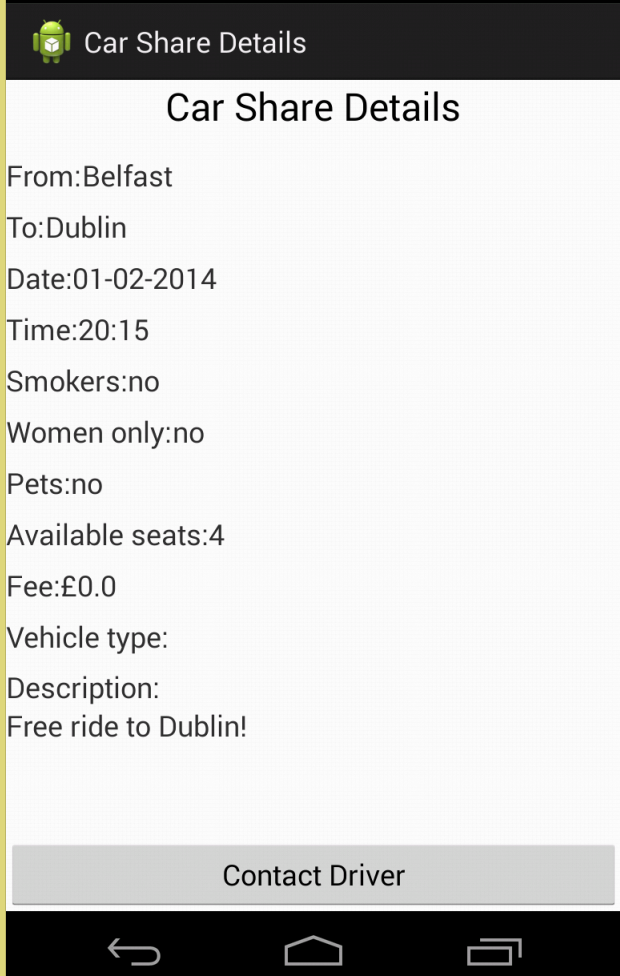
And the actual row looks like this.

The custom listview adapter which I called CarSharesListViewAdapter encapsulates another layout inside a single row of a regular list view to include all of the above information. Although time consuming, this gives me a lot of flexibility in customizing existing controls to suit my needs.

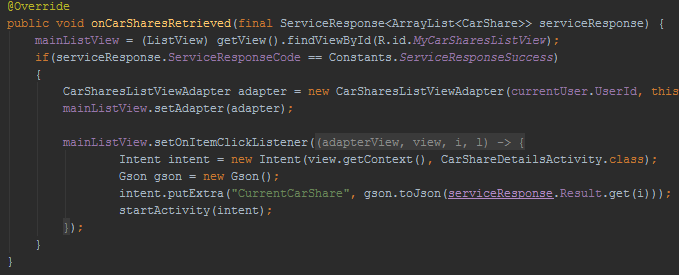


## Car share details screen.

After adding an onclick listener to the above listview, I proceeded onto creating a screen which displays all the information about selected car share.

At the time of creation, the car share details activity gets the currently selected car share object passed in. It then extracts that information and puts into the appropriate textviews to display the information to the user.

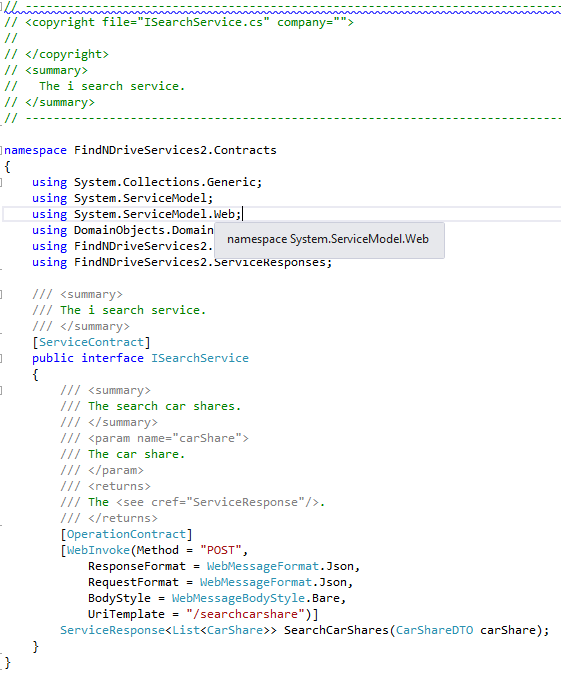
The contact driver button is not currently hooked up to an event handler because the contact driver functionality has not yet been developed.



**TODO: Revise the layout of the CarShare details screen.**

## Search functionality - basic.

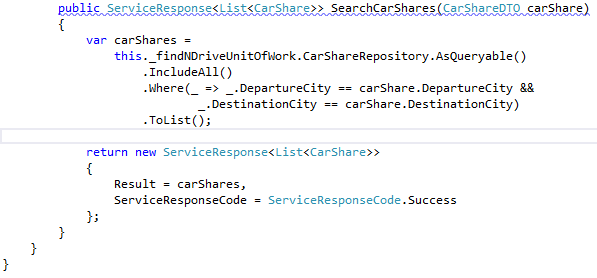
For now, I decided to implement basic search functionality allowing the user to search for car shares by departure and destination cities only. Full search functionality will be developed later.

The reason I only decided to go for the basic search functionality now is that other features which I committed to for my interim demo still require a lot of work.

For the search to work properly and to keep separation of concerns as high as possible, I decided to deploy the search functionality in its own WFC service. For this reason, SearchService was created.

The service contract on the diagram illustrates the operation contracts for this service developed to date.

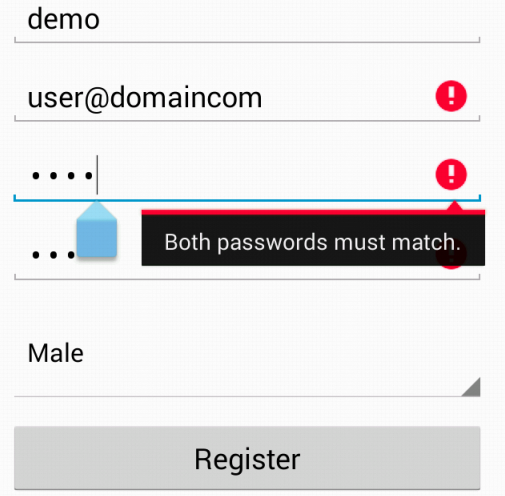
Also, as you can see on the screenshot below, the SearchCarShares method is very basic at this stage. It simply checks the CarChare repository for entries matching the departure and destination cities passed in the CarShareDTO object from the Android client application.



## Client Side Data Validation

Although data validation is already present on the WCF side to prevent erroneous information from being entered into the database, it does not provide user with immediate feedback about data they have just entered into for example a textview. The WCF validation only kicks in at the time of making a request to the WCF service.

For this reason, I developed additional layer of input validation which works on the client side and providers the user with immediate feedback on the values they have just entered.

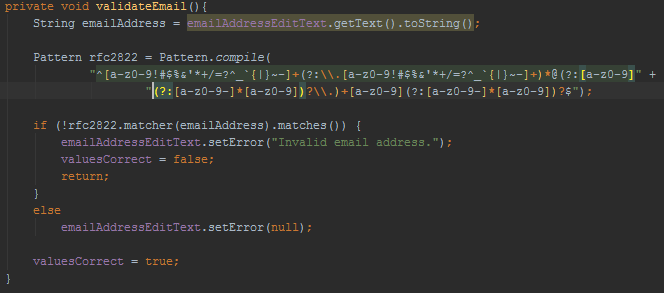
After some research, I decided that the best way to visually show input errors is to use the Android’s built in functionality.

It uses visual warning to warn the user of any input errors immediately after the focus is set on another text view. It shows a red error sign with a warning message informing the user of what the error is.

The user then knows what caused the error and how to correct it.

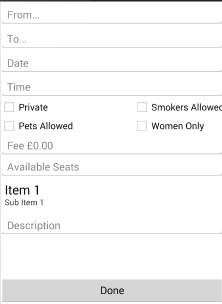
This particular validation method uses the TextView’s setError method which shows the error sign and displays a message.

The code implementing validation rules has been developed entirely by myself. Below is an example of the validateEmail() method which uses a regex expression to check the format of the email address entered by the user as soon as they focus out of the text view.



**TODO: Validation is only currently hooked up to user registration. Make sure it is connected to other parts of the application also.**

## Add new car shares functionality.

Another one of the features I have committed to for my interim demonstration was adding new car share listings to the database. Although still very basic, the current system allows the user to enter all the necessary information that a car share must have in order to be valid.

Please bear in mind that the current layout of the screen is just a placeholder for the time being and will be replaced by a much improved version after the interim demo.

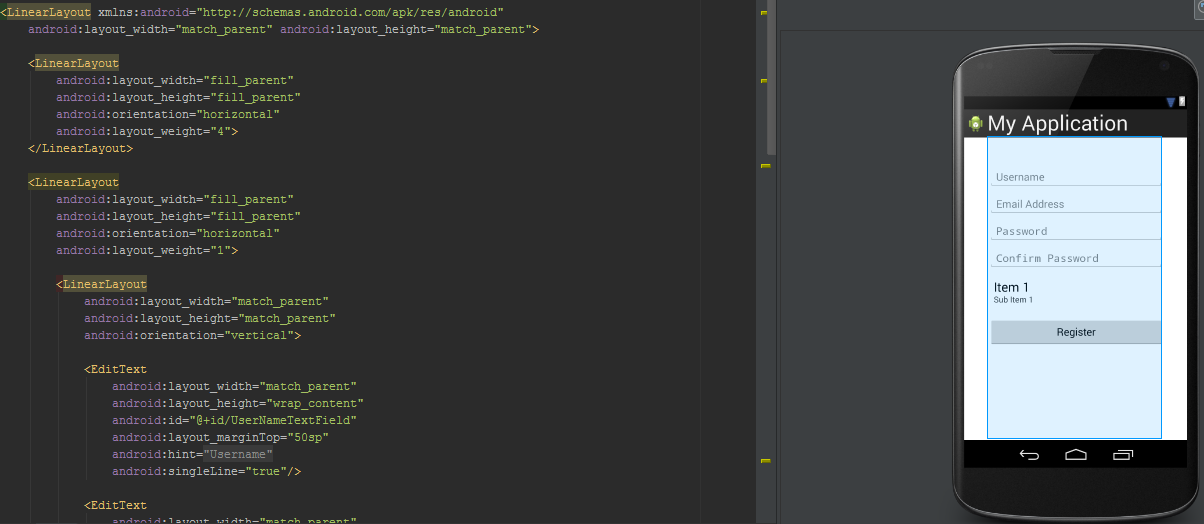
When the Done button is pressed, the client application builds a new CarShare object from the data entered by the user. It will then use the PostNewCarShare task to contact the WCF service that will add the new car share to the database.

**TODO:** **Revise and improve the layout of the ‘post new CarShare’ activity.**

## Refactored UI, scalable user controls, cleaned a lot of XML code.

I finally found some time to refactor some of the layout XML files and make sure they scale properly on screens of various sizes. I have achieved this by removing the width & height attributes from each of the TextViews, Button controls etc. and have instead used LinearLayouts together with their weight properties to ensure the controls rescale properly.

Below is an screenshot of code of the much improved registration activity which now scales properly on all screen sizes.



As you can see, there are multiple nested LinearLayouts. This is OK since LinearLayouts are very easy on the memory and do not require a lot of processing power to generate like the RelativeLayouts do.

**TODO:** **Revise the layout of all other activities to make sure they scale properly.**

## Refactored a lot of web services code.

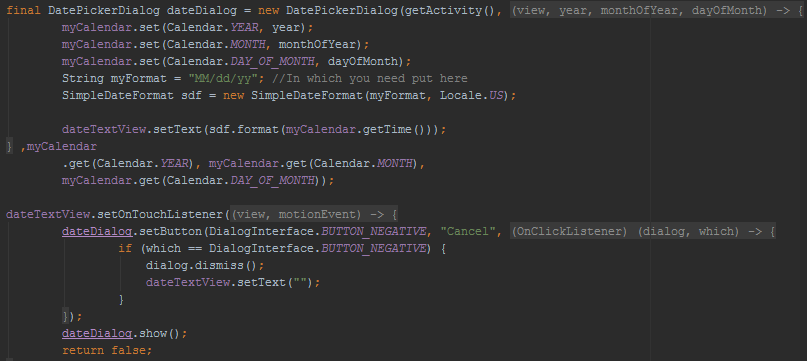
In this exercise which I just finished, I spent about two hours on refactoring and rewriting a lot of the code in my WCF services that I was not 100% happy with. This included removing any code smells, temporary workarounds and writing comments.

# 09th December – 23th December

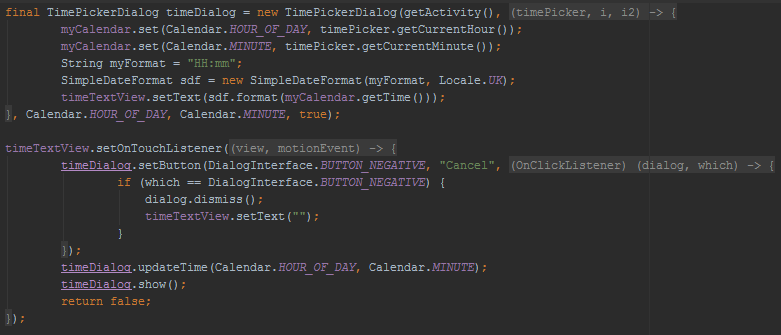
## Full search functionality.

Having completed all the features for my interim demonstration, I decided to fully implement the search facility in the name of the famous saying “Under commit and over deliver”

I started by connecting the date & time fields in the search pane to appropriate event handlers. Surprisingly, this took a lot more work than I originally anticipated. Below is a block of code required to properly initialise and display a date picker on the screen.



And the Time picker event handler:



After wiring up the event handlers to the rest of the search fields, I proceeded onto writing additional code on the WCF side to make sure the CarShare repository filters the list of CarShare objects properly according to the search criteria provided. Below is the code of the entire search method responsible for providing the android client application with correct search results.

public ServiceResponse<List<CarShare>> SearchCarShares(CarShareDTO carShare)

{

var carShares =

this.\_findNDriveUnitOfWork.CarShareRepository.AsQueryable()

.IncludeAll()

.Where(\_ => \_.DepartureCity == carShare.DepartureCity &&

\_.DestinationCity == carShare.DestinationCity)

.ToList();

if (carShare.SmokersAllowed)

{

carShares = carShares.Where(\_ => \_.SmokersAllowed).ToList();

}

if (carShare.WomenOnly)

{

carShares = carShares.Where(\_ => \_.WomenOnly).ToList();

}

if (carShare.PetsAllowed)

{

carShares = carShares.Where(\_ => \_.PetsAllowed).ToList();

}

if (carShare.Free)

{

carShares = carShares.Where(\_ => \_.Fee == 0.00).ToList();

}

return new ServiceResponse<List<CarShare>>

{

Result = carShares,

ServiceResponseCode = ServiceResponseCode.Success

};

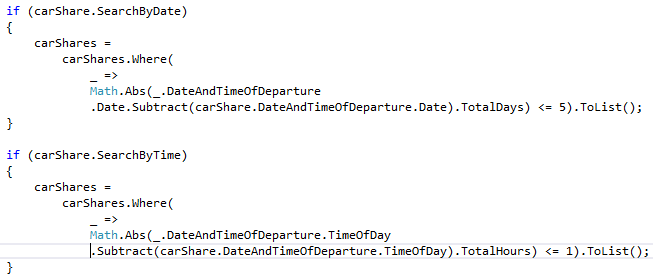
}

Once again, we can see how much Entity Framework together with the Repository and Unit of work patterns benefits the entire application. There is no need to write complex SQL queries and data is retrieved from the database using short LINQ statements which operate on collections. LINQ allows me to filter the entire list by using the .Where clause which automatically filters the list for me based on a specific criteria.

## Flexible dates/times search.

Just like an airline booking system allows to search for tickets with a flexible dates option, I though it would be a great idea to implement a similar system in my application but extend it to provide a flexible times as well.

With Entity Framework and LINQ, this was quite easy to implement. The following block of code was added to my SearchForCarShares method:



The range values for the flexible dates and times search feature have been hard coded for the time being to allow quick testing. 5 is the value of days which the query takes into consideration when filtering the list of car shares. This means that any car share with the departure date within 5 days either before or after the specified date will be included in the search results. The same rule has been applied to the time, where all car shares which are set to depart within 1 hour either before or after the specified time will be included in search results.

The actual calculation to find out the date and time difference is extremely simple. The .NET framework provides a subtract method on its Date object which then can be used to find the exact values by calling the TotalDays or TotalHours methods. In order to account for both negative and positive values (for example, if we subtract a larger date from smaller date object, this will result in a negative number of total days i.e. -3) I encapsulate the entire calculation in Math.Abs() function which will convert any negative integer into its positive counterpart. The only thing that’s left to do here is to replace the hardcoded values of 5 and 1 with parameters supplied to the WCF service from the Android client which will allow users to specify the flexible date & time range.

**TODO:** **Replace hard coded flexible search values with extra parameters being passed in from the Android client.**

## Finalised the application navigation structure.

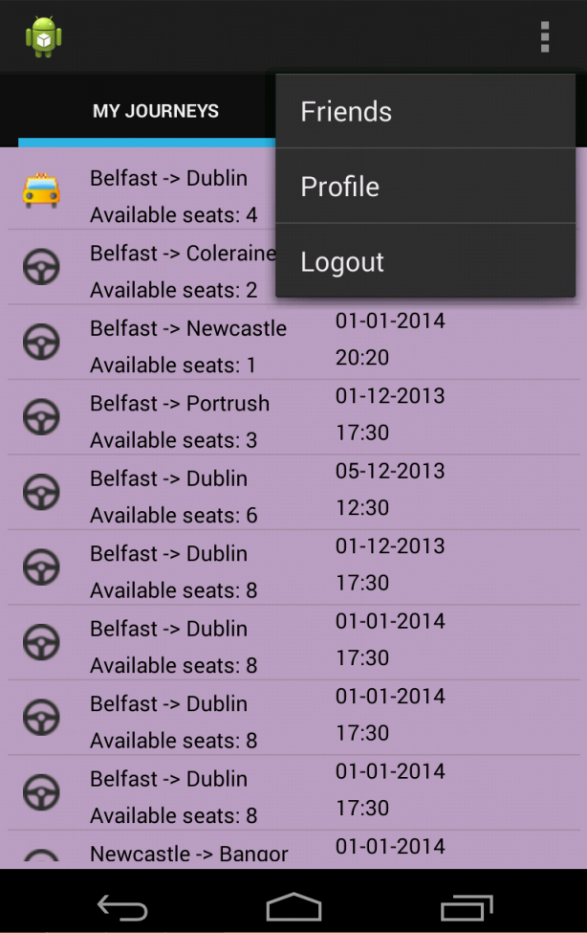
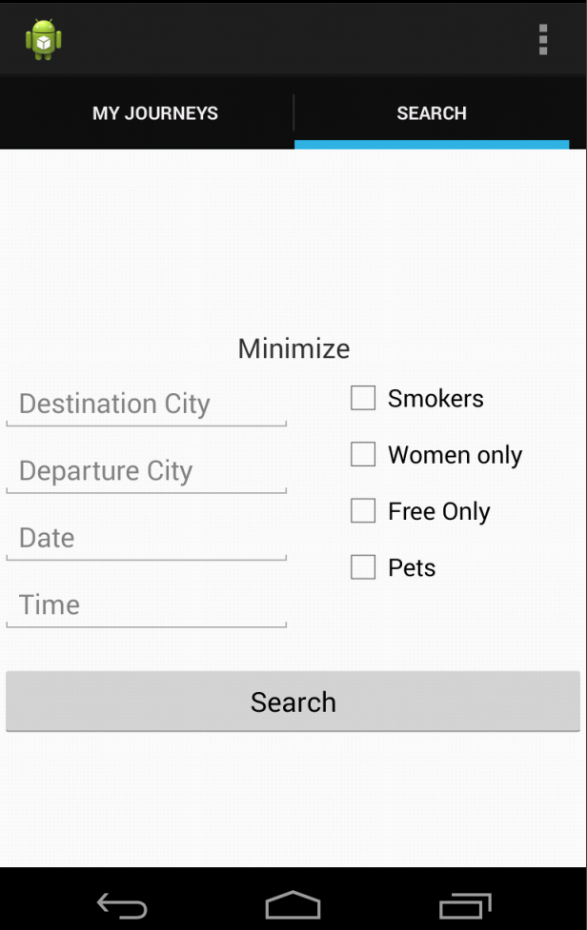
After a lot of experimenting with various navigation structures, I realized that the simplest solution is going to work best for my application. This means I am going to proceed with implementing a Spinner navigation system with a slight change in the user home screen.

The spinner navigation will provide the user with a menu button built into the action bar which will allow the user to navigate between various activities by selecting one of the menu options.

The change to the user home screen which I mentioned above involves encapsulating two separate views into a single home screen and providing tab controls with swipe-able views to enable users to quickly navigate between them.

One of the tabs will list all car shares associated with the user while the other one will implement the existing search functionality with search results being displayed inside the same activity for better user experience.

This layout allows users to quickly switch between the two views by either tapping on the tab or swiping their finger across the screen to switch to the other view.



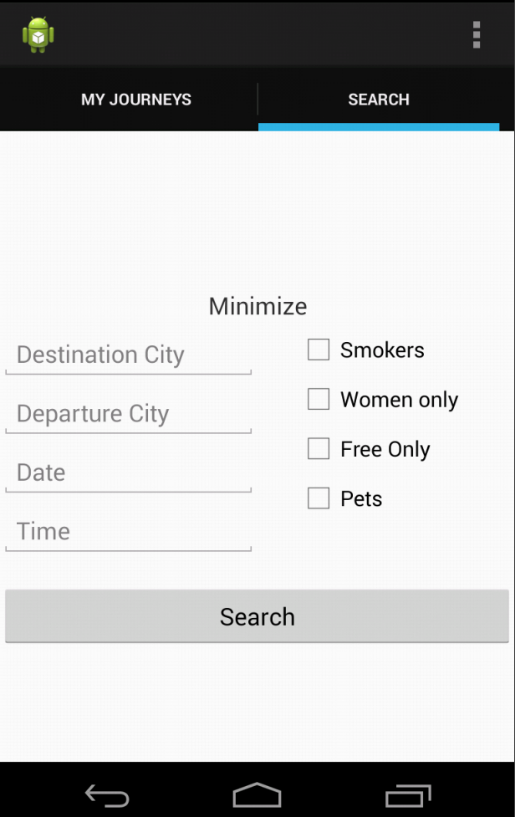
**TODO:** **Add controls to the “My Journeys” view to allow users to navigate to the activity responsible for adding new car shares.**

**TODO:** **Develop transition for the search view to allow search results to be displayed in the same tab.**

# 24th December – 6th January

## Refactored a lot of code on client side.

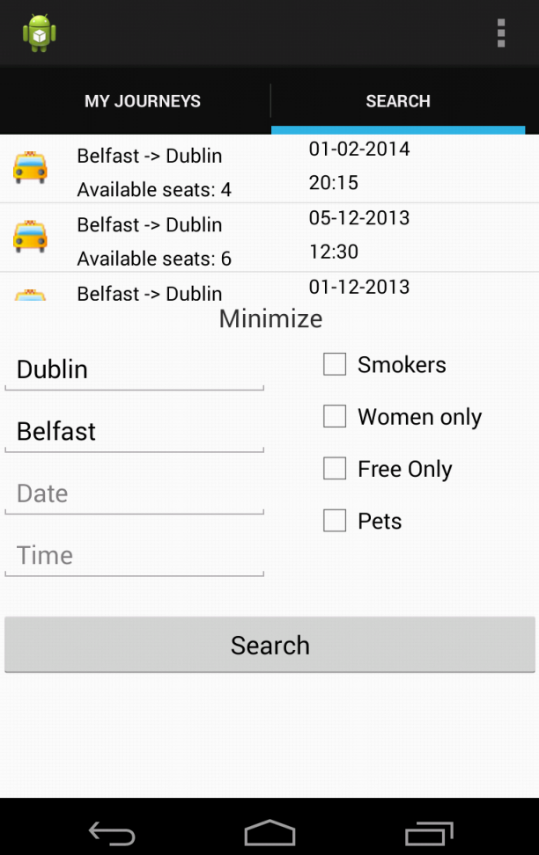
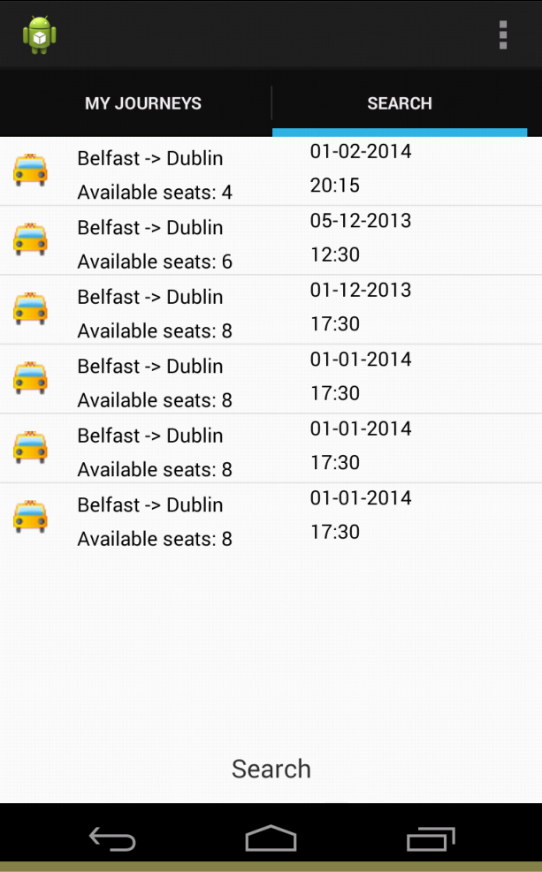
Now that the semester is over and all of my assignments have been handed in, I decided to spend some time on refactoring the code on the client side. This involve removing any code smells that and temporary workarounds that were put in place.

I have also renamed a lot of classes, variables and wrote additional comments in code that was not 100% clean.

## Search animation.

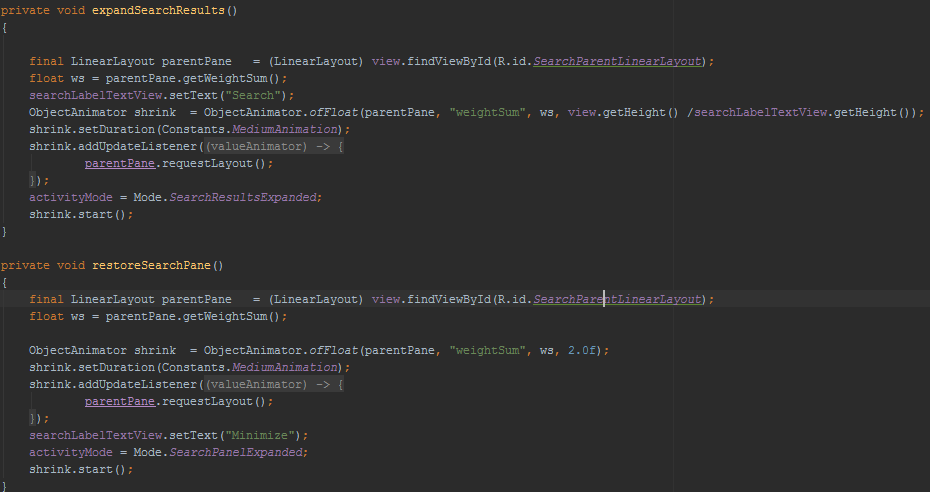
Just last night, I finished working on an animation in the search view which uses Android’s ObjectAnimator to shrink and expand the search panel as requested.

When the user first opens the search view, they will see an empty ListView and an expanded search panel with all the search options clearly visible. Once they fill in their search criteria and hit the search button, the search panel will minimize itself and the giving the ListView 90% of the available screen space to display the search results.

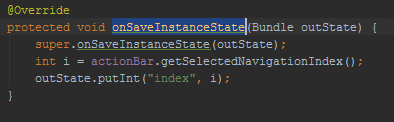
****The remaining 10% is dedicated to display the “Search” on the label on the bottom of the screen which when tapped, will minimize the search results ListView and expand the search panel. The difference this time however, is that the search results ListView although minimized, is still visible and the user can still scroll through it and select individual elements from it. As you can see, the Search label changed its text to Minimize which when tapped will minimize the search panel giving the search results ListView 90% of the available screen again.

The animation is performed using the weightSum property which means that regardless of screen size the animation will work and be displayed on all screen sizes and orientations.

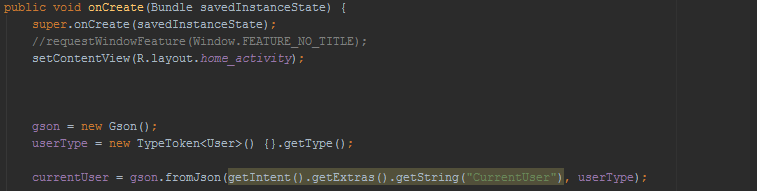
Below is the code responsible for performing the animation. The Android’s ObjectAnimator provides a great way of defining custom animations on the fly without the need for XML files.



## Handling screen orientation changes.

I have also developed some code to partially handle the issue where change of the screen orientation causes the application to crash due to nullreference exception. The current approach which I am taking is to override the onSaveInstanceState method and save all the necessary variables into the Bundle which is one of Android’s ways of passing data between activities.

After the orientation changes, the Activity goes through its creation process again and the key to reload all the necessary values back into place the code into the onCreate method as it is illustrated in the example below:



An ideal solution would be to create a base class which implements this system and make every Activity inherit from it.

**TODO: Create a base class which implements the system for saving and restoring state of an Activity and make all other activities inherit from it.**

TODO List

* **TODO: Add more models as the application grows.**
* **TODO:** **Code is still to be properly refactored and commented. UserService and CarShare service methods to be partially redesigned and possibly relocated to another service to enable further separation of concerns.**
* **TODO: Make it possible for the user to log in to the system using either their username or email address.**
* **TODO: Create a basic User Interface for the client application.**
* **TODO:** **Revise the layout to make sure it scales properly on screens of various sizes and in both orientations.**
* **TODO:** **As more domain objects and WCF services are being created, more repositories will need to be added to the unitofwork class.**
* **TODO:** **Investigate using proper cookie files instead of a custom cookie-like file.**
* **TODO: Streamline the process of adding information to the HTTP header on the client side.**
* **TODO: Validation is only currently hooked up to user registration. Make sure it is connected to other parts of the application also.**
* **TODO:** **Revise the layout of all other activities to make sure they scale properly.**
* **TODO:** **Replace hard coded flexible search values with extra parameters being passed in from the Android client.**
* **TODO:** **Add controls to the “My Journeys” view to allow users to navigate to the activity responsible for adding new car shares.**
* **TODO:** **Develop transition for the search view to allow search results to be displayed in the same tab.**
* **TODO: Create a base class which implements the system for saving and restoring state of an Activity and make all other activities inherit from it.**

Issues List

* **~~ISSUE: I’m having major issues with converting dates and times to and from the WCF service. After a lot of research, I found that when sending a serialized Date & Time object, WCF will only accept it in the following epoch format: "/Date("+epochlong+")/" which means I’m going to have to write my own WCF date/time converter. Medium priority, will investigate later.~~**
* **~~ISSUE: WebMatrix.WebSecurity throws nullreference exception while attempting to log user in. Blocker, must investigate immediately.~~**
* **ISSUE: Each time the device is rotated and its orientation changes, Android tears down the entire activity and goes through the process of creating the activity from scratch again. This creates the need to develop a system whose responsibility will be to save the state of each activity before it’s torn down and reload its state after its recreated. Without such a system in place, I am currently experiencing nullreference exceptions when the Activity attempts to reload information which has not been saved or reinitialized and is therefore null. This crashes the application but is still a rather low priority issue which will be addressed later. – Partially addressed.**