Technical Manual

“ParkLet”

Shaun Kinsella – Student ID -: 14740175
Supervisor: Dr Stephen Blott
Date Completed: 08/05/20

Table of Contents

Abstract

ParkLet is an Android app aimed at helping homeowners easily rent their driveways and commuters find competitive rates for parking. The app is written in Java using the MVVM architectural pattern and utilises Firebase Realtime Database and Firebase Cloud functions. Its principal objective is to address the lack of affordable parking for daily commuting and large events.

"ParkLet: Airbnb for parking"

Motivation

In cities such as Dublin with high levels of commuters, there is a substantial lack of parking facilities available and those that are available are quite often charged at a premium rate. A census taken in 2016 [1] shows that 130,477 people commute to Dublin on a daily basis, the lowest percentage of commuters travelling by private car in a given area was 60%. There are circa 20,000 private ticket or pay and display parking spots available and as mentioned are charged at exuberant hourly rates.

Even on campus in DCU where pricing is competitive, the demand outweighs the availability and quite often students/staff have to often resort to parking in nearby estates. Some local residents rent out their driveways to students on a cash on hand basis. ParkLet seeks to make this process easier by offering commuters a platform that allows them to easily identify the availability of the cheapest nearby spots to park. And in the case of the homeowner, an easy way to track and manage their bookings.

There have been previous attempts at apps that serve this purpose but they are often aimed at serving commercial car parks. ParkLet offers an alternative in an Airbnb approach and utilises NFC to check-in and out of the rented driveway, allowing contactless management of booking slots. Emphasis is placed on making the process streamlined to allow people to generate supplemental income and ease the stress of commuting long distances.

Glossary

- **MVVM**: Model, View, ViewModel. Architectural pattern which separates applications into 3 distinct layers for separation of concerns and highly decoupled layers.
- **View**: The UI aspect of the application. Responsible for the appearance of the app and relaying user inputs to the ViewModel. In this context, fragments and activities mainly.
- **ViewModel**: A Mediator between the Model and the View. Relays state, transforms Model data into a format consumable by the View.
- **Model**: Represents the domain model of the application. Simple entity classes.
- **Repository**: Class that encapsulates the logic required to fetch data from respective data sources. Often used with the model to map the fetched data to Entity classes.
- **LiveData**: Lifecycle aware data holder class. Enables use of the observer pattern. Only emits changes when it is observed by a “live” component.
- **NFC**: Near Field Communication. Allows data exchange between devices and or an NFC chip at extremely short distances.
- **Geohash**: A means of expressing a geographic location using alphanumeric strings. Utilises Z order curves to break geographic locations up into a grid. The longer the string the more arbitrary precision of the location.
- **Geocoding**: The process of turning a text-based address into a latitude, longitude coordinate.
- **Activity**: Class which acts as a potential entry point in the Android app, provides the window in which the UI is drawn.
- **Fragment**: Class representing a proportion of the UI, hosted in an activity. Fragments mirror the hosting Activity’s lifecycle but may be independently paused or resumed.
- **Databinding**: Allows the binding of UI components to the app’s data sources. Simplifies updating the UI and reduces calls to frameworks.
- **RecyclerView**: Android framework class allowing the efficient display of a collection of data in a grid or list format.

Research

The below summarises a non-exhaustive list of some of the research undertaken in the design and implementation of ParkLet.

Android framework

This was my first-time coding/designing an Android app so at the beginning a significant amount of time was invested in researching the Android Framework components, the life cycles of each, and suitable architectures for the app's requirements. This resulted in taking the MVVM architecture approach.

MVVM

MVVM (Model, View, ViewModel) is now considered the recommended architecture for Android apps [2]. It helps ensure proper separation of concerns, a high level of decoupling, and encourages the idea of a passive view that reacts to the streams of data that are prepared and exposed by the ViewModel. Various guides warn that a familiarity with the Android framework and experience making Android apps is required to implement this effectively.

However, this architectural pattern seemed to lend itself to the project, particularly in combination with Firebase Realtime database.

Firebase Realtime Database

Adjusting to using the Realtime database in the beginning was difficult. It required ensuring that the app's data was structured in a shallow and sometimes de-normalised way in order to fit its limited query functionality. However, the real time updates ensure that the users were always dealing with the most up to date snapshot for each property or booking. The horizontal scaling it would afford meant that ParkLet could grow or downscale relative to its usage demands.

Google Maps API

The Map View forms an integral part of ParkLet's main objective. Presenting users with a real time map that is both easy to understand and navigate through the properties on offer. I investigated exactly what level of customisation was available and how responsive it would be in ParkLet's context in comparison to its competitors such as OpenMaps.

LiveData

The LiveData class forms the backbone of communication between the various layers of the Android MVVM architecture. As a wrapper class, I could expose whatever object which was required by the View or ViewModel as a stream. Learning about how transformations could be triggered "mid-stream" in order to change the objects to a format that was ready to present on the UI, and how to combine data streams coming from more than one repository was crucial in the development of the project.

Geohashes
As Firebase does not allow querying by more than one term, filtering properties based on their proximity to a Latitude and Longitude pair was not possible. Instead, leveraging the GeoFire library [3], I was able to store a query-able geohash that could be checked for a property's proximity to the user's area of interest. This geohash would be stored under the same key as a property but under a different node. While the usage of the library is simple, I required more of an understanding of how geohashes correspond to real world coordinates in order to implement the "average pricing in an area" functionality. This led to the implementation of the "GeoPriceBuckets" explained later in the manual.

Firebase Cloud functions
In order to implement the notification functionality for new bookings, renter check-ins and cancellations, Firebase Cloud functions were used. Learning how to use watchers for specific database updates for triggering these notifications was crucial. I had only no real prior experience with either JavaScript or TypeScript in which these functions were written, as well as setting up the Firebase project to use them and deployment of the above functions.

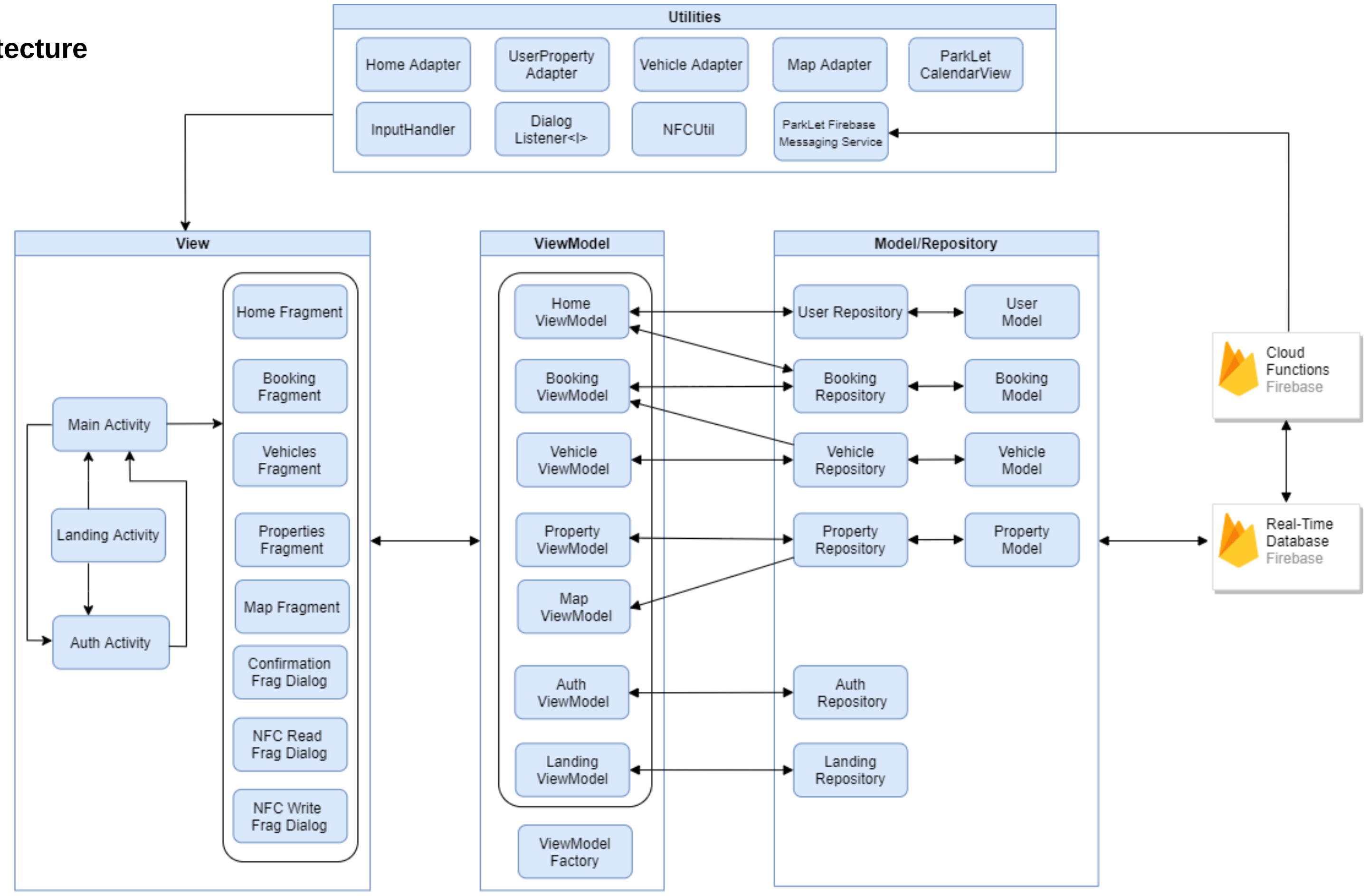
Researching asynchronous methods and promises was also required to make sure that the fetching of the required information was complete before sending notification payloads.

Dagger2
Dagger2 [4] is a dependency injection framework that provides compile time dependency injection. This differs from the Spring method of runtime injection that tends to be more verbose and is not validated until runtime. However, dagger is more difficult to implement. The mechanism by which ViewModels are provided to the View does not allow for non-zero argument ViewModel constructors. So, in order to make sure that a ViewModel was not tightly coupled with its required Repository classes, I had to research Dagger to learn how to inject the ViewModels dependencies so they could later be mocked for unit testing.

NFC
One of the main promised features of ParkLet was the contactless check-in functionality, this required research into how NFC worked with Android, associating ParkLet with its own NFC tags, writing the property UID to tags and working with intents to intercept NFC event.

System Design

System Architecture



The MVVM architecture as mentioned is broken up into three main layers to provide a high level of decoupling and abstraction from each other.

View

Activities

The View is broken up into Activities and Fragments. ParkLet utilises 3 Activities in total. The Landing Activity acts as an entry point to the App and checks if a user is Authenticated and signed in. If not, they are redirected to the Auth Activity which allows them to register. If they are registered and signed-in they are redirected to the Main Activity. While familiarising myself with the architecture a tutorial provided by Alex Mamo was used, as is accredited in the Java docs of the Landing and Auth Activity. [5]

The Main Activity sets up the Nav Controller which decides which Fragment is to be inflated and attached to the Main Activity. By default, it is set to the Home fragment and then listens for user interactions with the navigation drawer. The Main Activity is also responsible for hosting the NFC adapter and the pending intent filter which catches and processes what to do with NFC tags being detected. Having a singular Activity that acts as a main and hosts many Fragments is preferred due to the how lightweight Fragments are in comparison to Activities.

Fragments

Each Fragment represents a portion of ParkLet's UI and the functionality required by each. The Home Fragment displays a user's bookings and allows them to navigate to the Map Fragment to create new vehicle bookings for example. Each fragment is subscribed to one and only one ViewModel. In the case of the Home Fragment, it observes the HomeViewModel which requests information on the user and their bookings from the respective User and Booking Repositories on behalf of the Home Fragment.

Fragments and Activities are allowed to share a ViewModel for the passing of information between each but when the ViewModel is declared consideration must be given to who owns it. In this case, the Main Activity is given ownership of the ViewModel as it is aware when the fragments have been destroyed and therefore the ViewModel can safely be destroyed as well. This was commonly used when a Fragment and a RecyclerView belonging to that Fragment required the ViewModel.

ViewModel

The ViewModel layer is responsible for requesting data on behalf of the View and transformation of that data to be consumed by that View. It also relays user interactions with the UI to update data to the Repository layer. A View subscribes to a ViewModel by observing the result of the ViewModels public methods. It is of the utmost importance that a ViewModel holds no reference to a View as this may result in life cycle leaks where the ViewModel out-lives the View it was relaying information to.

Each ViewModel holds a reference to the Repository(s) it is interested in. Some ViewModels share Repositories with other ViewModels but for different use cases. For example, both the Booking ViewModel and Home ViewModel use the Booking Repository. The Booking

ViewModel uses it to expose only the dates of bookings relevant to a particular property, whilst the Home ViewModel uses it to expose the full booking info of a user.

The ViewModelFactory is a common pattern used to provide ViewModels with non-zero argument constructors, in this case the Repositories which it requires. These Repositories are injected using Dagger2 and are provided as singletons that can be shared. The ViewModelFactory itself is injected into the necessary Fragment or Activity.

Model/Repository

The Repository layer acts as a data access level abstraction, in the case of ParkLet, interactions with Firebase Realtime Database. Due to this implementation, the rest of the layers are unaware of how data is updated or fetched. This means that at a later time the Repository layer can be refactored to use a different database, or implement a rest style service to a backend server without affecting any other functionality of ParkLet.

The entity models form the basis of a contract, a representation of how each individual object should be mapped to on the database and what they should be transformed back to when fetched from the database. Each entity model contains an empty constructor which is used by firebase to map the results of a query back to the object. It then fills the objects fields using the getters and setters provided. They also provide methods for any business logic associated with that class, for example in the case of the Booking object, methods to return an appropriate string for whether a user has just checked-in or out.

Utilities

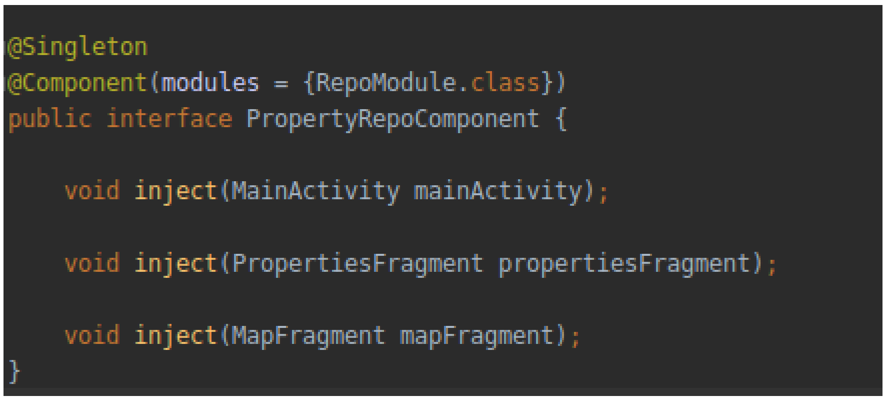
The Utilities module contains classes that are either commonly used by Fragments or classes when it was appropriate to abstract out of the parent Fragment or Activity. Doing so would allow refactoring or replacement of a particular component such as the DatePicker for the Booking Fragment.

- DialogListener Interface: Used to notify the Main Activity when a Dialog Fragment was attached or detached from it.
- InputHandler: Utility class used to hide the keyboard when a user completed or cancelled an input form.
- NfcUtil: responsible for the reading and writing of ParkLet NFC tags.
- ParkLetCalendarView: extends from DateRangePicker [6]. Used to set the appearance and behaviour of the DatePicker used for placing Bookings.
- ParkLetFirebaseMessagingService: utility class responsible for handling downstream messages from Firebase Cloud Messaging, creates the ParkLet channels on which messages are processed, and notifies the app when a user’s device token has been updated.
- Adapters: These are various adapters used by Fragments to specify how a RecyclerView should bind objects to its respective ViewHolder. The Adapter is also responsible for attaching onClickListeners to any button contained within each item it displays. In the case of the PropertyAdapter, the deletion of a property or writing an NFC tag for that property.

Dagger

The DI (Dependency injection) module contains Dagger resources for providing DI to Parklet. It consists of a RepoModule which specifies how each of the repository dependencies are to be provided to the ViewModels requiring it. The “@singleton” annotation ensures that only one instance of the Repository is alive at a time, and is shared amongst the ViewModels that require it. The “@provides” annotation is necessary for informing Dagger that it is to be injected.

This RepoModule is then used by the various RepoComponents, which are interfaces that provide the methods used to inject the dependencies required. Each of these methods are given a context by stating exactly which Activities or Fragments will be using them. For example, the PropertyRepoComponent provides the dependencies needed by the ViewModels in MainActivity, PropertyFragment and MapFragment.

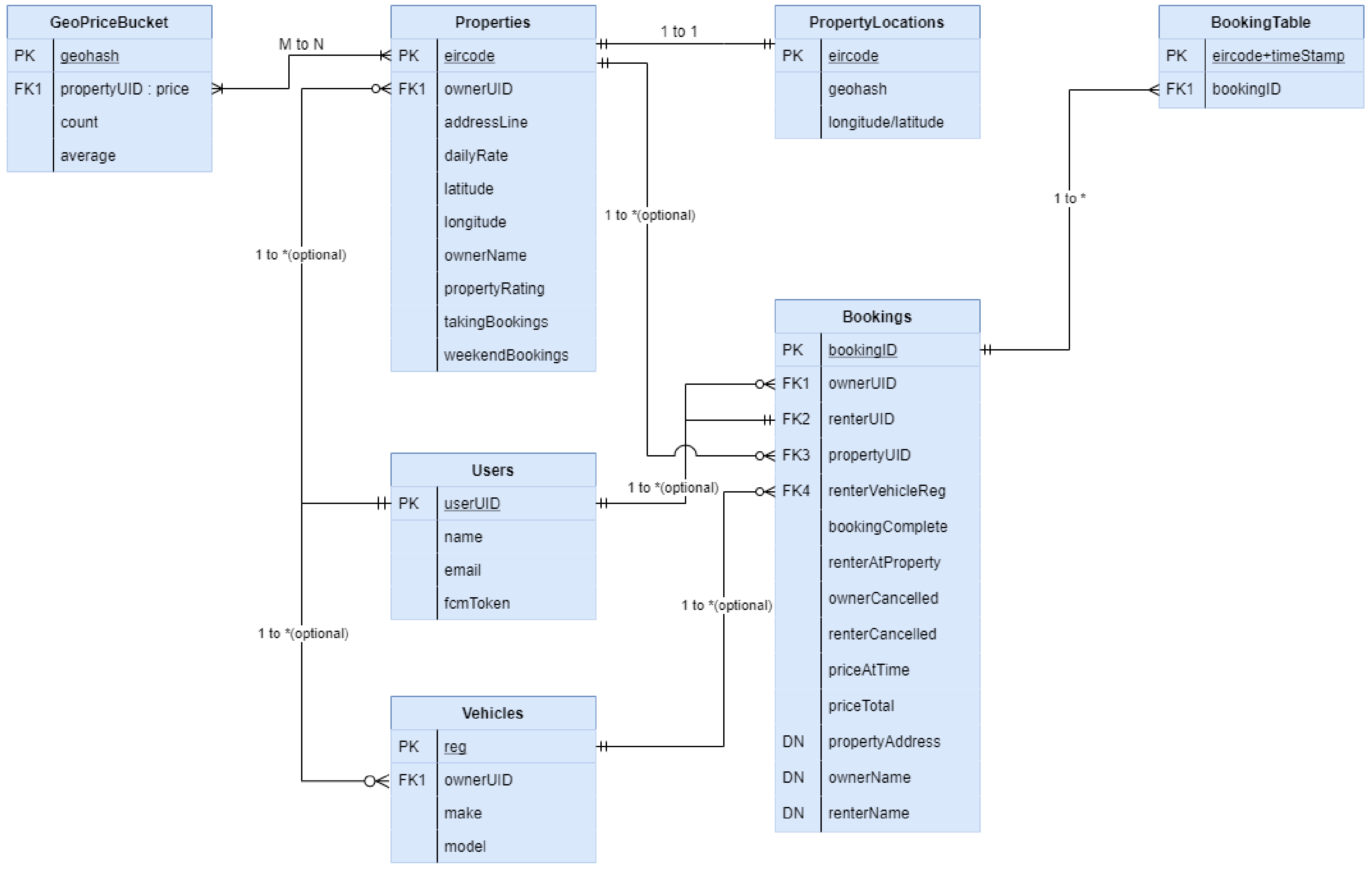


```
@Singleton
@Component(modules = {RepoModule.class})
public interface PropertyRepoComponent {

    void inject(MainActivity mainActivity);

    void inject(PropertiesFragment propertiesFragment);

    void inject(MapFragment mapFragment);
}
```

Firebase Cloud Functions

The Firebase module contains the Index.js file which is a collection of JavaScript functions that are deployed to the ParkLet Cloud instance. These functions provide the notification functionality used in conjunction with the ParkLetFirebaseMessagingService class to notify users about events concerning their bookings. Each function watches a specific node and triggers on certain actions performed such as “onCreate” in the case of a new booking being made. The price comparison functionality is also provided here through the “aggregatePropertyPrices” function.

Entity Relationship Model



ParkLet uses Firebase Realtime Database, a NoSQL json key value pair cloud database. While the app’s data has relational properties, these many to many relationships can easily be modelled in firebase with careful consideration as to how the data is structured. Preference is placed on flattened data structures that ensure that read operations are as efficient as possible, heavily nested data structures are wasteful and will result in degraded performance for fetching data.

Another strategy implemented in designing the “schema” was denormalization of data. For example, if a user’s name associated with a booking was required often, store it in the booking as well. Denormalized fields are signified in the above schema diagram by “DN”.

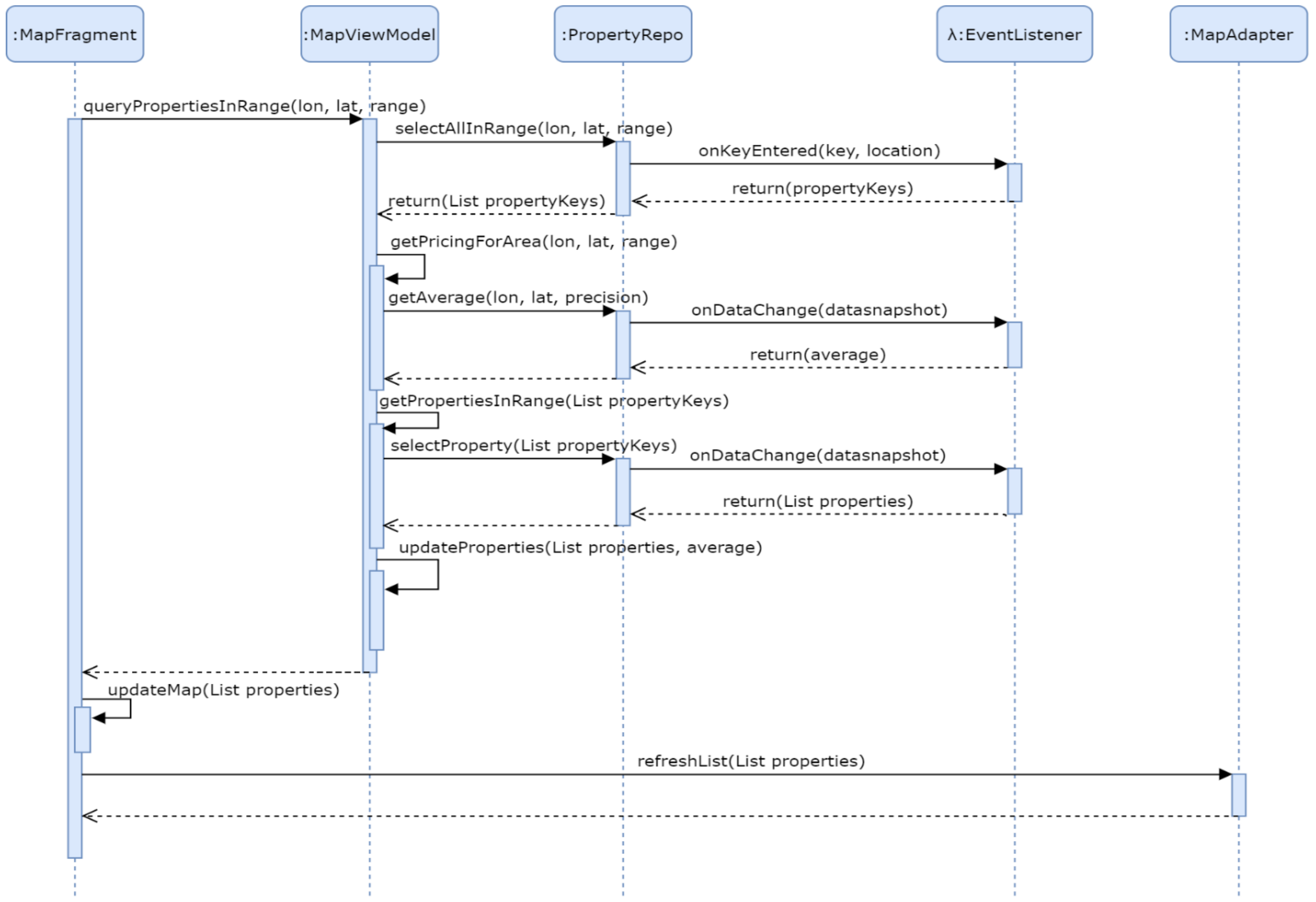
The BookingTable was designed as a lookup table for the individual days belonging to each booking entity. These individual entries point to parent booking for each individual day. The unique keys are generated by combining the ID of the property and the epoch timestamp of the day the property is booked for.

As each property may only have one booking per day these are unique. When a renter attempts to check-in, the property id is read from the tag and combined with the timestamp for that day. This concatenated value is then used as the key to fetch from the BookingTable, which then in turn contains the key pointing directly to the booking. If the user turns up at the wrong property or on the wrong day, the BookingTable entry will not exist and the user will be informed.

When a new booking is created, both the entry under the Booking node and BookingTable node are atomically written using a request map. Both of these must succeed or both will fail. This data fan out strategy allows many to many relations to be implemented.

Implementation

SD: MapFragment searchPropertiesInRange()

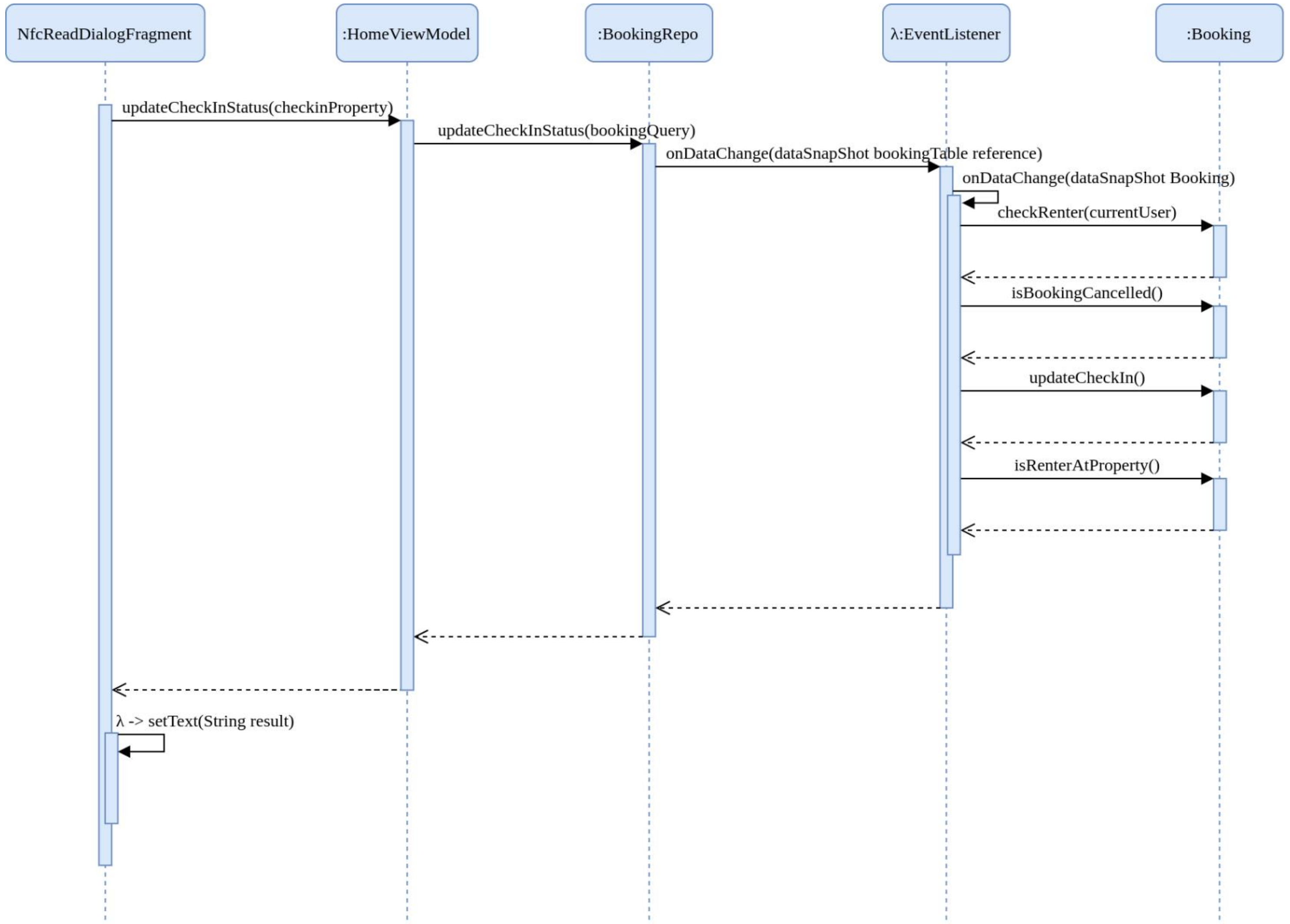

:MapFragment
:MapViewModel
:PropertyRepo
λ:EventListener
:MapAdapter
queryPropertiesInRange(lon, lat, range)
selectAllInRange(lon, lat, range)
onKeyEntered(key, location)
return(propertyKeys)
return(List propertyKeys)
getPricingForArea(lon, lat, range)
getAverage(lon, lat, precision)
onDataChange(datasnapshot)
return(average)
getPropertiesInRange(List propertyKeys)
selectProperty(List propertyKeys)
onDataChange(datasnapshot)
return(List properties)
updateProperties(List properties, average)
updateMap(List properties)
refreshList(List properties)

The above sequence diagram details the chain of events from the user submitting a search query on the Map Fragment, the MapViewModel requesting the necessary details from the Property Repo and how the resulting data returned is transformed for consumption by the Map Fragment, updating both the MapView and the RecyclerView.

The Map ViewModel first passes a query to the Property Repo for the keys of any property that falls in range of the geohash. The list of keys falling within this range is returned asynchronously. Meanwhile the average price of a property in that search area is returned. Using the list of keys fetched by “selectAllInRange”, the Property Repo is queried for all property nodes stored under those keys.

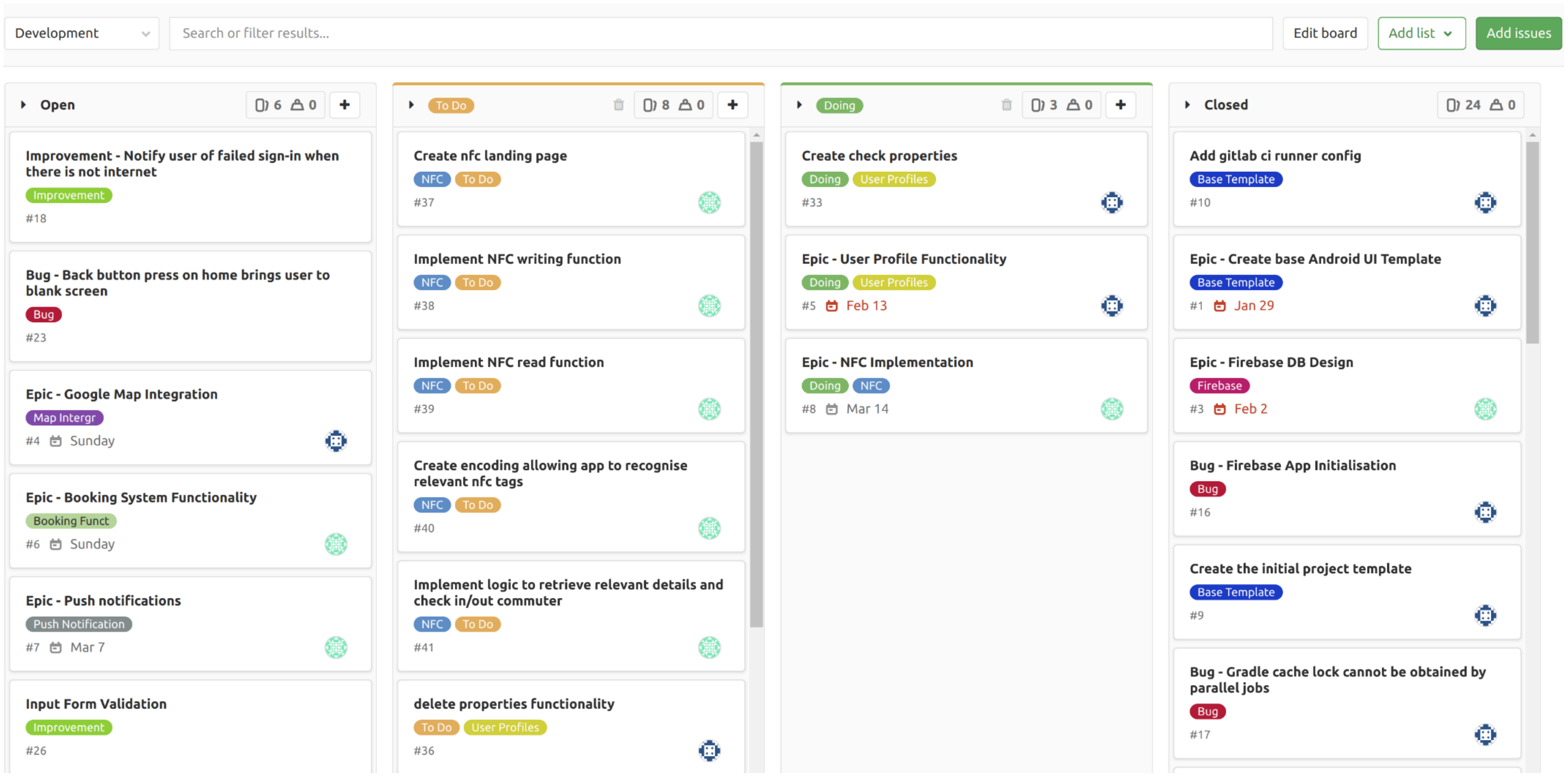
When both the list of properties and area average is returned, each individual property’s daily rate is compared to the current average and has their priceComparison field set. Once this transformation is complete the Map Fragment is notified of the new update and calls the “updateMap” and “refreshList” methods to update the map and RecyclerView respectively.

SD: NfcReadDialogFragment updateCheckInStatus()


NfcReadDialogFragment
:HomeViewModel
:BookingRepo
λ:EventListener
:Booking
updateCheckInStatus(checkinProperty)
updateCheckInStatus(bookingQuery)
onDataChange(dataSnapShot bookingTable reference)
onDataChange(dataSnapShot Booking)
checkRenter(currentUser)
isBookingCancelled()
updateCheckIn()
isRenterAtProperty()
λ -> setText(String result)

The flow that follows a renter successfully reading a property NFC tag is shown above. The property’s id is passed to the Home ViewModel and then concatenated with the day’s current date. This is used as a unique key to query the booking lookup table, which is passed to the Booking Repo. If the user is at the right property on the right day, the booking table will contain the key pointing to a booking reference. This booking reference is then checked to see if the booking is still valid, i.e. not cancelled, belonging to that user. And then returns a String response that informs the user whether the check-in was successful or not.

Development Workflow



Open 6 0

Improvement - Notify user of failed sign-in when there is not internet
Improvement
#18

Bug - Back button press on home brings user to blank screen
Bug
#23

Epic - Google Map Integration
Map Intergr
#4 Sunday

Epic - Booking System Functionality
Booking Funct
#6 Sunday

Epic - Push notifications
Push Notification
#7 Mar 7

Input Form Validation
Improvement
#26

To Do 8 0

Create nfc landing page
NFC To Do
#37

Implement NFC writing function
NFC To Do
#38

Implement NFC read function
NFC To Do
#39

Create encoding allowing app to recognise relevant nfc tags
NFC To Do
#40

Implement logic to retrieve relevant details and check in/out commuter
NFC To Do
#41

delete properties functionality
To Do User Profiles
#36

Doing 3 0

Create check properties
Doing User Profiles
#33

Epic - User Profile Functionality
Doing User Profiles
#5 Feb 13

Epic - NFC Implementation
Doing NFC
#8 Mar 14

Closed 24 0

Add gitlab ci runner config
Base Template
#10

Epic - Create base Android UI Template
Base Template
#1 Jan 29

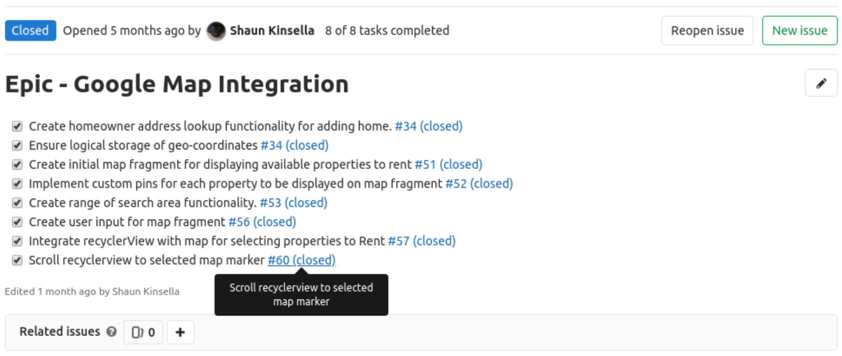
Epic - Firebase DB Design
Firebase
#3 Feb 2

Bug - Firebase App Initialisation
Bug
#16

Create the initial project template
Base Template
#9

Bug - Gradle cache lock cannot be obtained by parallel jobs
Bug
#17

In order to gauge progress and prioritize core functionality, a GitLab issues board was created as seen above. The functional requirements of the project were examined and created as Epics on the issues board. Each Epic was then further examined to break down into smaller tasks/issues. These smaller tasks would be tied to the Epic by creating a checklist in the Epic and then creating labels so each task could be related back to the parent Epic.



Closed Opened 5 months ago by Shaun Kinsella 8 of 8 tasks completed

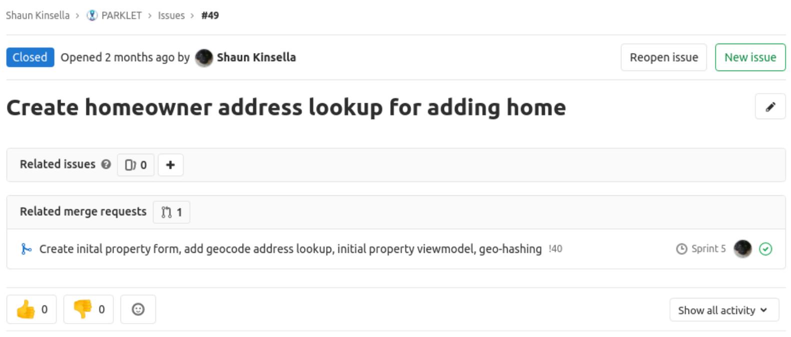
Epic - Google Map Integration

- [x] Create homeowner address lookup functionality for adding home. #34 (closed)
- [x] Ensure logical storage of geo-coordinates #34 (closed)
- [x] Create initial map fragment for displaying available properties to rent #51 (closed)
- [x] Implement custom pins for each property to be displayed on map fragment #52 (closed)
- [x] Create range of search area functionality. #53 (closed)
- [x] Create user input for map fragment #56 (closed)
- [x] Integrate recyclerView with map for selecting properties to Rent #57 (closed)
- [x] Scroll recyclerview to selected map marker #60 (closed)

Edited 1 month ago by Shaun Kinsella

This gave me a product backlog to work with in prioritising features based on their importance and ensure that the promised functionality was being delivered. The initial prioritization was based on the Gantt chart created for the Functional specification document but would be evaluated each week based on the overall progress of the project.

When working on a feature of the project, the latest copy of the master branch would be pulled down and a new feature branch would be created from this with a name matching the issue number. Only work relating to the issue would be committed on this branch. When a task is completed an appropriate commit message would be attached, describing the work done and then pushed to the project's repository.



Shaun Kinsella › PARKLET › Issues › #49

Closed Opened 2 months ago by Shaun Kinsella

Create homeowner address lookup for adding home

Related issues 0

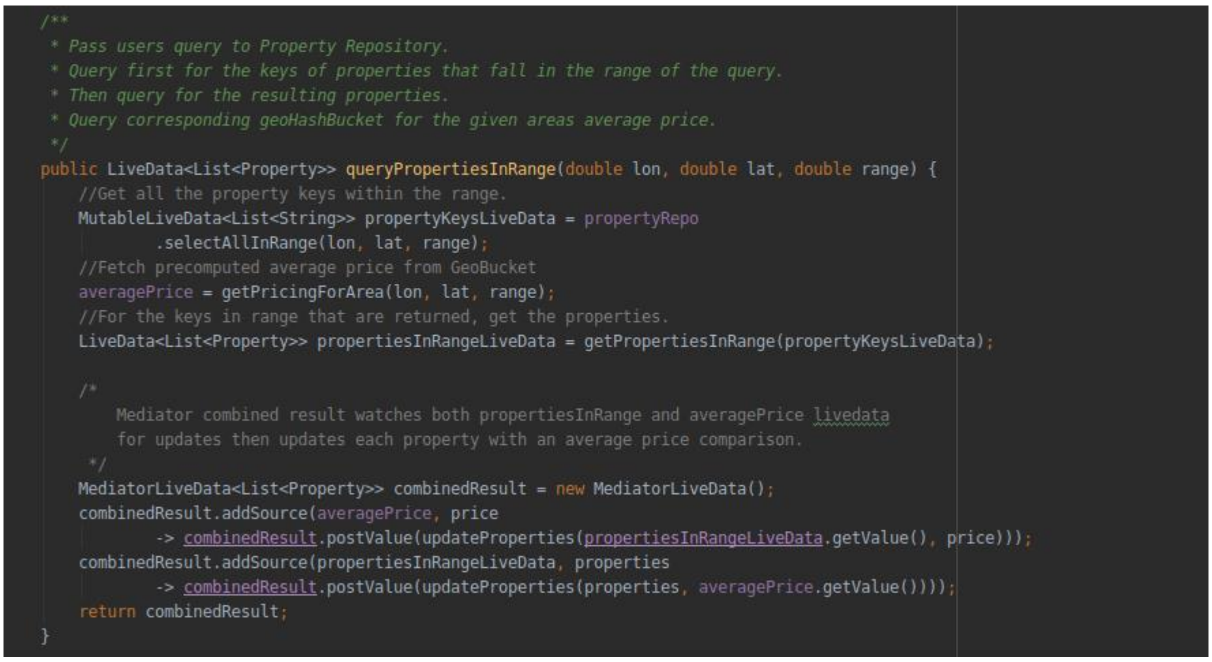
Related merge requests 1

Create inital property form, add geocode address lookup, initial property viewmodel, geo-hashing !40 Sprint 5

A merge request would then be created that will only merge if the branches tests passed, as further detailed in the testing documentation. Due to the branch naming convention, when a merge request is successfully accepted, the related issue would be automatically tied to the commit and merge, and then closed.

Sample Code

Map ViewModel: queryPropertiesInRange()



```java
/**
 * Pass users query to Property Repository.
 * Query first for the keys of properties that fall in the range of the query.
 * Then query for the resulting properties.
 * Query corresponding geoHashBucket for the given areas average price.
 */
public LiveData<List<Property>> queryPropertiesInRange(double lon, double lat, double range) {
    //Get all the property keys within the range.
    MutableLiveData<List<String>> propertyKeysLiveData = propertyRepo
            .selectAllInRange(lon, lat, range);
    //Fetch precomputed average price from GeoBucket
    averagePrice = getPricingForArea(lon, lat, range);
    //For the keys in range that are returned, get the properties.
    LiveData<List<Property>> propertiesInRangeLiveData = getPropertiesInRange(propertyKeysLiveData);

    /*
        Mediator combined result watches both propertiesInRange and averagePrice livedata
        for updates then updates each property with an average price comparison.
     */
    MediatorLiveData<List<Property>> combinedResult = new MediatorLiveData();
    combinedResult.addSource(averagePrice, price
            -> combinedResult.postValue(updateProperties(propertiesInRangeLiveData.getValue(), price)));
    combinedResult.addSource(propertiesInRangeLiveData, properties
            -> combinedResult.postValue(updateProperties(properties, averagePrice.getValue())));
    return combinedResult;
}
```

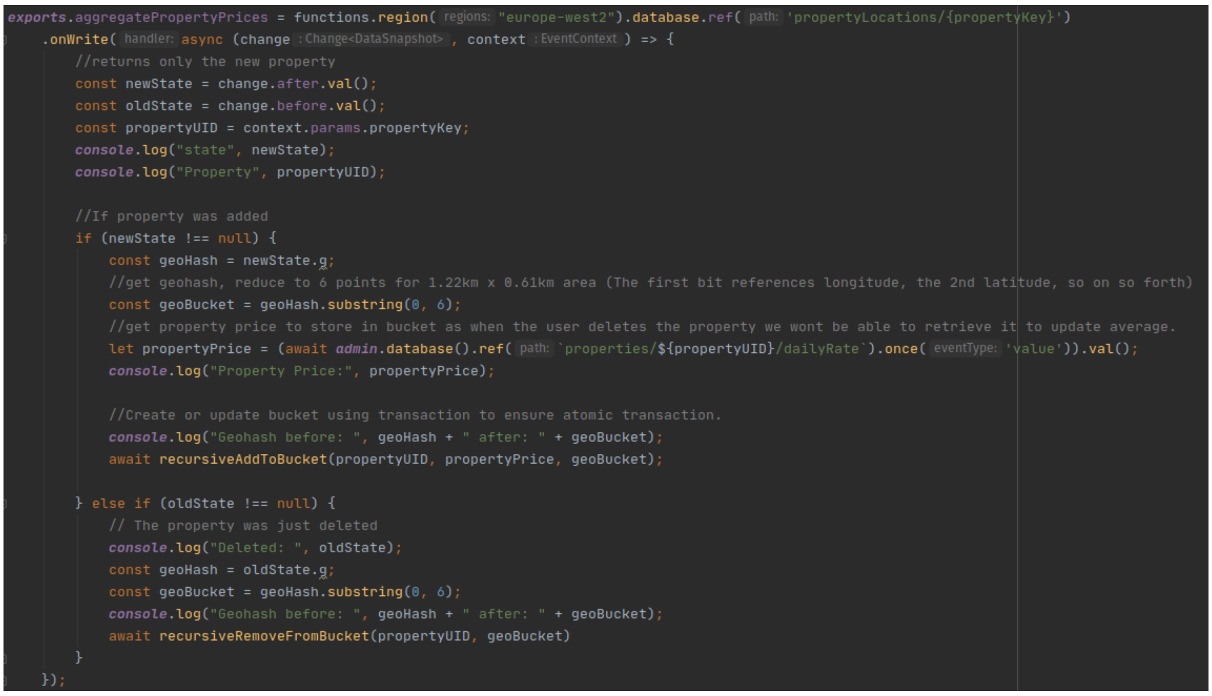
The Map ViewModel shows how a user's query is sent upstream to the Property Repository and then prepares the result for the View. A MutableLiveData list of keys is asynchronously returned from the call to “selectAllInRange”. These keys correspond to the properties in range of the user's search query. Initially when it returns the value will be null, but as property keys are found the propertyKeysLiveData will post the updated values from a background thread.

The average price is fetched by mapping the users given range to a geohash precision, e.g. a range of less than 2km is mapped to a geohash precision of length 6. This geohash will be then used as a key to look up the appropriate GeoPriceBucket which will contain the average price for properties falling under that geohash.

As mentioned, when property keys in range are found, the propertyKeysLiveData will emit values notifying observers, causing the “getPropertiesInRange” method to be called. This returns a LiveData list of properties matching the user's query.

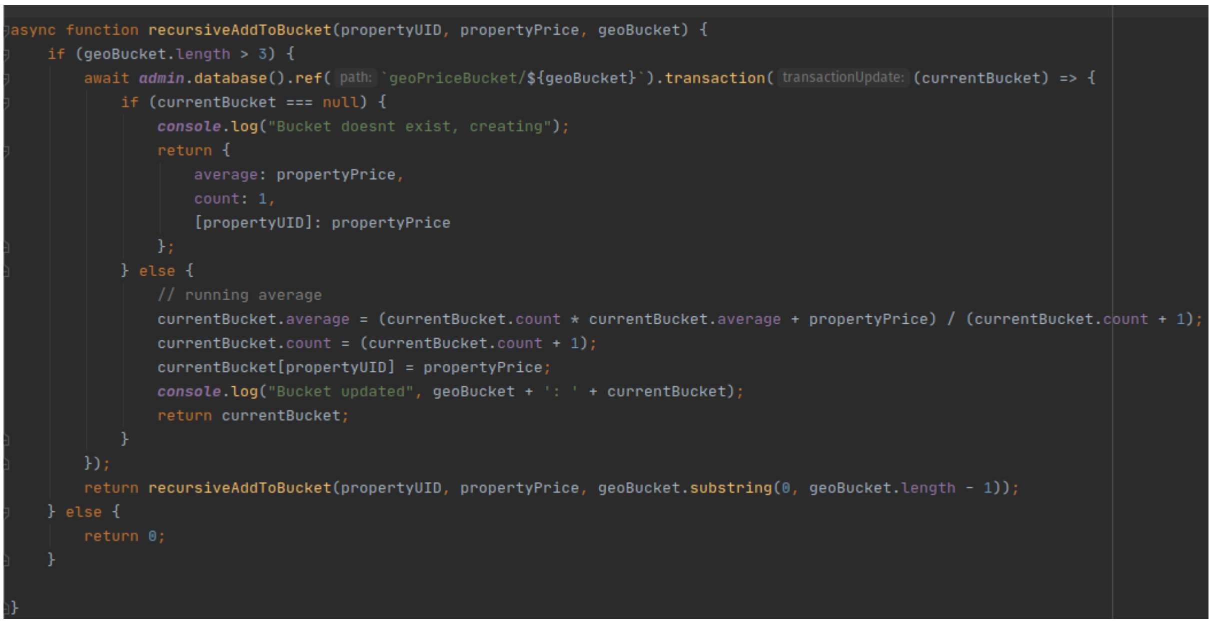
To combine both the averagePrice and propertiesInRangeLiveData a MediatorLiveData is used. MediatorLiveData takes another LiveData as a source and observes it, when triggered it will call the supplied function. So, both the average price and properties in range are added as sources. When either of them emits values, “update Properties” is called and checks to see if both the average and list of properties are present. If only one is ready the method returns null and the Map Fragment is not yet notified. When both are present, each property has its average price comparison calculated with respect to the area's average. Then the Map Fragment is notified of fresh values being emitted by the “queryPropertiesInRange” observer, and the “updateMap” and “refreshList” methods are called.

Cloud Functions: aggregatePropertyPrices()



```
exports.aggregatePropertyPrices = functions.region( regions: "europe-west2").database.ref( path: 'propertyLocations/{propertyKey}')
    .onWrite( handler: async (change :Change<DataSnapshot> , context :EventContext ) => {
        //returns only the new property
        const newState = change.after.val();
        const oldState = change.before.val();
        const propertyUID = context.params.propertyKey;
        console.log("state", newState);
        console.log("Property", propertyUID);

        //If property was added
        if (newState !== null) {
            const geoHash = newState.g;
            //get geohash, reduce to 6 points for 1.22km x 0.61km area (The first bit references longitude, the 2nd latitude, so on so forth)
            const geoBucket = geoHash.substring(0, 6);
            //get property price to store in bucket as when the user deletes the property we wont be able to retrieve it to update average.
            let propertyPrice = (await admin.database().ref( path: `properties/${propertyUID}/dailyRate`).once( eventType: 'value')).val();
            console.log("Property Price:", propertyPrice);

            //Create or update bucket using transaction to ensure atomic transaction.
            console.log("Geohash before: ", geoHash + " after: " + geoBucket);
            await recursiveAddToBucket(propertyUID, propertyPrice, geoBucket);

        } else if (oldState !== null) {
            // The property was just deleted
            console.log("Deleted: ", oldState);
            const geoHash = oldState.g;
            const geoBucket = geoHash.substring(0, 6);
            console.log("Geohash before: ", geoHash + " after: " + geoBucket);
            await recursiveRemoveFromBucket(propertyUID, geoBucket)
        }
    });
```

The above function is part of the Parklet Cloud functions module. It is triggered whenever a new property is added, or a property is removed under the propertyLocations node. We can see if a new property was added by checking the value of newState, if it is not null then it was just newly created. We then get the geohash value of the property's location, and trim it to length/precision of 6. The price of the property is also queried using the property key that is shared by both propertyLocations and the property itself. When this query is complete a call is then made to "recursiveAddToBucket".



```
async function recursiveAddToBucket(propertyUID, propertyPrice, geoBucket) {
    if (geoBucket.length > 3) {
        await admin.database().ref( path: `geoPriceBucket/${geoBucket}`).transaction( transactionUpdate: (currentBucket) => {
            if (currentBucket === null) {
                console.log("Bucket doesnt exist, creating");
                return {
                    average: propertyPrice,
                    count: 1,
                    [propertyUID]: propertyPrice
                };
            } else {
                // running average
                currentBucket.average = (currentBucket.count * currentBucket.average + propertyPrice) / (currentBucket.count + 1);
                currentBucket.count = (currentBucket.count + 1);
                currentBucket[propertyUID] = propertyPrice;
                console.log("Bucket updated", geoBucket + ': ' + currentBucket);
                return currentBucket;
            }
        });
        return recursiveAddToBucket(propertyUID, propertyPrice, geoBucket.substring(0, geoBucket.length - 1));
    } else {
        return 0;
    }

}
```

This function will recursively loop, calling itself with a reduced geohash until the length of which reaches 4, equivalent to a 39km x 19.5km bounding rectangle. On each pass of the function the presence of a “geoPriceBucket” is checked for. These represent a bucket of properties that fall under a common area. If the bucket already exists, the average price in an area is recalculated in a single pass, the count is increased and the property id and current price is added as a key value pair. This is in the case that if a property is ever removed that the average can be recalculated without having to fetch the price for every property. If it is an entirely new bucket, the average will be set to the price of the only property.

As this function runs 3 times for each property, it might be the case that under a geohash of length 6, no properties share the geohash, but as we begin to reduce the size of it, properties start to be grouped under the same bucket. For example, a property being the only one in an estate might share a geohash prefix of 4 at the town level with other properties and therefore be included in the average price. In the case of a property being removed, the geoPriceBucket is updated in much the same way.

Problems Solved

1. Changing workload:
   Initially the project's aim and workload were provisioned based on two people developing it. This ended up not being the case and as a solution, I reprioritized my time and the features that I implemented. This meant sometimes cutting back on certain features but ensuring that the core functionality of the app was still delivered.

   The agile approach in organising the work really helped in assessing what to prioritize on a week by week basis and making sure that there was a demo-able product at the end of each week.

2. Defining an "Area":
   When faced with adding the average price comparison for an area functionality, it was quite difficult to define how properties should be grouped. I wanted to make sure that an accurate average was given for each user's query. I had already utilized GeoFire to implement the property search, but due to how that worked with firebase it would not be suitable for storing precomputed averages without massive denormalization on the database.

   The solution was to group properties under common prefix geohashes and consult these "geoPriceBuckets" for an average, by mapping a user's query range to an approximate geohash. As detailed in the code sample, these buckets are created recursively and simulate an "road", "estate" and "town" level of accuracy for the average.

3. Unit testing:
   As noted in the testing documentation I faced difficulties with testing my Fragments and Activities due to framework issues. The solution was to implement use case scenario testing with high coverage, and Firebase TestLab instrumentation tests to compensate for the missing unit tests.

4. Duplicate Vehicles/Lifecycle issue:
   Initially when the vehicle fragment was being implemented, if a list of vehicles belonging to a user was fetched and then the vehicles page was repeatedly navigated to, the vehicles list would grow exponentially.

   I had assumed this was similar to another issue I had with properties which involved how I implemented a query. This was not the case and in fact due to the improper setting of the ViewModel lifecycle owner. When the Vehicle Fragment was being destroyed the ViewModel was not, which resulted in multiple observers being attached to the Vehicle ViewModel, resulting in the users list of vehicles growing with duplicates. This required quite a few hours of debugging before I realised my mistake and properly set ownership of the ViewModel and observer.

5. MVVM Java

   When initially proposing the project idea, it was considered whether to implement it using Java or Kotlin. I chose Java due to being slightly more familiar with it, and I had not developed in an object orientated language at this scale in a while. Both my third-year project and internship were spent developing in Clojure a functional lisp-based language.

   However, when it came midway through developing ParkLet I realised that the level of support for Android MVVM in Java had dropped significantly in comparison to the Kotlin libraries. Even the android example documentation was now mostly Kotlin, with some in Java. Some of the Java libraries were even deprecated. So, some compromises had to be made in terms of architecture decisions.

   For example, I could not fully use two-way data binding as it would require writing custom adapters for each form or object and would quickly lead to duplicate code as the adapters cannot be generified. And I was unwilling to reference android components in ViewModels as this would lead to life cycle leaks. So as a compromise, some of the presentation logic remains in the Fragment classes.

Results

I am quite pleased with the level of functionality achieved in ParkLet considering the above difficulties. As mentioned, some functionality had to be postponed, such as the review system and QR code based check-in. However, the app was still designed with these features in mind and they can easily be later implemented due to the architecture's modularity and decoupling.

I believe the app could be further extended to other areas, such as function room rental, combined with the NFC check-in functionality.

I quite enjoyed working on an Android application and utilising NFC technology. Working using the MVVM and Repository pattern was challenging but rewarding in the experience I gained. I wish to continue working on the app, implementing some of the future work mentioned below.

Future Work

Development:

I'd like to investigate porting the app over to Kotlin to take advantage of some of the newer libraries and fully utilizing the MVVM pattern, extracting logic out of the views and making them as passive as possible to make unit testing them even easier.

To reduce some of the boilerplate, an abstract BaseFragment class could be written which the existing Fragments could then be refactored to extend. An interface could be written for each repository that could then be implemented by concrete classes. If it was decided to change the database, the new repository classes can just implement the methods specified in the interface.

Quality of life changes:

1. Make it easier for homeowners to check the price in their area while they are creating a new property.
2. Editable properties, let a user set when exactly a property is available to rent for events, i.e. Slane.
3. Allow homeowners to set a time range for parking.
4. Give renters the ability to filter by price on the map view.
5. Filter for booking type on the Homepage.

New Features:

1. Messaging:
   Implement private messaging between a renter and homeowner. This can be accomplished by extending the functionality of the ParkLet Firebase Messaging Service class.

2. QR code generation:
   Alternative form of checking for homeowners that don't have access to NFC tags or an NFC enabled phone. A utility class can be created to provide QR code generation and scanning. This can then call to the same Home ViewModel method as the NFC check-in version as it would be storing and reading the same info as an NFC tag.

3. Shortest route:
   Instead of a user stating where they wish to park, instead allow them to enter their end destination and calculate the closest available parking space with respect to their current location and their end destination. This would have been an interesting problem to solve however it would rely on using an API to get valid routes rather than using "as the crow flies" distances. Google maps API provides this but it must be utilised on their maps only. So, alternatives such as OpenMaps could be used instead.

References

1. 2016 Commuters Census
   https://www.cso.ie/en/releasesandpublications/ep/p-cp6ci/p6cii/p6www/
2. Recommended App Architectures:
   https://developer.android.com/jetpack/docs/guide
3. Android GeoFire
   https://github.com/firebase/geofire-android
4. Dagger
   https://dagger.dev/dev-guide/android.html
5. Authentication in MVVM tutorial
   https://medium.com/firebase-tips-tricks/how-to-create-a-clean-firebase-authentication-using-mvvm-37f9b8eb7336
6. DateRangePicker
   https://github.com/savvisingh/DateRangePicker