



Agenda

Technical Specification

Conor Smyth 12452382

Supervisor: Dr. Geoff Hamilton

Blog: <http://blog.smythconor.com/>

Email: cnrsmyth@gmail.com

Contents

[Contents](#)

[Abstract](#)

[Introduction](#)

[Overview](#)

[Glossary](#)

[Motivation](#)

[Design](#)

[High Level Design](#)

[Architecture Overview Diagram](#)

[Use cases](#)

[Registration](#)

[Create Group](#)

[Add member](#)

[Create Calendar Entry](#)

[Create To-Do List Entry](#)

[Send Message](#)

[System Architecture Diagram](#)

[User Interface - Android Application](#)

[Tomcat Server - Spring](#)

[Database - MySQL](#)

[Implementation](#)

[Database](#)

[Server](#)

[MyBatis/Dao](#)

[Service](#)

[Controller](#)

[Exception Handling](#)

[Encryption](#)

[Model](#)

[Maven](#)

[Jackson](#)

[User Interface](#)

[Utility Packages](#)

- [Adapter](#)
- [Cache](#)
- [Constants](#)
- [Dialog](#)
- [Helpers](#)
- [Utils](#)
- [Screen Packages](#)
 - [Agenda](#)
 - [Calendar](#)
 - [Group](#)
 - [Messaging](#)
 - [Registration](#)
 - [Todolist](#)
- [Service](#)
- [Validation](#)
 - [Unit Testing](#)
 - [Functional Testing](#)
 - [User Acceptance Testing](#)
- [Problems solved](#)
 - [User Interface Service Calls](#)
 - [Calendar View](#)
 - [To-Do List Sorting](#)
 - [Stored Procedures](#)
 - [DateTimes with MySQL](#)
 - [Server Exception Handling](#)
 - [Messaging with GCM](#)
- [Results](#)
- [Future work](#)
- [References](#)

Abstract

Agenda is an android application that allows groups of people to communicate and collaborate easily. The application allows users to register and login. Upon logging in a user can create a group and be added to a group. A group consists of a collection of users and communication and collaboration tools such as a Shared Calendar, Instant Messaging System and a Shared To-Do List. Users can create entries in the calendar and to-do list that will be shared with the other users. Users can message each other in real time with the instant messaging system.

Introduction

Overview

Agenda is an android application designed to allow groups of people collaborate and communicate more efficiently. Agenda offers groups of people the ability to create groups and use built in communication and collaboration tools.

The main tools are a shared calendar, messaging system and a shared to do list. All of these features are available through a group that users can create and invite people to join. The shared calendar is a calendar view in which users can create an entry which will be shared with other users in the group. The To-Do list is a simple to do list which allows users to create items of work that need to be done and they can assign priority for the item. The Messaging system is a simple messaging system where users can instant message each other. The application requires an internet connection as most data is stored and retrieved from the server and database.

Glossary

- Android: The target phone architecture used for developing the user interface.
- GCM: Google Cloud Messaging, an API provided by google for connecting user applications for broadcasting of messages.
- REST: Underlying framework for network communications using HTTP keywords like GET, POST, PUT and DELETE. Stateless protocol meaning clients are independent of the server allowing for transparency and scalability between how the resources are gathered and supplied. Primary means of communication is JSON.
- JSON: Javascript Object Notation, package information sent by the server for easy consumption from the client. Also allows for scalability as the data sent is uniform and cross platform.
- Spring: Java REST web framework that allows for clean implementation of RESTful web services.
- MyBatis: Data access framework that allows for clean and simple database communication with a database.
- MySQL: Relational Database for storing data for the application based on SQL.
- Tomcat: Apache server containers that integrates well with Java for deploying and hosting Java web application.
- Maven: Dependency management tool for build automation.
- JUnit: Java unit testing library for simple and effective implementation of unit tests.

Motivation

The motivation behind this project was to develop an application that can be used by groups to help collaborate and communicate easily. The application combines a number of helpful tools to allow groups of people communicate efficiently.

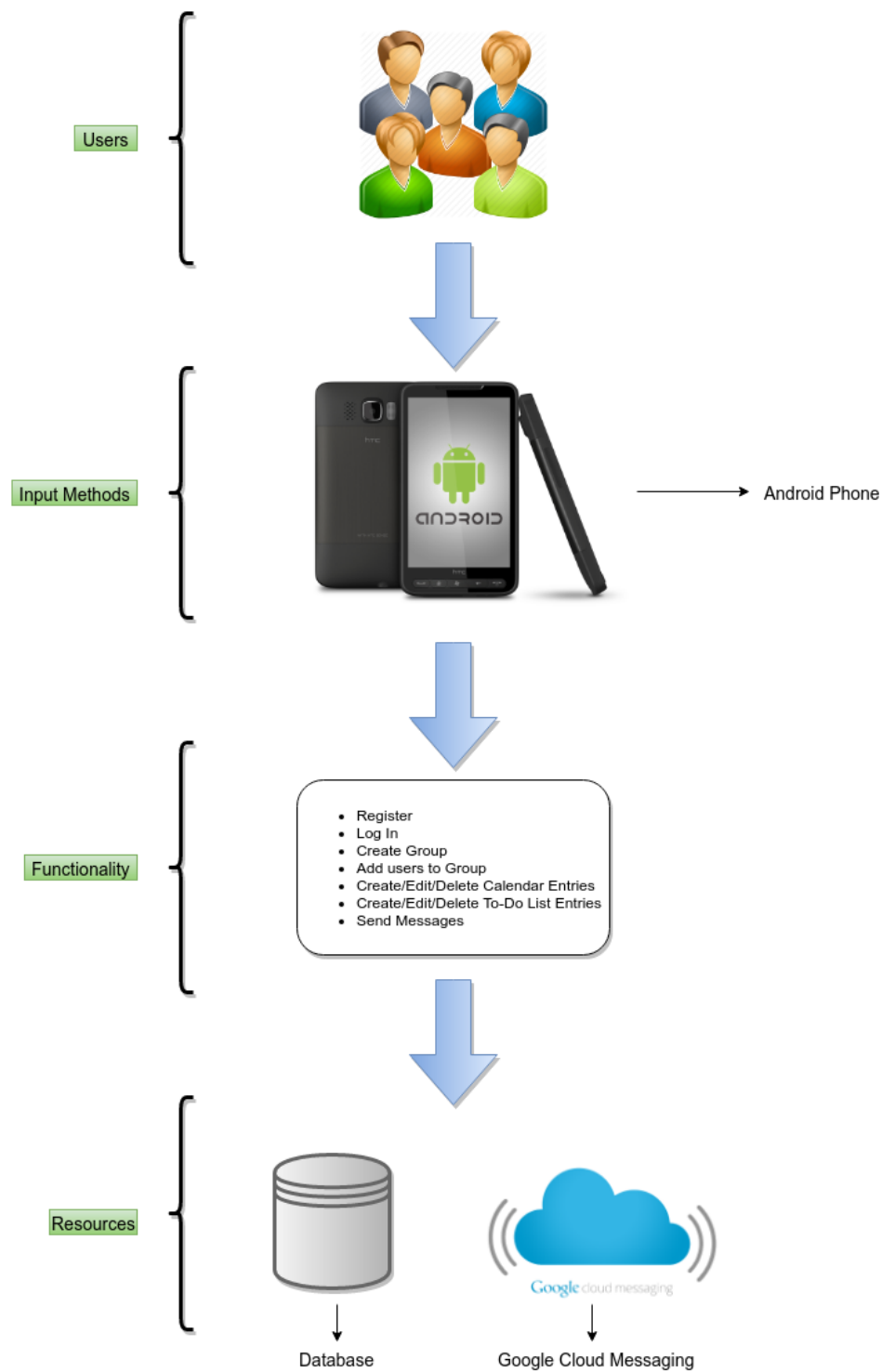
The idea came from myself and my roommates struggling to keep track of household tasks such as when bills were due and when the bins had to go out. I think this application would be a huge help to people in similar situations. The application wouldn't be just limited to roommates it would be very beneficial for all type of groups, ranging from families trying organise chores for their children to college classmates who need to keep track of assignment deliverables and work that needed to be done.

My first idea was to build a cross platform API and have Android and iOS client application however this would be out of scope for this project. The API is setup to work cross-platform however I have focused on delivering an Android application as I had never worked on Android before and felt it would be a suitable challenge.

Design

High Level Design

Architecture Overview Diagram

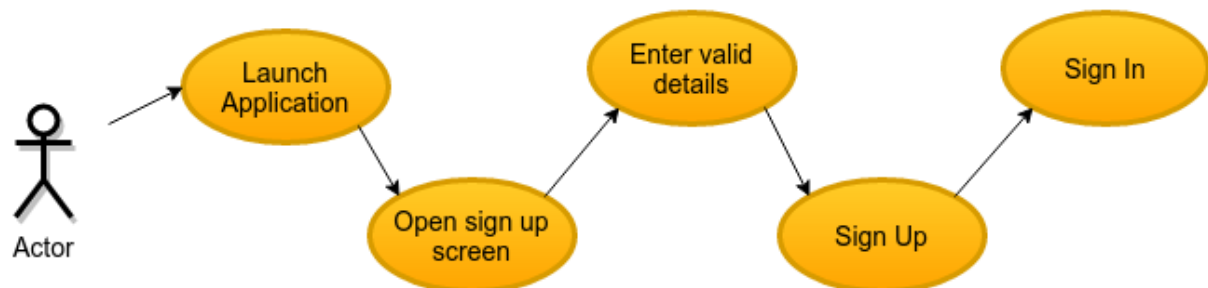


Here is an architecture overview diagram for the system. This gives an easy to understand overview of how the system is structured. This diagram shows how a user will interact with the system through an android phone, to use the functionality provided where any data that needs to be stored is stored in a database.

Use cases

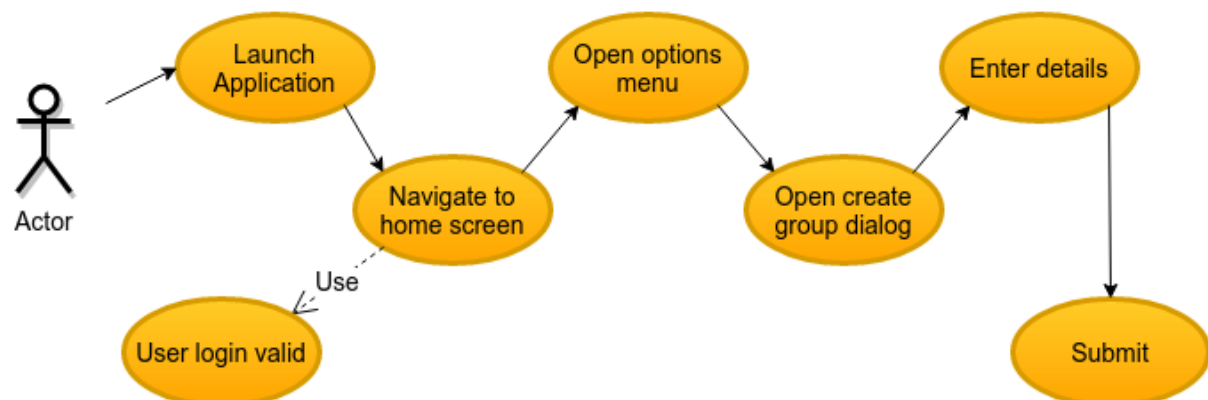
Here are six architecturally significant use case diagrams that show how the user interacts with the system for key functionality within the system. I chose these diagrams as I feel they are the most important and show the key functionality that a user expects from the system.

Registration



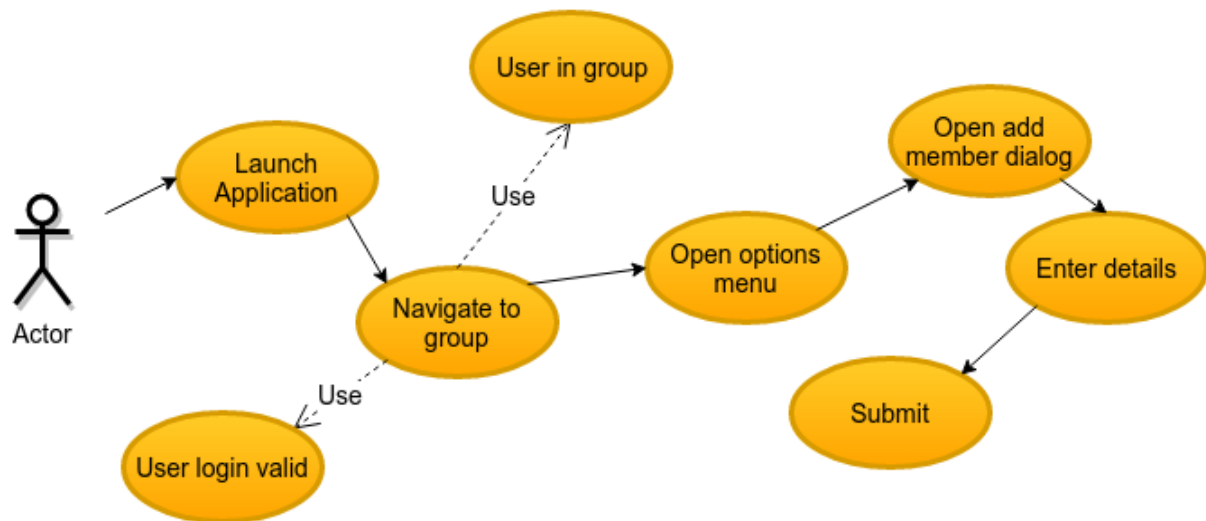
This use case diagram shows how a user will sign up to the application. The user must navigate the sign up screen upon launching the application. From there they will enter their details and press sign up. Upon successful signup, the user will be notified that their registration is successful and will be prompted to sign into the application with the email address and password that they supplied.

Create Group



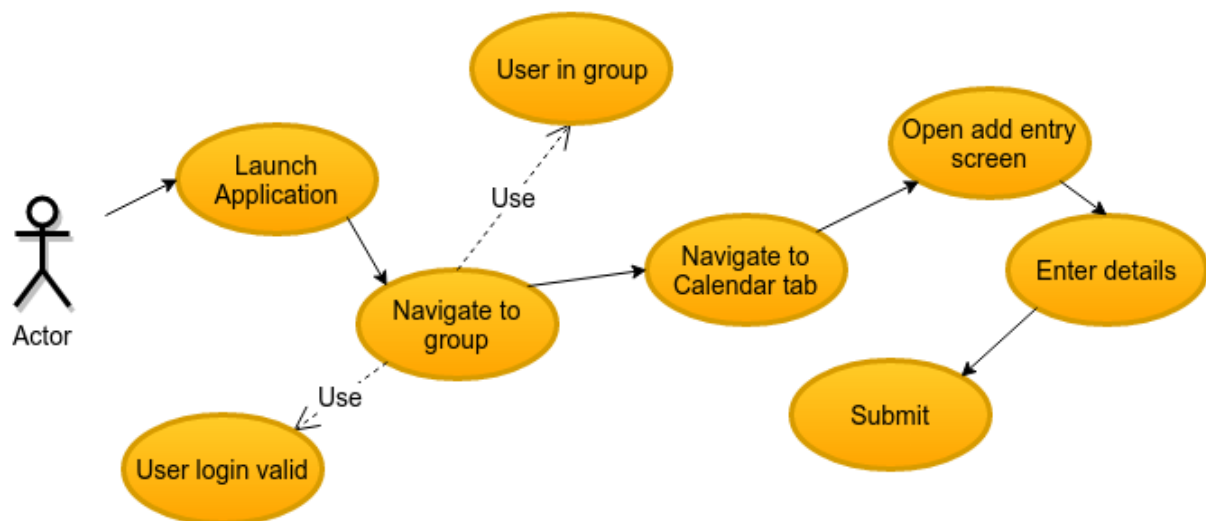
This use case diagram shows how a user will create a group in the application. Assuming the user has successfully registered and signed in, they can navigate to the home screen where they can create a new group from the option menu in the toolbar. Here they will be prompted for a name and then they will be notified upon successful creation. The new group will show on screen and they will have all group functionality among that group.

Add member



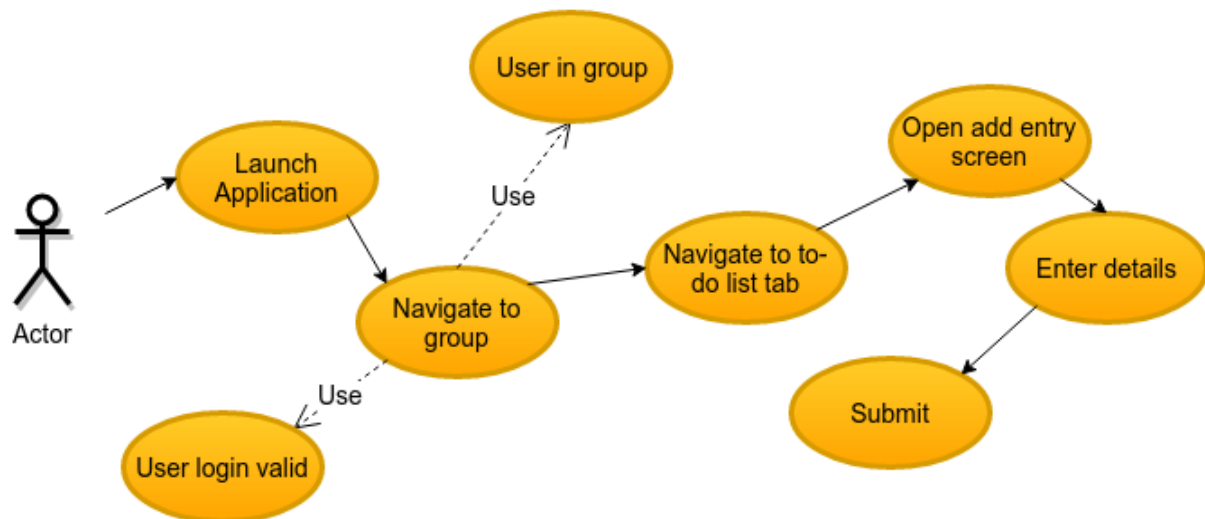
This use case diagram shows how a user can add another user to a group. Upon navigating to a group screen, they will have the ability to add another user to the group. They will be prompted for the email address of the other user and if the input is valid, the user entered will be added to the group.

Create Calendar Entry



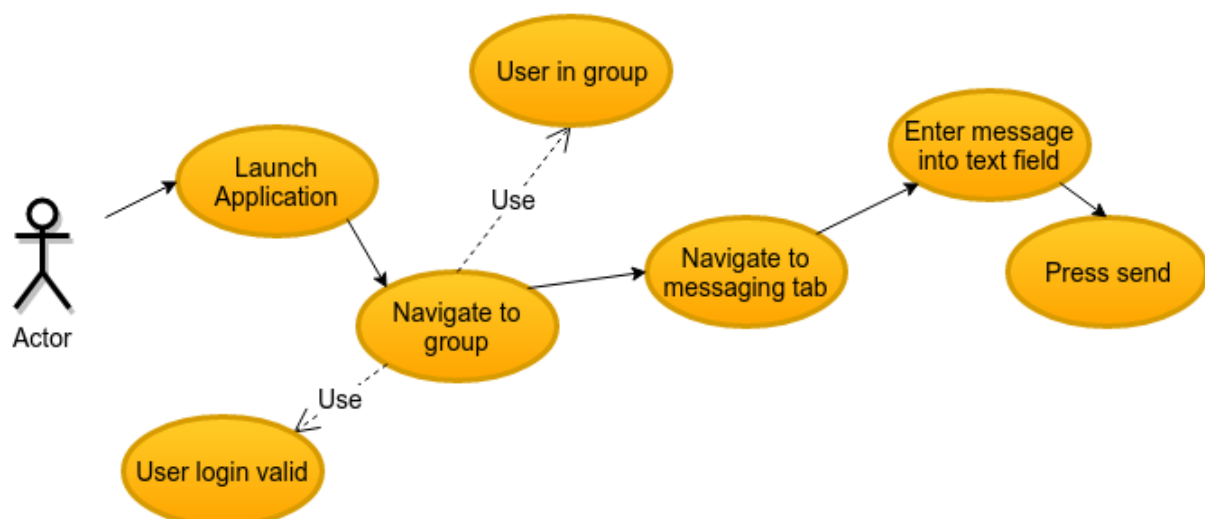
This use case diagram shows how a user can create a Calendar entry. Upon navigating to the Calendar screen inside a group, the user will be able to press on the add entry button which will open up a new screen where they can enter in the details of the entry. Upon clicking create entry, the user will receive confirmation of the entry creation assuming all inputs are valid.

Create To-Do List Entry



This use case diagram shows how a user can create a To-Do list entry. Upon navigating to the To-Do List screen inside a group, the user will be able to press on the add entry button which will open up a new screen where they can enter in the details of the entry. Upon clicking create entry, the user will receive confirmation of the entry creation assuming all inputs are valid.

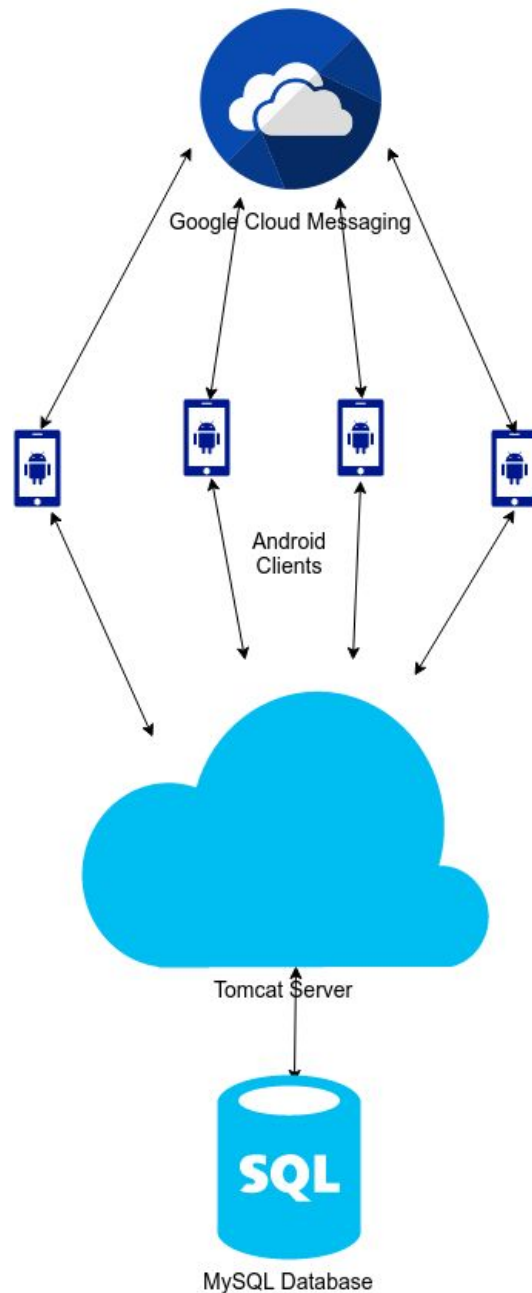
Send Message



This use case diagram shows how a user can send a message to other users in a group. Upon navigating to the messages screen the user can enter a message into the text field at the bottom of the screen. Upon pressing send the user's message will be sent to all other members of the group and will be shown in the list of messages on the screen. The messages are displayed as the message with the name of the person that send the message.

System Architecture Diagram

Here we have a system architecture diagram. This diagram shows how the various components of the system will interact with each other and how the three tier layering of the system is structured. I chose this three tier layering as it allows for good abstraction between the various components and allows for easier maintainability and scalability. The application client will communicate with the server and the server will communicate with the database. This also shows how the client applications connect with GCM.



User Interface - Android Application

The android application uses Spring's RestTemplate object to execute calls to the server at specified endpoints and relevant data.

All of the service calls will be done on a spin off thread from the main UI thread to stop the UI thread from blocking which service calls are being made.

When the user signs up to the application, a random salt is generated and concatenated with the password, which is then hashed, in order to ensure the password is stored securely.

The user interface for the home screen is a list of the groups for a user allowing for clear identification of the groups the user is involved with. The user interface for a group is a tabbed interface with the calendar, messaging and to do list on separate tabs.

The android application interacts with the Google Cloud Messaging API for messaging functionality. The users get a registration id from GCM which is used to add them to a group under a notification key that is generated upon group creation.

Tomcat Server - Spring

The server that the application will communicate is hosted on a tomcat server. This tomcat server is a web container that allows for easy deployment of Java web applications.

The server specifies the endpoints for the RESTful API for the client application to call for the various elements of functionality that require server communication.

The server itself has a three layer architecture for levels of transparency and division of work among different components in the server itself. There will be more detail in the implementation section.

The various data types that are required for the user data are declared in a POJO format and are used for handling the marshalling and unmarshalling of data to and from the UI.

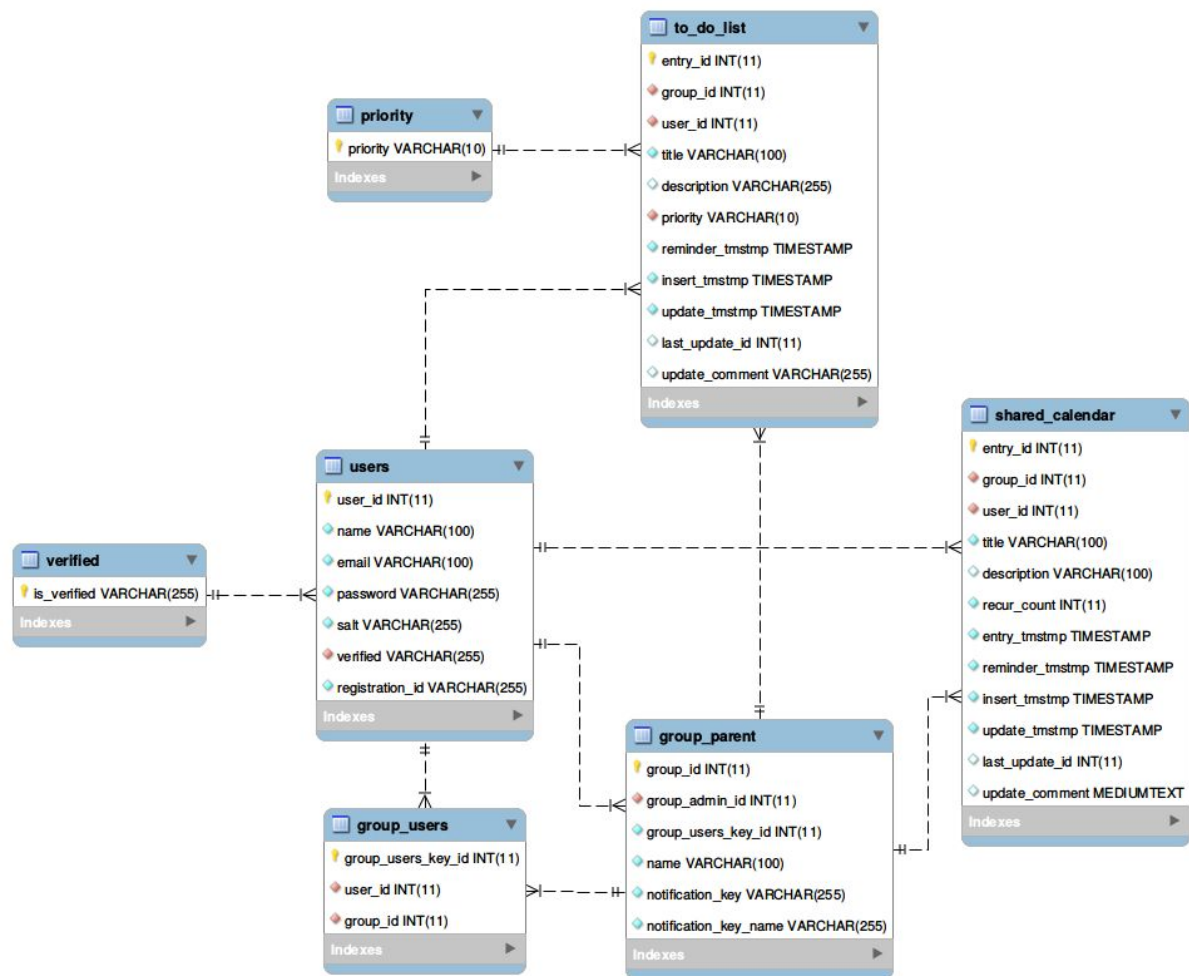
There is a global exception handling mechanism to prevent the server from crashing from defects introduced through the code. This exception handling handles how exception are caught and communicated back the client application so that a well structured framework is in place and the service remains intact in the case of a defect.

The server communicates with the database for storing and retrieving all data.

Database - MySQL

The database for storing the data is a MySQL database. MySQL is a relational database that perform well.

Here is an Entity Relationship diagram that shows how the various tables and entities are connected with each other.



Implementation

Database

The database implementation is completed entirely using MySQL. The database has a user called `mt_user` that is the user for the application server to use to connect and execute calls to the database. This user has a secure password that can only be accessed by the server.

I implemented an automated database deployment script that I could use to quickly create a database whenever changes were made or features were added. This script allows for quick and easy deployment so the database is relatively plug and play i.e if I needed to change the server to a new machine, I could simply run this script to create the database. This would have to change if the application was released as there would be data that would need to be backed up and restored however for this project it made the database work very efficient. MySQL allows for easy backing up of data so updating the script to include a backup and restore would be relatively simple. The script takes advantage of the directory structure implemented clearly consolidating table creation, seeding, package creation and grants. Here is an example of the script:

```
0
1
2
3
4
5
6
7 source ../ddl/create_table_verified.sql
8 source ../ddl/create_table_priority.sql
9 source ../ddl/create_table_users.sql
10 source ../ddl/create_table_group_parent.sql
11 source ../ddl/create_table_group_users.sql
12 source ../ddl/create_seq_group_users_key_id.sql
13 source ../ddl/create_table_shared_calendar.sql
14 source ../ddl/create_table_to_do_list.sql
15
16 source ../dml/insert_into_verified.sql
17 source ../dml/insert_into_priority.sql
18 source ../dml/insert_into_seq_grp_usr_key_id.sql
19
20 source ../package/get_user_by_email.sql
21 source ../package/create_user.sql
22
23 source ../package/create_group.sql
24 source ../package/get_groups_for_user.sql
25 source ../package/add_group_member.sql
26 source ../package/update_name_group.sql
27 source ../package/remove_group_member.sql
28
29 source ../package/get_entries_calendar.sql
30 source ../package/create_entry_calendar.sql
31 source ../package/update_entry_calendar.sql
32 source ../package/delete_entry_calendar.sql
33 source ../package/get_entries_calendar_by_id.sql
34
35 source ../package/get_entries_todo_list.sql
36 source ../package/create_entry_todo_list.sql
37 source ../package/update_entry_todo_list.sql
38 source ../package/delete_entry_todo_list.sql
39 source ../package/get_entries_todo_list_by_id.sql
40
41 source ../dml/seed_users.sql
42 source ../dml/seed_groups.sql
43 source ../dml/seed_calendar_entries.sql
44 source ../dml/seed_todo_list_entries.sql
45
46 source ../grants/grant_users.sql
47 source ../grants/grant_create_group.sql
48 source ../grants/grant_get_group_for_user.sql
49 source ../grants/grant_add_group_member.sql
50
51 FLUSH PRIVILEGES;
```

The table creation scripts are all consolidated into one directory, ddl, which stands for Data Definition language. This allows for uniform addition of tables. Any seeding required for these tables, such as the Priority table, is consolidated into the dml folder. This stands for data manipulation language and this sets up the data required for certain tables. An example of each are shown below.

```
1 CREATE TABLE IF NOT EXISTS agenda.users
2 (
3     user_id INTEGER NOT NULL AUTO_INCREMENT,
4     name VARCHAR(100) NOT NULL,
5     email VARCHAR(100) NOT NULL UNIQUE,
6     password VARCHAR(255) NOT NULL,
7     salt VARCHAR(255) NOT NULL,
8     verified VARCHAR(255) NOT NULL,
9     registration_id VARCHAR(255) NOT NULL,
10    PRIMARY KEY (user_id),
11    FOREIGN KEY (verified) REFERENCES verified(is_verified)
12 );
```

- DDL:

```
1 INSERT INTO priority
2     (priority)
3 VALUES ('LOW');
4
5 INSERT INTO priority
6     (priority)
7 VALUES ('MEDIUM');
8
9 INSERT INTO priority
10    (priority)
11 VALUES ('HIGH');
```

- DML:

The server communicates with the database through stored procedures. These stored procedures are used as an interface for the server so that the underlying call to the database is transparent. This allows for easier data model and implementation changes that are oblivious to the server. Here is an example of a stored procedure that is very simple to the server but relatively complex:

```

1 DELIMITER //
2 CREATE PROCEDURE
3   create_group(IN i_group_admin_id INTEGER,
4               IN i_name VARCHAR(255),
5               IN i_notification_key VARCHAR(255),
6               IN i_notification_key_name VARCHAR(255))
7 BEGIN
8   UPDATE group_users_key_id_seq
9   SET   id=last_insert_id(id + 1);
10
11   INSERT INTO group_parent
12   (
13       group_users_key_id,
14       group_admin_id,
15       name,
16       notification_key,
17       notification_key_name
18   )
19   SELECT id,
20          i_group_admin_id,
21          i_name,
22          i_notification_key,
23          i_notification_key_name
24   FROM   group_users_key_id_seq;
25
26   INSERT INTO group_users
27   (
28       group_users_key_id,
29       user_id,
30       group_id
31   )
32   SELECT group_users_key_id,
33          i_group_admin_id,
34          group_id
35   FROM   group_parent
36   WHERE  group_id=Last_insert_id();
37 END//
38 DELIMITER ;
39

```

The database is hosted on the same machine as the server on a Digital Ocean ubuntu instance.

Server

The server is a Java web service built using the Spring framework. The server handles communication with the database, processing information and then sending that information on to the user and visa versa. The server is hosted on a Digital Ocean ubuntu instance.

The implementation of the server is a three tiered architecture which separates out the responsibilities of the various different components in the server. I will go through the functionality of each layer and how they all tie together. Here is a screenshot of the package structure showing logical separation between the layers and various snippets of functionality:



MyBatis/Dao

MyBatis is an easy to use framework that allows for clean implementation of database calls through defining the calls in XML and using Data access Objects (DAO) to call the procedures.

The MyBatis implementation starts with defining the things such as type adapters for certain data types and aliases for the custom object that you use. You define the calls to the database through XML assigning a mapper to a DAO. These mappers are implementations of DAO interface functions that make the calls to the stored procedure in the database. You start by defining how MyBatis should map the data sent and returned from the database by define a result map for each object. The DAO interface that applies to the mapper file is defined at the top in mapper namespace so that correct functions are called. You can assign custom type handlers to the result properties so that MyBatis can properly handle the types and marshal them into Java objects. Here is an example of the result map:


```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
3 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
4
5 <mapper namespace="co.agendaapp.dao.CalendarDao">
6   <resultMap id="calendarMap" type="calendarEntry">
7     <id property="entryId" column="entry_id" />
8     <result property="groupId" column="group_id" />
9     <result property="userId" column="user_id" />
10    <result property="title" column="title" />
11    <result property="description" column="description" />
12    <result property="recurCount" column="recur_count" />
13    <result property="entryTimestamp" column="entry_tmstmp"
14      typeHandler="co.agendaapp.util.DateTimeTypeHandler" />
15    <result property="reminderTimestamp" column="reminder_tmstmp"
16      typeHandler="co.agendaapp.util.DateTimeTypeHandler" />
17    <result property="insertTimestamp" column="insert_tmstmp"
18      typeHandler="co.agendaapp.util.DateTimeTypeHandler" />
19    <result property="lastUpdateId" column="last_update_id" />
20    <result property="updateTimestamp" column="update_tmstmp"
21      typeHandler="co.agendaapp.util.DateTimeTypeHandler" />
22    <result property="updateComment" column="update_comment" />
23  </resultMap>
24

```

Here is an example a type handler for the date times between java and SQL:

```

16 @MappedTypes(DateTime.class)
17 public class DateTimeTypeHandler implements TypeHandler {
18
19   @Override
20   public Object getResult(ResultSet rs, String columnName)
21     throws SQLException {
22     Timestamp rowValue = rs.getTimestamp(columnName);
23
24     return rowValue != null ? new DateTime(rowValue.getTime()) : null;
25   }
26
27   @Override
28   public Object getResult(ResultSet rs, int columnIndex) throws SQLException {
29     return null;
30   }
31
32   @Override
33   public Object getResult(CallableStatement cs, int columnIndex)
34     throws SQLException {
35     return null;
36   }
37
38   @Override
39   public void setParameter(PreparedStatement ps, int columnIndex,
40     Object parameter, JdbcType jdbcType) throws SQLException {
41     DateTime dateTime = (DateTime) parameter;
42     if(dateTime != null) {
43       long timeInMillis = dateTime.getMillis();
44
45       Timestamp date = new Timestamp(timeInMillis);
46
47       ps.setTimestamp(columnIndex, date);
48     } else {
49       ps.setTimestamp(columnIndex, null);
50     }
51   }
52 }

```

This type handler is applied where defined to data type going to and from the database so that data is communicated properly.

The stored procedures are called through calls on the DAO interface objects which are then mapped to defined procedures in the XML. The XML defines the inputs and outputs of the stored procedure and are given names equal to the interface functions in the DAO objects. Here is an example of a DAO interface:

```

5 import co.agendaapp.model.Group;
6
7 import org.apache.ibatis.annotations.Param;
8
9 public interface GroupDao {
10
11     /**
12      * Dao function to retrieve the groups for a user ID
13      * @param userId the id of the user of the groups
14      * @return list of groups the user is a member of
15      */
16     public List<Group> getGroups(Integer userId);
17
18     /**
19      * May not need this, leaving for now
20      * Dao function to retrieve the group selected
21      * @param groupId the id of the group to get
22      * @return group with the selected group ID
23      */
24     public Group getGroup(Integer groupId);
25
26     /**
27      * Dao function to create a group
28      * @param group the group to create
29      * @return the group created
30      */
31     public Group createGroup(Group group);
32
33     /**
34      * Add a user to a group
35      * @param user the user to add
36      */
37     public void addMember(
38         @Param("groupId") Integer groupId,
39         @Param("userId") Integer user);
40
41     /**
42      * Dao function to remove a member from a group
43      * @param groupId the id of the group to remove from
44      * @param userId the id of the user to remove
45      */
46     public void removeMember(
47         @Param("groupId") Integer groupId,
48         @Param("userId") Integer userId);
49
50     /**
51      * Dao function to update the group name
52      * @param groupId the id of the group
53      * @param name the name to change it to
54      */
55     public void updateName(
56         @Param("groupId") Integer groupId,
57         @Param("name") String name);
58 }

```

Here is the corresponding XML which defines the name, which must be equal to the interface function and the body of the call which is essentially a function call for the database. You can define the input and output type through parameterType and resultMap which marshalls the objects in the Java POJO objects.

```

14 <select id="getGroups" parameterType="int" resultMap="groupMap">
15 {
16   call get_groups_for_user
17   (
18     #{id, jdbcType = INTEGER, mode = IN}
19   )
20 }
21 </select>
22
23 <select id="createGroup" resultMap="groupMap" parameterType = "group" statementType = "CALLABLE">
24 {
25   call create_group
26   (
27     #{groupAdminId, jdbcType = INTEGER, mode = IN},
28     #{name, jdbcType = VARCHAR, mode = IN},
29     #{notificationKey, jdbcType = VARCHAR, mode = IN},
30     #{notificationKeyName, jdbcType = VARCHAR, mode = IN}
31   )
32 }
33 </select>
34
35 <select id="addMember" parameterType="map" statementType="CALLABLE">
36 {
37   call add_group_member
38   (
39     #{groupId, jdbcType = INTEGER, mode = IN},
40     #{userId, jdbcType = VARCHAR, mode = IN}
41   )
42 }
43 </select>
44
45 <select id="removeMember" parameterType="map" statementType="CALLABLE">
46 {
47   call remove_group_member
48   (
49     #{groupId, jdbcType = INTEGER, mode = IN},
50     #{userId, jdbcType = INTEGER, mode = IN}
51   )
52 }
53 </select>
54
55 <select id="updateName" parameterType="map" statementType="CALLABLE">
56 {
57   call update_name_group
58   (
59     #{groupId, jdbcType = INTEGER, mode = IN},
60     #{name, jdbcType = VARCHAR, mode = IN}
61   )
62 }
63 </select>

```

The DAO interface objects are injected as beans to the service layer. The bean definitions go into the application-context so that Spring knows where the mappers and DAO objects are located. Here is where the beans are defined:

```

30
31 <!-- Configure bean to convert JSON to POJO and vice versa -->
32 <bean id="jsonMessageConverter"
33   class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter" />
34
35 <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
36   <property name="dataSource" ref="dataSource" />
37   <property name="configLocation" value="classpath:mybatis/SqlConfig.xml" />
38   <property name="mapperLocations" value="classpath:mybatis/mappers/*.xml" />
39 </bean>
40
41 <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer"
42   scope="prototype">
43   <property name="basePackage" value="co.agendaapp.dao" />
44 </bean>
45

```

This structure of the MyBatis/DAO layer allows for separation of the work of all database procedures. This also allows for specific unit testing of the data access to help minimise the scope of the search for fixing a bug.

Service

The service layer handles all business logic and calls to the DAO layer. The service objects handle things such as checking user existence or user validation upon signing in. This removes the need for the DAO layer to handle any of this logic. The service layer is called upon by the controller layer which takes in the requests. Here is an example of a service layer object:

```

8 import co.agendaapp.encrypt.EncryptionUtils;
9 import co.agendaapp.encrypt.Encryptor;
10 import co.agendaapp.exception.AppLayerException;
11 import co.agendaapp.model.User;
12 /**
13  * Log in service implementation
14  * @author Conor Smyth <cnrsmyth@gmail.com>
15  * @since 2016-02-17
16  * @version 0.1
17  */
18 @Service
19 public class LogInServiceImpl implements LogInService {
20
21     @Autowired
22     private UserDetailsDao userDetailsDao;
23
24     /**
25      * Log In service implementation
26      * @param user User to log in
27      * @throws AppLayerException
28      */
29     public User logIn(User user) throws AppLayerException {
30         User dbEntry = userDetailsDao.getUserByEmail(user.getEmail());
31
32         if(dbEntry == null) {
33             throw new AppLayerException("User does not exist", HttpStatus.NOT_FOUND);
34         }
35
36         String inputPassword = user.getPassword();
37         String storedSalt = dbEntry.getSalt();
38
39         byte[] bytePassword = EncryptionUtils.stringToBytes(inputPassword);
40         byte[] salt = EncryptionUtils.hexToBytes(storedSalt);
41
42         String storedPassword = dbEntry.getPassword();
43         String hashedPassword = Encryptor.hashPassword(bytePassword, salt);
44
45         if(!hashedPassword.equals(storedPassword)) {
46             throw new AppLayerException("Incorrect Password", HttpStatus.UNPROCESSABLE_ENTITY);
47         }
48
49         User returnUser = new User();
50
51         returnUser.setUserId(dbEntry.getUserId());
52         returnUser.setName(dbEntry.getName());
53         returnUser.setEmail(dbEntry.getEmail());
54         returnUser.setIsVerified(dbEntry.getIsVerified());
55
56         return returnUser;
57     }
58 }

```

Each service layer class is annotated with the `@Service` annotation so Spring can identify the function of the class. Here you can see that the service layer handles ensuring the user exists if the user is trying to sign in and also ensuring that the password they have entered is correct. The `UserDetailsDao` bean is autowired into the class so the functions of the DAO object can be called. This function throws an `AppLayerException` which is the exception handling mechanism implemented for the server. I will go into more detail on this later. Spring handles the injection of the beans, which are declared as prototype beans so they can be used more than once.

Controller

The controller layer defines the endpoints for the RESTful API. These are the URIs that the client application can hit in order to use the functionality provided by the server. The controller layer can only communicate with the service layer and completely handles the receiving and returning of requests. This removes the responsibility of business logic from the controllers and the request handling from the service layer. Here is an example of a controller for the server:


```

29 @Controller
30 @RequestMapping(value = "/calendar")
31 public class CalendarController {
32     final Logger logger = LoggerFactory.getLogger(CalendarController.class);
33
34     @Autowired
35     private CalendarService calendarService;
36
37     /**
38      * Controller function to get entries for a group
39      * @param groupId id of the group
40      * @return the groups entries
41      */
42     @RequestMapping(value = "/getEntries/{groupId}", method = GET)
43     public ResponseEntity<List<CalendarEntry>> getCalendarEntries(@PathVariable Integer groupId) {
44
45         List<CalendarEntry> calendarEntries = calendarService.getEntries(groupId);
46
47         return new ResponseEntity<>(calendarEntries, HttpStatus.OK);
48     }
49
50     /**
51      * Controller function to create an entry for a group
52      * @param calendarEntry the calendar entry to create
53      * @return the created calendar entry
54      */
55     @RequestMapping(value = "/createEntry", method = POST)
56     public ResponseEntity<CalendarEntry> createCalendarEntry(@RequestBody CalendarEntry calendarEntry) {
57         calendarService.createEntry(calendarEntry);
58         System.out.println("Here");
59
60         return new ResponseEntity<>(calendarEntry, HttpStatus.CREATED);
61     }
62
63     /**
64      * Controller function to update a calendar entry for a group
65      * @param calendarEntry the calendar entry to update with
66      * @return the updated calendar entry
67      * @throws AppLayerException for stale update
68      */
69     @RequestMapping(value = "/updateEntry", method = PUT)
70     public ResponseEntity<CalendarEntry> updateCalendarEntry(@RequestBody CalendarEntry calendarEntry)
71         throws AppLayerException {
72         calendarService.updateEntry(calendarEntry);
73
74         return new ResponseEntity<>(calendarEntry, HttpStatus.OK);
75     }
76 }

```

Each controller object is annotated with `@Controller` to define it as a controller so that Spring can handle them properly. As you can see each function here is an endpoint for the API. The root endpoint is declared at the beginning with annotation `@RequestMapping` and then each function is given the next value in the URI as well as the HttpMethod that the endpoint accepts using the annotation `@RequestMapping`.

The service objects are also injected into this layer as singleton beans, using the annotation `@Autowired`, that get destroyed after one use. This adds memory efficiency as the objects only live as long as they need.

The controller takes in data in two different ways, through path variables and request bodies. The path variables are values in the URI itself and are extracted as parameters to the function call that are annotated with the `@PathVariable`. The RequestBody parameters are JSON objects send to the server that are unmarshalled into the objects defined. These parameters are annotated as `@RequestBody` to define that the parameter will come in the body of the call to the endpoint. The controllers return the Spring type `ResponseEntity`, which is a type that allows you to send a type as the body with a `HttpStatus` code indicating the response status code for the client application to know whether the call was a success or not.

Any controller that could result in an exception from the service layer throw the exception that was thrown from the service layer as they are handled by a global exception handler that defines how to handle the exceptions and send them on.

Exception Handling

Exceptions for the server are a custom exception type called `AppLayerException`. These exceptions are the only exceptions that are thrown by the controllers that are handled by the global exception handler as other exceptions are unexpected and should be fixed. Here is the global exception handler:

```
1 package co.agendaapp.exception;
2
3 import org.slf4j.Logger;
4 import org.slf4j.LoggerFactory;
5 import org.springframework.dao.DuplicateKeyException;
6 import org.springframework.http.HttpStatus;
7 import org.springframework.http.ResponseEntity;
8 import org.springframework.web.bind.annotation.ControllerAdvice;
9 import org.springframework.web.bind.annotation.ExceptionHandler;
10 import org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;
11
12 /**
13  * Controller to handle exception and return appropriate message
14  * @author Conor Smyth <cnrsmyth@gmail.com>
15  * @since 21 Mar 2016
16  * @version 0.1
17  */
18 @ControllerAdvice
19 public class AppLayerExceptionHandler extends ResponseEntityExceptionHandler {
20
21     final Logger logger = LoggerFactory.getLogger(AppLayerExceptionHandler.class);
22
23     /**
24      * Handle custom exceptions
25      * @param ex the exception thrown
26      * @return ResponseEntity with error message and status
27      */
28     @ExceptionHandler(AppLayerException.class)
29     public ResponseEntity<Errors> handleAppLayerException(AppLayerException ex) {
30         Errors err = new Errors();
31         HttpStatus status = ex.getStatus();
32
33         err.setMessage(ex.getMessage());
34
35         logger.error(err.toString());
36
37         return new ResponseEntity<Errors>(err, status);
38     }
39
40     /**
41      * Duplicate key handler
42      * @param ex the exception thrown
43      * @return ResponseEntity with the error message and status
44      */
45     @ExceptionHandler(DuplicateKeyException.class)
46     public ResponseEntity<Errors> handleDuplicateKeyException(DuplicateKeyException ex) {
47         Errors err = new Errors();
48         HttpStatus status = HttpStatus.UNPROCESSABLE_ENTITY;
49
50         String errorMessage = "Error message: User already exists";
51
52         err.setMessage(errorMessage);
53
54         return new ResponseEntity<Errors>(err, status);
55     }
56 }
```

This class acts as a controller as annotated by the `@ControllerAdvice` and applies to all controllers in the package. The type of exception to handle is defined in the `@ExceptionHandler` annotation and the function to handle that exception is then called. This is a clean of handling user exceptions that should not affect the functionality of the server.

Encryption

There is an encryption package that handles all the encryption related functionality. This package has a utility class and an encryptor class for executing the encryption. The utility class handles the generation of 128-bit random salts for password hashing. This class also offers other utility functions such as concatenating the byte arrays and converting the hashed passwords to Hexadecimal notation for storage in the database. Here is a screenshot of the utility class:

```
3 import static java.nio.charset.StandardCharsets.UTF_8;
4 import java.math.BigInteger;
5 import java.util.Random;
6
7 public class EncryptionUtils {
8     public static final String HASH_TYPE = "SHA-256";
9
10    public static byte[] getSalt() {
11        byte[] salt = new byte[16];
12
13        new Random().nextBytes(salt);
14
15        return salt;
16    }
17
18    public static byte[] concatArray(byte[] a, byte[] b) {
19        byte[] c = new byte[a.length + b.length];
20
21        System.arraycopy(a, 0, c, 0, a.length);
22        System.arraycopy(b, 0, c, a.length, b.length);
23
24        return c;
25    }
26
27    public static String bytesToHex(byte[] data) {
28        return (new BigInteger(data)).toString(16);
29    }
30
31    public static byte[] hexToBytes(String data) {
32        return (new BigInteger(data, 16)).toByteArray();
33    }
34
35    public static byte[] stringToBytes(String str) {
36        return str.getBytes(UTF_8);
37    }
38 }
```

The main encryption class uses these encryption utilities for executing the encryption. This encryption class takes a password and salt as byte arrays and concatenates them and hashes the password for secure storage. Here is a screenshot of the encryption class:

```

1 package co.agendaapp.encrypt;
2
3 import java.security.MessageDigest;
4 import java.security.NoSuchAlgorithmException;
5
6 import org.slf4j.Logger;
7 import org.slf4j.LoggerFactory;
8
9 public class Encryptor {
10     final static Logger logger = LoggerFactory.getLogger(Encryptor.class);
11     private static MessageDigest messageDigest;
12
13     /**
14      * Hash the password with the salt and return as string
15      * @param password as byte array
16      * @param salt as byte array
17      * @return hashed password as string
18      */
19     public static String hashPassword(byte[] password, byte[] salt) {
20         byte[] dataToHash = EncryptionUtils.concatArray(password, salt);
21
22         try {
23             messageDigest = MessageDigest.getInstance(EncryptionUtils.HASH_TYPE);
24         } catch (NoSuchAlgorithmException e) {
25             logger.debug("Error getting digest type");
26         }
27
28         messageDigest.update(dataToHash);
29
30         byte[] digest = messageDigest.digest();
31
32         return EncryptionUtils.bytesToHex(digest);
33     }
34 }

```

Model

I separated all of my models into a separate package for holding the basic data structures required for data storage and manipulation. These classes implement Serializable so that they can be serialized and sent over a network. Jackson handles the marshalling and unmarshalling of these objects into JSON objects. These models are exactly the same as the client application so that marshalling and unmarshalling is uniform. Here is a small snippet of one of the models:

```

1 package co.agendaapp.model;
2
3 import java.io.Serializable;
4
5 import org.joda.time.DateTime;
6
7 import com.fasterxml.jackson.annotation.JsonInclude;
8 import com.fasterxml.jackson.annotation.JsonInclude.Include;
9
10 @JsonInclude(Include.NON_NULL)
11 public class CalendarEntry implements Serializable {
12     private static final long serialVersionUID = 8017249840010395872L;
13
14     private Integer entryId;
15     private Integer groupId;
16     private Integer userId;
17     private String title;
18     private String description;
19     private Integer recurCount;
20     private DateTime entryTimestamp;
21     private DateTime reminderTimestamp;
22     private DateTime insertTimestamp;
23     private DateTime updateTimestamp;
24     private Integer lastUpdateId;
25     private String updateComment;
26
27     /**
28      * @return the entryId
29      */
30     public Integer getEntryId() {
31         return entryId;
32     }
33
34     /**
35      * @param entryId the entryId to set
36      */
37     public void setEntryId(Integer entryId) {
38         this.entryId = entryId;
39     }
40 }

```


Maven

Maven is a dependency management tool for handling whatever external libraries that I need for the server. You define the name of the jar that you need along with the version you want and maven downloads and applies it to your project. Maven also handles build automation so it compiles all the files, packages them and outputs it into a war file for deployment onto the server. You can specify the version of java compiler you wish to use. The dependencies are defined in a pom.xml file which maven parses and downloads the relevant packages. Here is a snippet of the pom file for the server:

```
34 <build>
35   <plugins>
36     <plugin>
37       <groupId>org.apache.maven.plugins</groupId>
38       <artifactId>maven-compiler-plugin</artifactId>
39       <version>3.1</version>
40       <configuration>
41         <target>${java.version}</target>
42         <source>${java.version}</source>
43       </configuration>
44     </plugin>
45   </plugins>
46 </build>
47
48 <dependencies>
49   <dependency>
50     <groupId>com.fasterxml.jackson.core</groupId>
51     <artifactId>jackson-databind</artifactId>
52     <version>2.7.3</version>
53   </dependency>
54
55   <!-- Spring MVC -->
56   <dependency>
57     <groupId>org.springframework</groupId>
58     <artifactId>spring-webmvc</artifactId>
59     <version>${spring-framework.version}</version>
60   </dependency>
61
```

As you can see the compiler version is set as a property (Java 8) for maven to build the project with and an example of two dependencies are shown at the bottom where maven looks to see the name and version of the package to download.

Jackson

Jackson is a framework for automatic marshalling and unmarshalling of java objects to JSON objects. Jackson is automatically applied to the object for network communications.

User Interface

The user interface is an android application written in Java. The UI interacts with the server and external APIs such as GCM. The packages for the UI are structured in a logical fashion with a package dedicated to each screen and it's various related classes. Here is a screenshot of the packages:

■	adapter
■	agenda
■	cache
■	calendar
■	constants
■	dialog
■	encrypt
■	exception
■	group
■	helpers
■	messaging
■	model
■	registration
■	service
■	todolist
■	utils

Utility Packages

There are various utility packages setup to handle various utility functions.

Adapter

The adapter package stores all the adapters for list views. Adapters allow you to display a complex object in a custom list view, but still have all the information for that object in the background for list item selection. Here is an example of an adapter:

```

24 public class ToDoListEntryAdapter extends ArrayAdapter<ToDoListEntry> {
25     private Context context;
26     private List<ToDoListEntry> values;
27
28     public ToDoListEntryAdapter(Context context, List<ToDoListEntry> values) {
29         super(context, android.R.layout.simple_list_item_1, values);
30
31         this.context = context;
32         this.values = values;
33     }
34
35     @Override
36     public View getView(int position, View convertView, ViewGroup parent) {
37         View rowView = convertView;
38
39         if(rowView == null) {
40             LayoutInflater inflater = (LayoutInflater) context.getSystemService(Context
41                 .LAYOUT_INFLATER_SERVICE);
42
43             rowView = inflater.inflate(R.layout.to_do_list_item, parent, false);
44         }
45
46         TextView priorityView = (TextView) rowView.findViewById(R.id.priority);
47         TextView titleView = (TextView) rowView.findViewById(R.id.titleField);
48         TextView descriptionView = (TextView) rowView.findViewById(R.id.descriptionField);
49
50         if(priorityView != null) {
51             Priority priority = values.get(position).getPriority();
52
53             String strPriority = "";
54             Resources resources = context.getResources();
55
56             if(priority == Priority.HIGH) {
57                 strPriority = "H";
58                 priorityView.setBackgroundColor(resources.getColor(R.color.highPriority));
59             } else if(priority == Priority.MEDIUM) {
60                 strPriority = "M";
61                 priorityView.setBackgroundColor(resources.getColor(R.color.mediumPriority));
62             } else if(priority == Priority.LOW) {
63                 strPriority = "L";
64                 priorityView.setBackgroundColor(resources.getColor(R.color.lowPriority));
65             }
66
67             priorityView.setText(strPriority);
68         }
69         if(titleView != null) {
70             String titleValue = values.get(position).getTitle();
71
72             titleView.setText(titleValue);
73         }
74         if(descriptionView != null) {
75             String description = values.get(position).getDescription();
76
77             descriptionView.setText(description);
78         }
79
80         return rowView;

```

This class extends ArrayAdapter which is what android uses for ListViews. Here you can see that the data is passed through to getView, which is used to display the list item. Here you can define how the view will look fill out the details of the view using the object that was passed to the function.

Cache

This package handles calls to read and write from the cache. I have implemented a generic read and write for easy reading and writing. The class CacheHelper helps as a facade for writing to cache and read or writes whatever value is supplied as a parameter to a specific file, defined in CacheFileNames. Here is an example of the class:

```

10 import java.io.FileNotFoundException;
11 import java.io.IOException;
12 import java.io.InputStream;
13
14 import co.agendaapp.constants.Tag;
15
16 /**
17  * Cache Helper class for storing things in the cache
18  * @author Conor Smyth <cnrsmyth@gmail.com>
19  * @version 0.1
20  * @since 2016-03-02
21  */
22 public class CacheHelper {
23
24     public static void write(File cacheDir, String fileName, Object obj) {
25         try {
26             File cache = getCache(cacheDir, fileName);
27
28             new ObjectMapper().writeValue(cache, obj);
29         } catch (FileNotFoundException e) {
30             Log.e(Tag.TAG, "Unable to open cache");
31         } catch (IOException e) {
32             Log.e(Tag.TAG, e.getMessage());
33         }
34     }
35
36     /**
37      * Read an object from a cache file
38      * @param cacheDir the get directory
39      * @param fileName the file to read
40      * @param clazz the type of object to read
41      * @return the object read
42      */
43     public static Object read(File cacheDir, String fileName, Class<?> clazz) {
44         Object obj = new Object();
45
46         try {
47             File cache = getCache(cacheDir, fileName);
48
49             InputStream in = new BufferedInputStream(new FileInputStream(cache));
50
51             obj = new ObjectMapper().readValue(in, clazz);
52
53             in.close();
54         } catch (FileNotFoundException e) {
55             Log.e(Tag.TAG, "Unable to open cache");
56         } catch (IOException e) {
57             Log.e(Tag.TAG, "Unable to read cache");
58         }
59
60         return obj;
61     }
62
63     public static File getCache(File cacheDir, String fileName) {
64         return new File(cacheDir, fileName);
65     }
66 }

```

Constants

This package holds various constants that need to remain consistent across the application. There are various constants defined in this package that are a central point of reference so that the values used are consistent. Here's an example of the ServiceConstants class that defines the constants relative to API calls:

```

1 package co.agendaapp.constants;
2
3 import android.util.Log;
4
5 import org.springframework.http.HttpStatus;
6
7 import java.util.Arrays;
8 import java.util.List;
9
10 /**
11  * Class for service constants
12  * @author Conner Smyth <cnrsmyth@gmail.com>
13  * @version 0.1
14  * @since 2016-01-31
15  */
16 @SuppressWarnings("unused")
17 public class ServiceConstants {
18     public static final String BASE_URL = "http://178.62.98.122:8080/agenda-mt";
19     public static final String LOCAL_URL = "http://10.0.2.2:8080/agenda-mt";
20     public static final String TEMP_URL = "http://192.168.0.24:8080/agenda-nt";
21
22     public static final String SIGN_UP_URI = "/signUp";
23     public static final String SIGN_IN_URI = "/signIn";
24     public static final String GROUP_URI = "/group";
25     public static final String CALENDAR_URI = "/calendar";
26     public static final String TO_DO_LIST_URI = "/ToDoList";
27
28     public static final String GCM_CREATE_URL = "https://android.googleapis.com/gcm/notification";
29     public static final String GCM_AUTH_KEY = "key=AizaSyD_anE-fYU6aFw-0wrNAmMVP36-3nZ0xSE";
30     public static final String GCM_PROJECT_ID = "795145934105";
31     public static final String GCM_OPERATION = "operation";
32     public static final String GCM_REG_IDS = "registration_ids";
33     public static final String GCM_NOTIFICATION_KEY_NAME = "notification_key_name";
34     public static final String GCM_NOTIFICATION_KEY = "notification_key";
35     public static final String GCM_GROUP_ADD = "add";
36     public static final String GCM_GROUP_CREATE = "create";
37     public static final String GCM_SEND = "https://gcm-http.googleapis.com/gcm/send";
38     public static final String GCM_TO = "to";
39     public static final String GCM_DATA = "data";
40     public static final String GCM_MESSAGE = "message";
41
42     public static final String NET_ERR = "Network error";
43
44     private static final List<HttpStatus> statuses = Arrays.asList(
45         HttpStatus.OK,
46         HttpStatus.CREATED,
47         HttpStatus.ACCEPTED,
48         HttpStatus.UNAUTHORIZED,
49         HttpStatus.UNPROCESSABLE_ENTITY,
50         HttpStatus.NOT_FOUND,
51         HttpStatus.INTERNAL_SERVER_ERROR
52     );
53 }

```

Dialog

This package holds generic dialog interfaces that can be used by different screens for common functionality. This helps avoid code repetition by offering some parameters to set so that any relevant dialog can be created. Here's an example of a generic AddCreate dialog used multiple times for different things:


```

17 /**
18  * General Purpose dialog for Single field input field
19  * @author Connor Smyth <cnrsmyth@gmail.com>
20  * @version 0.1
21  * @since 2016-04-03
22  */
23 public class AddCreateDialog extends DialogFragment {
24     private String hint;
25     private String message;
26     private String positiveButton;
27
28     AddMemberDialogListener mListener;
29
30     public void setHint(String hint) {
31         this.hint = hint;
32     }
33
34     public void setMessage(String message) {
35         this.message = message;
36     }
37
38     public void setPositiveButton(String positiveButton) {
39         this.positiveButton = positiveButton;
40     }
41
42     public interface AddMemberDialogListener {
43         public void onDialogPositiveClick(DialogFragment dialog, String value);
44     }
45
46     @Override
47     public Dialog onCreateDialog(final Bundle savedInstanceState) {
48         final AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
49
50         final EditText input = new EditText(getActivity().getApplicationContext());
51
52         input.setTextColor(Color.BLACK);
53         input.setHintTextColor(Color.GRAY);
54         input.setInputType(InputType.TYPE_CLASS_TEXT);
55         input.setHint(hint);
56
57         builder.setView(input);
58         builder.setMessage(message).setPositiveButton(positiveButton, new DialogInterface.OnClickListener() {
59             public void onClick(DialogInterface dialog, int id) {
60                 String email = input.getText().toString().trim().toLowerCase();
61
62                 if(!isBlank(email)) {
63                     mListener.onDialogPositiveClick(AddCreateDialog.this, email);
64                 } else {
65                     Toast.makeText(getActivity().getApplicationContext(), "Field cannot be blank", Toast.LENGTH_SHORT).show();
66                 }
67             }
68         }).setNegativeButton(CANCEL, new DialogInterface.OnClickListener() {
69             public void onClick(DialogInterface dialog, int id) {
70                 dismiss();
71             }
72         });
73     }

```

Helpers

This package holds classes that offer helping functions across the application. The purpose of this package was to avoid code repetition across the codebase. Here is an example of the BundleHelper class:

```

1 package co.agendaapp.helpers;
2
3 import android.os.Bundle;
4 import android.util.Log;
5
6 import com.fasterxml.jackson.databind.ObjectMapper;
7
8 import java.io.IOException;
9
10 import co.agendaapp.constants.BundleConstants;
11 import co.agendaapp.constants.Tag;
12 import co.agendaapp.model.Group;
13
14 /**
15  * Helper class to get selected group
16  * @author Conor Smyth <cnrsmyth@gmail.com>
17  * @version 0.1
18  * @since 2016-04-09
19  */
20 public class BundleHelper {
21     public static Group getSelectedGroup(Bundle extras) {
22         String value = (String) extras.get(BundleConstants.GROUP);
23
24         try {
25             return new ObjectMapper().readValue(value, Group.class);
26         } catch (IOException e) {
27             Log.e(Tag.TAG, "Error reading selected group");
28         }
29
30         return null;
31     }
32 }

```

This class is used multiple times across the application for extracting the selected group from the bundle data structure passed across screens. It's also acts as a facade for the objectMapper reads to help keep the main code cleaner.

Utils

This package contains general utility classes that don't fit into other packages. They offer similar benefits as the other packages. Here is an example of a class:

```

10 /**
11  * Compator for to do list entry list
12  * @author Conor Smyth <cnrsmyth@gmail.com>
13  * @since 2016-05-16
14  */
15 public class ToDoListComparator implements Comparator<ToDoListEntry> {
16     @Override
17     public int compare(ToDoListEntry lhs, ToDoListEntry rhs) {
18         if (lhs != null && rhs != null) {
19             DateTime ldt = lhs.getReminderTimestamp();
20             DateTime rdt = rhs.getReminderTimestamp();
21
22             Priority l = lhs.getPriority();
23             Priority r = rhs.getPriority();
24
25             if (ldt.equals(rdt)) {
26                 return comparePriorities(l, r);
27             } else if (ldt.isBefore(rdt)) {
28                 return comparePriorities(l, r);
29             } else {
30                 return comparePriorities(l, r);
31             }
32         }
33
34         return -1;
35     }
36
37     private int comparePriorities(Priority lhs, Priority rhs) {
38         int compared = lhs.compareTo(rhs);
39
40         if (compared == 0) {
41             return 0;
42         } else if (compared < 0) {
43             return -1;
44         } else {
45             return 1;
46         }
47     }
48 }

```

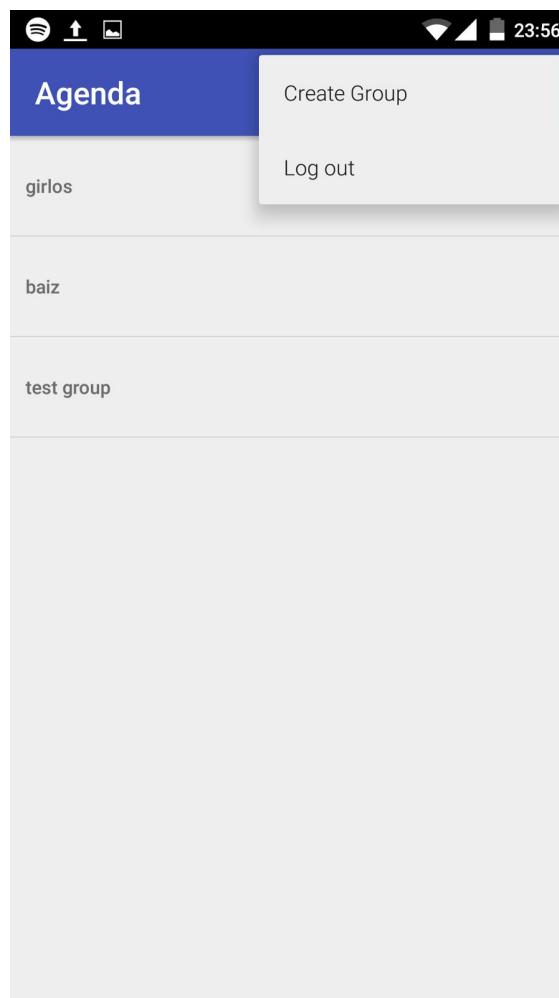
This class offer a clean way of sorting the to do list entry list based on reminder timestamps and priority, so the most recent, highest priority items are sorted to the top.

Screen Packages

These are the packages that control the screens, defining their views and their functionality:

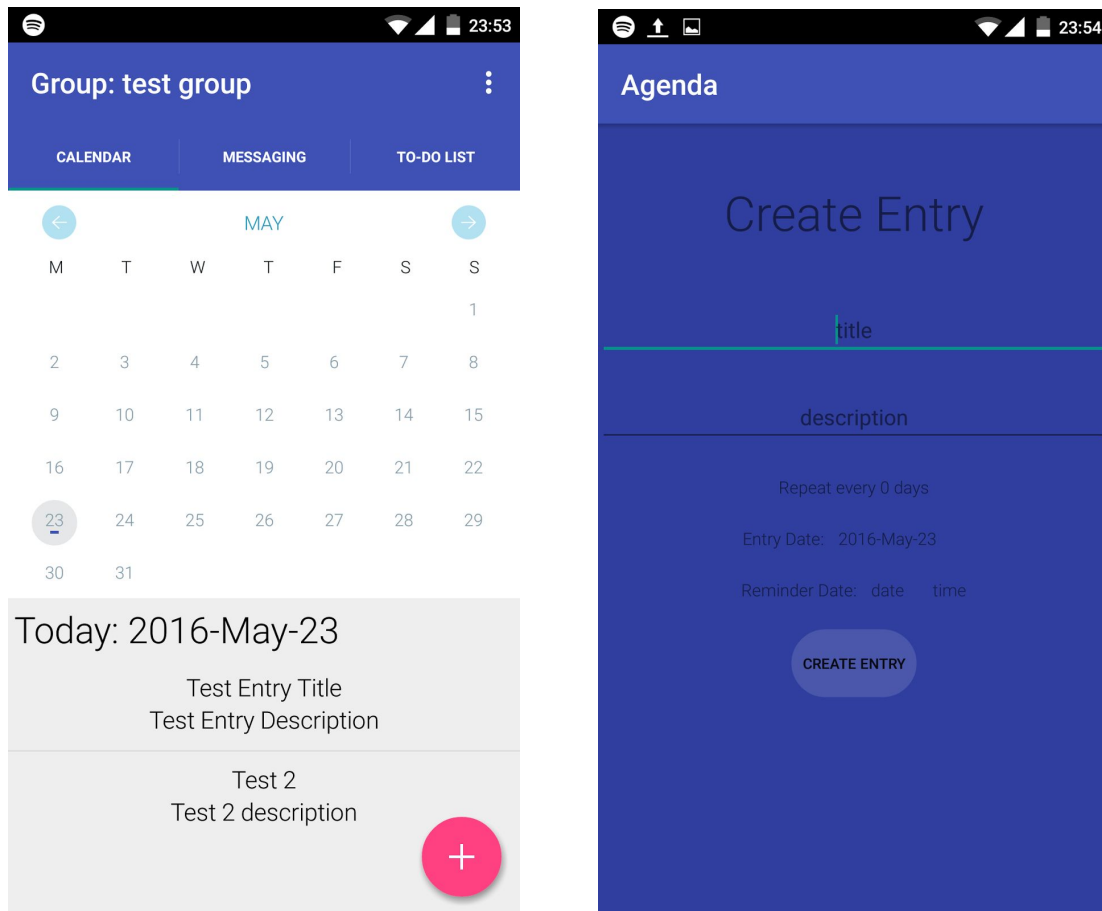
Agenda

This package holds the home screen. The home screen is the main screen that the user sees that contains the list of groups that user is associated with. Users can create groups on this screen, log out and enter a group. Here's a screenshot of the screen with the menu open:



Calendar

This package holds the screens for the calendar. The calendar consists of a calendar view and a list of entries for the selected day. There is an add entry button where the user will be navigated to a new screen where they can enter the details for the calendar entry. The user's entries are displayed by small marks under the date for each entry. When the user selects that day, the list at the bottom populates with the entries for that day where they can select an entry. Here's two screenshots showing the calendar view and the create entry screen:



Group

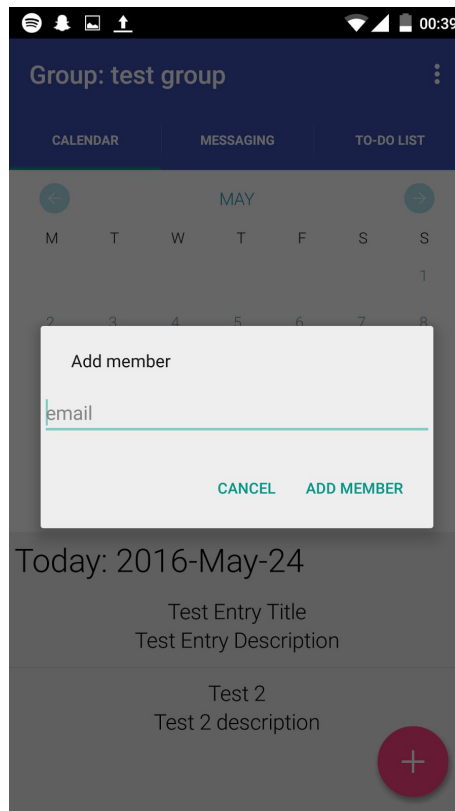
This package holds the tab manager for the screens within a group. In android, in order to create a tabbed set of views like above, you must have a tab pager adapter. This adapter handles the changing of the tabs, ensuring the correct screen is displayed when the selection changes. Here's the code that handles the changing of the screens:

```

8 import co.agendaapp.group.GroupScreen;
9 import co.agendaapp.calendar.CalendarScreen;
10 import co.agendaapp.todolist.ToDoListScreen;
11 import co.agendaapp.messaging.MessagingScreen;
12
13 /**
14  * Fragment tab page manager
15  * @author Conor Smyth <cnrsmyth@gmail.com>
16  * @version 0.1
17  * @since 2016-04-01
18  */
19 public class TabsPagerAdapter extends FragmentPagerAdapter {
20     private Bundle extras;
21     private GroupScreen.ResultListener[] notifiers;
22
23     public TabsPagerAdapter(FragmentManager fragmentManager, Bundle extras, GroupScreen
24         .ResultListener[] notifiers) {
25         super(fragmentManager);
26
27         this.extras = extras;
28         this.notifiers = notifiers;
29     }
30
31     @Override
32     public Fragment getItem(int index) {
33         switch(index) {
34             case 0:
35                 Fragment calendarScreen = new CalendarScreen();
36
37                 calendarScreen.setArguments(extras);
38                 notifiers[index] = (GroupScreen.ResultListener) calendarScreen;
39
40                 return calendarScreen;
41             case 1:
42                 Fragment messagingScreen = new MessagingScreen();
43
44                 messagingScreen.setArguments(extras);
45                 notifiers[index] = (GroupScreen.ResultListener) messagingScreen;
46
47                 return messagingScreen;
48             case 2:
49                 Fragment toDoListScreen = new ToDoListScreen();
50
51                 toDoListScreen.setArguments(extras);
52                 notifiers[index] = (GroupScreen.ResultListener) toDoListScreen;
53
54                 return toDoListScreen;
55         }
56
57         return null;
58     }
59
60     @Override
61     public int getCount() {
62         return 3;
63     }
64 }

```

Here is a screenshot of the add member dialog for adding a member:



Messaging

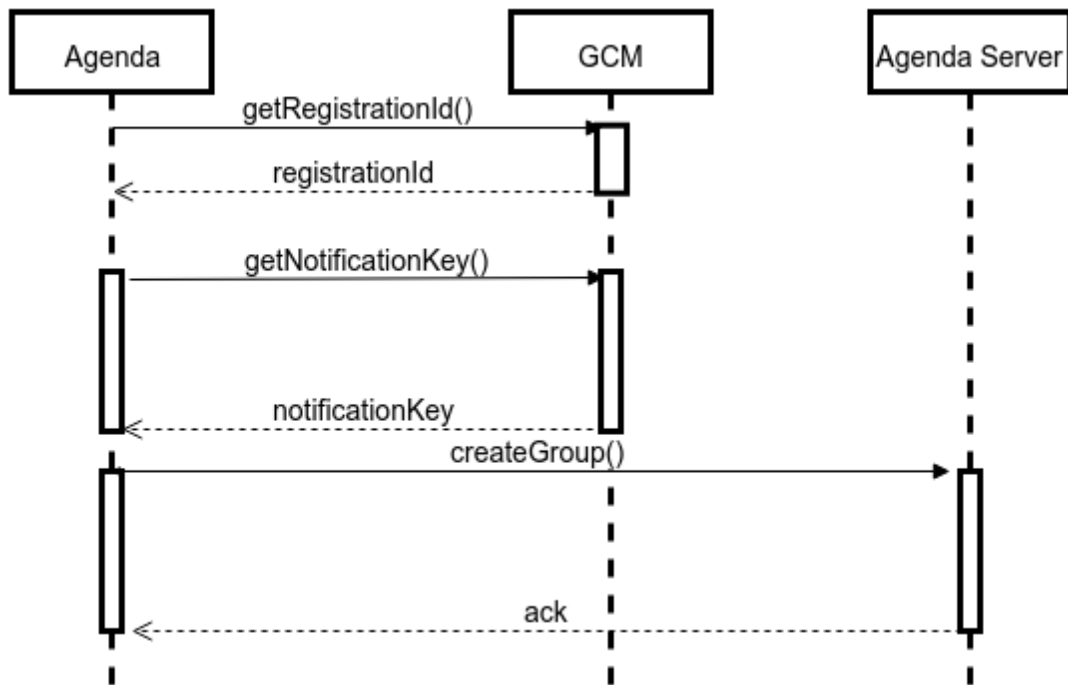
This package handles everything to do with the messaging. The messaging system is implemented using GCM. GCM is a google API for messaging device groups registered under certain notification keys that users must be registered with.

When the client launches the application there is an IntentService that triggers asynchronously to get a registration id from GCM. This key is used to register for groups. When a group is created, a notification key name is generated. The notification key name is used by GCM to generate a notification key which is used for broadcasting messages. The notification key name is the name of the group concatenated with a 128-bit UUID, ensuring that there is little to no chance of generating the same key name twice. This key name is then used to create the group sending a request to GCM. GCM will then return a notification key which is then registered with the group and the user's registration id is added to that group.

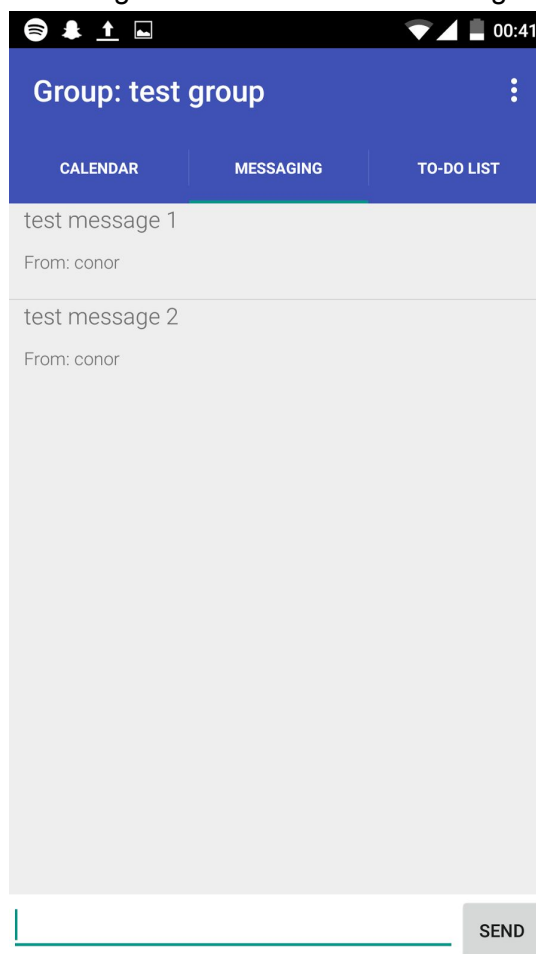
Therefore when a user sends a message, the message is packaged with their name and the message and sent with the notification key for that group. GCM then broadcasts that message to everyone else in that group.

There is a wakeful service set up to listen for messages and when received, sends the message on to the message screen fragment for displaying the message.

Here's a sequence diagram that shows the sequence of events when creating a group:



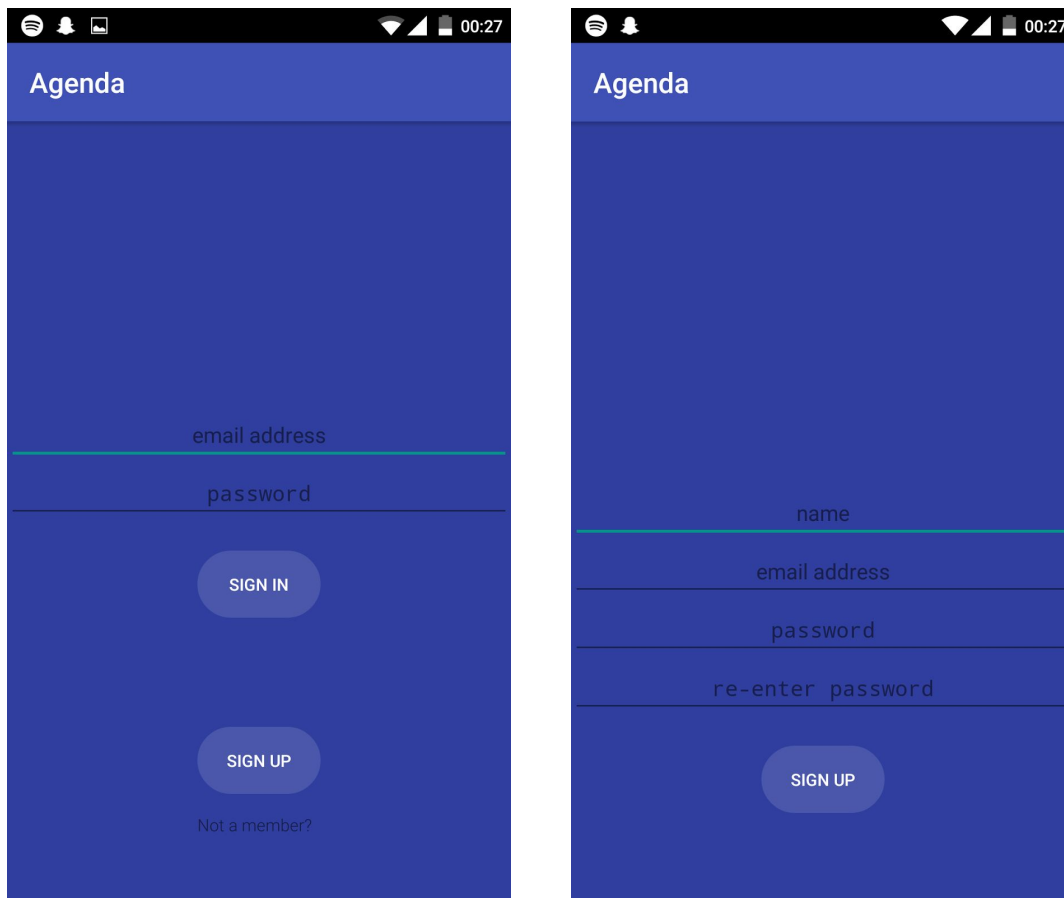
Here's a screenshot of the messages screen with some messages:



Registration

This package holds the registration and login functionality. Here users can sign up, which generates a random 128-bit salt and hashes the password before registering with the server. The user has to enter a name, email address and password.

After signing up the users can sign in with the email address and password they used to sign up. This will authenticate with the server and they will be given access to the application. Here's screenshots of the signup and signin screens:

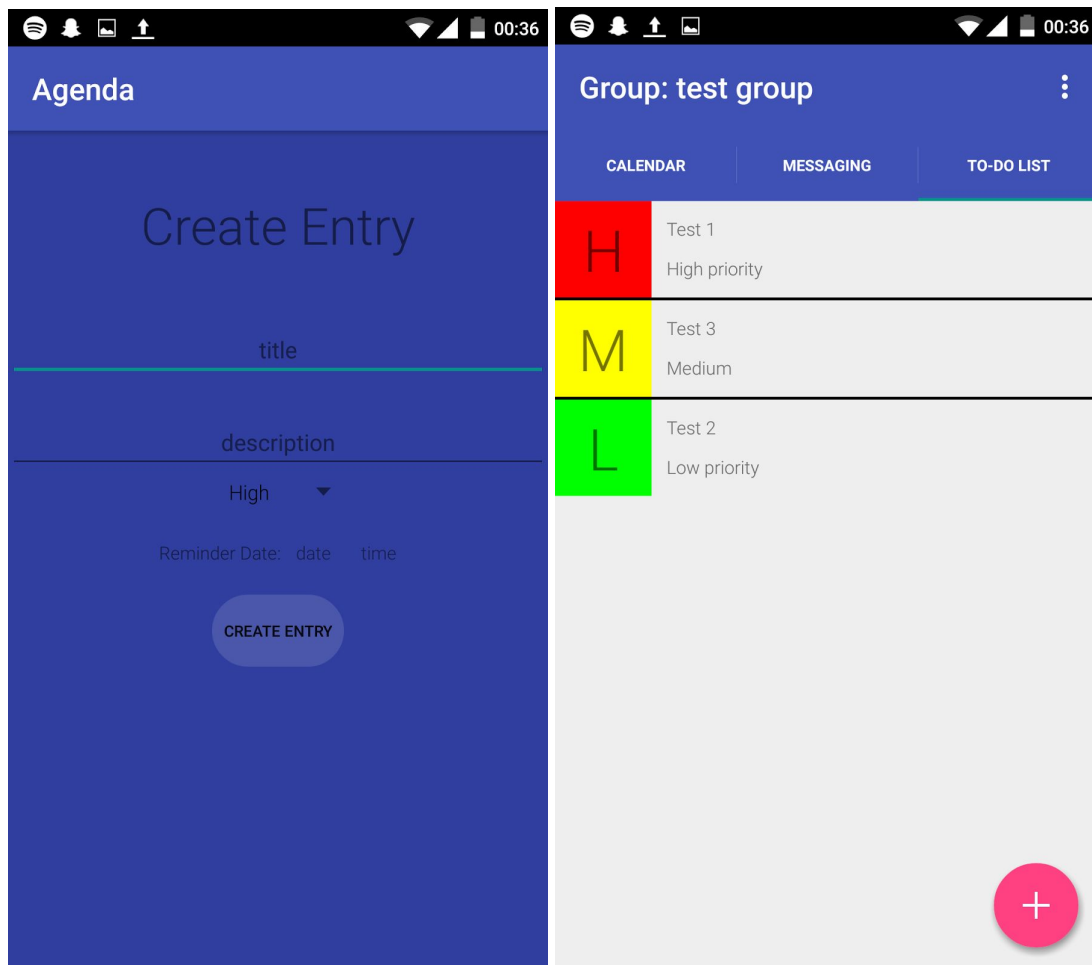


Todolist

This package holds the screens and functionality related to the to do list. The to do list is the third tab in the tabbed layout. The user will be provided a list of the to do list entries for their group sorted by most recent and highest priority. The user can create a to do list entry by clicking on the create entry button and they will be navigated to another screen to enter the details of the to do list entry.

The priority is displayed as a letter and an appropriate background colour. So for high priority it would be a H and a red background, medium priority would be M and a yellow background and low would be an L with a yellow background.

Here's an example screenshot of the to do list screen and the create entry screen:



Service

The service package holds all the functionality for making calls the API. I implemented a generic asynchronous task called `HttpRequestTask`, which can be called on for almost any call to the API. This was a necessity for avoiding code repetition and helped add global error handling with the `HttpRequestErrorHandler`. This was important to stop the application crashing on valid response codes from the server such as 404. This might not always be an error that throws an exception so these had to be dealt with. Here's an example of some code from each class:

```

132
133 private ResponseEntity<Object> doGet(String url) {
134     response = restTemplate.exchange(url, method, null, Object.class);
135
136     return response;
137 }
138
139 private ResponseEntity<Object> doPost(String url, Object... obj) {
140     HttpHeaders headers = new HttpHeaders();
141     headers.setContentType(APPLICATION_JSON);
142     headers.set("Accept", "application/json");
143
144     HttpEntity<Object> entity = new HttpEntity<>(obj[0], headers);
145
146     Log.i(Tag.TAG, entity.toString());
147
148     response = restTemplate.exchange(url, method, entity, Object.class);
149
150     return response;
151 }
152 }

```

```

20
21 /**
22  * Error handler to stop app crashing on results
23  * @author Conor Smyth <cnrsmyth@gmail.com>
24  * @version 0.1
25  * @since 2016-03-17
26  */
27 public class HttpRequestErrorHandler implements ResponseErrorHandler {
28     private Context applicationContext;
29
30     public HttpRequestErrorHandler(Context applicationContext) {
31         this.applicationContext = applicationContext;
32     }
33
34     @Override
35     public boolean hasError(ClientHttpResponse response) throws IOException {
36         HttpStatus status = response.getStatusCode();
37
38         return isError(status);
39     }
40
41     @Override
42     public void handleError(ClientHttpResponse response) throws IOException {
43         Toast.makeText(applicationContext, "Network Error", LENGTH_SHORT).show();
44     }
45
46     private boolean isError(HttpStatus status) {
47         List<HttpStatus> statuses = ServiceConstants.getHttpStatuses();
48
49         return !statuses.contains(status);
50     }
51 }

```


The `HttpRequestTask` has an interface defined in it called `ResultListener`. This result listener is implemented by classes that call `HttpRequestTask`. When the request completes the result listener calls the `onResultReceived` function which then triggers the function in the calling class that a result has been received. Then that screen can handle the request in whatever way they wish. Here's the interface and an example implementation:

```
65 public interface ResultListener {
66     public void onResultReceived(Activity activity, Object value, String url);
67 }

366 /**
367  * Result handler for request
368  * @param activity this Activity
369  * @param value the value returned from the request
370  */
371 @Override
372 @SuppressWarnings("unchecked")
373 public void onResultReceived(Activity activity, Object value, String uri) {
374     ResponseEntity<Object> response = (ResponseEntity<Object>) value;
375     swipeLayout.setRefreshing(false);
376
377     if(response != null) {
378         HttpStatus status = response.getStatusCode();
379
380         if(status.equals(HttpStatus.OK)) {
381             List<Object> grps = (List<Object>) response.getBody();
382
383             for(Object grp : grps) {
384                 Group g = ResponseCleaner.cleanGroup(grp);
385
386                 groups.add(g);
387             }
388
389             populateList(groups);
390         } else if(status.equals(HttpStatus.CREATED)) {
391             Toast.makeText(getApplicationContext(), ValidationConstants.GROUP_CREATED_SUCCESS,
392                 LENGTH_SHORT).show();
393
394             getGroups();
395         } else {
396             Toast.makeText(getApplicationContext(), NET_ERR, LENGTH_SHORT).show();
397         }
398     } else {
399         Toast.makeText(getApplicationContext(), NET_ERR, Toast.LENGTH_SHORT).show();
400     }
401 }
```


Validation

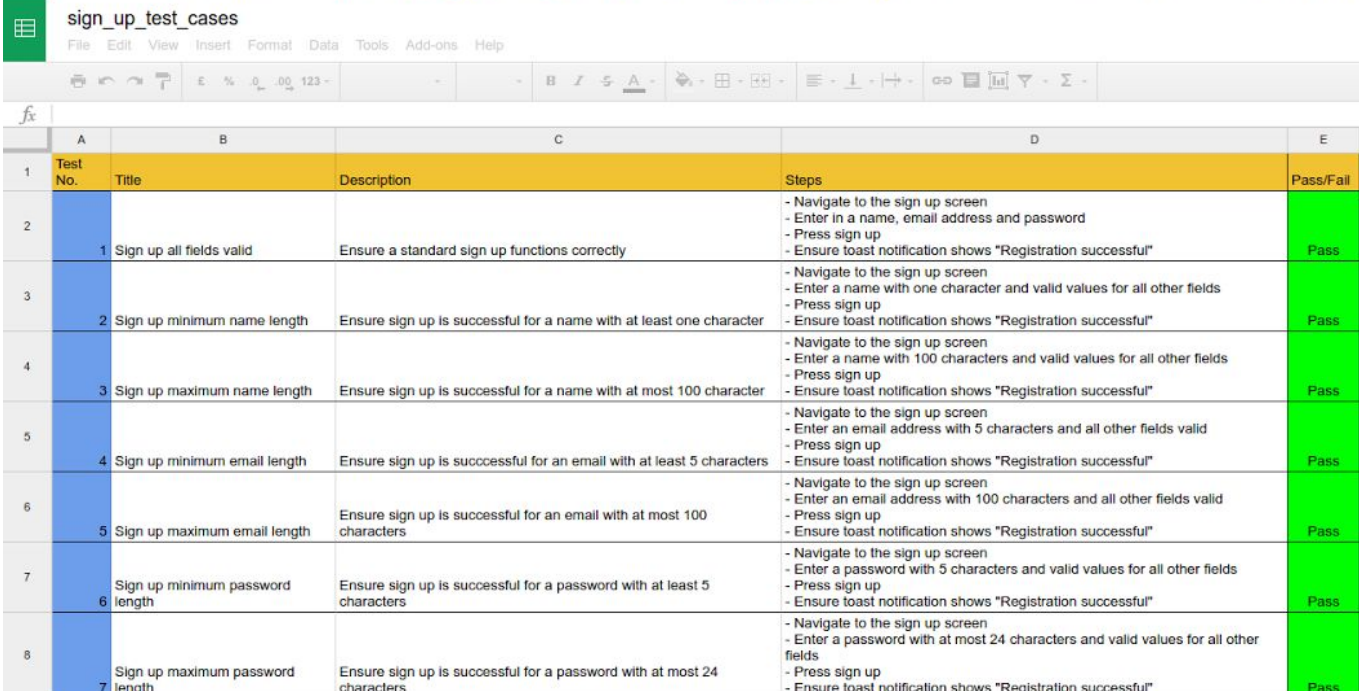
Unit Testing

Nearly all of the server functionality is unit tested. When maven builds the project, maven runs the unit tests ensuring that all the functions work as expected.

I did not implement unit tests for the user interface as this would require a lot of setup, however I created a suite of tests cases in excel for manual testing of the functionality of the system. More detail in the next section.

Functional Testing

All of the user interface functionality testing is documented in excel format. This acts as a manual testing suite for the functionality on the User interface. This suite of test cases can be easily updated and acts as a regression testing suite whenever new functionality is added. Here's a screenshot of the positive test cases for a user sign up, it covers all the edge cases ensuring everything works as it should.



sign_up_test_cases				
File Edit View Insert Format Data Tools Add-ons Help				
fx				
	A	B	C	D
1	Test No.	Title	Description	Steps
2	1	Sign up all fields valid	Ensure a standard sign up functions correctly	- Navigate to the sign up screen - Enter in a name, email address and password - Press sign up - Ensure toast notification shows "Registration successful"
3	2	Sign up minimum name length	Ensure sign up is successful for a name with at least one character	- Navigate to the sign up screen - Enter a name with one character and valid values for all other fields - Press sign up - Ensure toast notification shows "Registration successful"
4	3	Sign up maximum name length	Ensure sign up is successful for a name with at most 100 character	- Navigate to the sign up screen - Enter a name with 100 characters and valid values for all other fields - Press sign up - Ensure toast notification shows "Registration successful"
5	4	Sign up minimum email length	Ensure sign up is successful for an email with at least 5 characters	- Navigate to the sign up screen - Enter an email address with 5 characters and all other fields valid - Press sign up - Ensure toast notification shows "Registration successful"
6	5	Sign up maximum email length	Ensure sign up is successful for an email with at most 100 characters	- Navigate to the sign up screen - Enter an email address with 100 characters and all other fields valid - Press sign up - Ensure toast notification shows "Registration successful"
7	6	Sign up minimum password length	Ensure sign up is successful for a password with at least 5 characters	- Navigate to the sign up screen - Enter a password with 5 characters and valid values for all other fields - Press sign up - Ensure toast notification shows "Registration successful"
8	7	Sign up maximum password length	Ensure sign up is successful for a password with at most 24 characters	- Navigate to the sign up screen - Enter a password with at most 24 characters and valid values for all other fields - Press sign up - Ensure toast notification shows "Registration successful"
				Pass/Fail
				Pass
				Pass
				Pass
				Pass
				Pass
				Pass
				Pass

User Acceptance Testing

Throughout development I was using the application with my friends in order to try and find any bugs or take any input they had on the user experience. I told them what was implemented and what to try to see how the application would behave on other systems.

This testing was very beneficial to get a fresh set of eyes on the application and uncover some bugs. Most of the feedback was user experience related, i.e text sizes, general interaction with the user interface that would feel better for the end user.

Problems solved

User Interface Service Calls

The user interface service calls cause issue to begin with. As all web service calls should only be done off the UI thread, the suggestions from the android documentation stated creating an async task per class was the best best solution for this. This would create massive amounts of code repetition so creating the generic `HttpRequestTask` was essential. This solved the issue of repeated code and allowed for cleaner implementation of the exception on the UI from web service calls. Any 4xx or 5xx response codes from the web service caused issues that would cause the application to crash. The `HttpRequestErrorHandler` allowed me to whitelist these status codes and solve the issue of the application crashing.

Calendar View

The native android calendar view offers very little functionality. There is no capability to mark dates on it and there is little interaction capabilities. Initially I planned to create my sub implementation of this widget, however this proved to be too much work and so I resorted to just displaying the entries as a list. I still wasn't happy with this and did some research and found an implementation that did most things I wanted to do. I now combine the new calendar view with the list for the entries.

To-Do List Sorting

When initially sorting the to-do list entries I had a very verbose sort working that was very inefficient. I looked into comparators and implemented an all in one comparator that does all the sorting in one call using the java native collections sort function with my comparator solved this issue.

Stored Procedures

Initially I was using raw SQL to communicate with the database. This was very inefficient and gave rise to huge changes any time a change needed to be made. I was aware of stored procedures from work on my Internship so I looked into stored procedures for MySQL. The initial implementation was difficult and it was difficult to get the correct syntax however I worked through it and now have fully functional stored procedures.

DateTimes with MySQL

When storing and retrieving datetimes and timestamps in the database, the format for the times were getting corrupted as each were expecting different inputs. I solved this issue by implementing a type handler to define how MyBatis should behave for a certain type per row return from the database.

Server Exception Handling

Whenever an exception was thrown in the server, it caused messy results and sometimes cause the server to crash. This then gave rise to extremely unmaintainable controller coding with numerous try catch blocks. I was able to resolve this by implementing the Global Exception handler to define how the server should respond for any exception generated by the server, outside of standard Java exceptions.

Messaging with GCM

Initial integration with GCM was very difficult as their services off very little in the way of debugging issues. This resulted in a lot of time spent trying to get the actual calls to GCM working so that I could test out the messaging. I was able to resolve this but consolidating the calls to GCM into one GcmGroupService class that handles the services consistently.

Results

The resulting Application has the following functionality:

- Registration: allow users to sign up
- Log in: allow users to sign in
- Create Group: allows users to create a group
- Add Member to group: Allows users to add other member to their group by email
- Create & Delete Calendar Entries: allows users to create and delete calendar entries
- Create & Delete To-Do List Entries: allows users to create and delete to-do list entries
- Messaging: allows users to message other users in the group

Future work

There are a number of issues that arose through functional and user testing. My plan for future work is to fix these issues along with some user experience updates.

The issues that arose revolved around DateTimes from the android app to the server. The server rejects the datetimes from the android application and I was not able to fix this issue. I felt that working on implementing other functionality was beneficial. As a result to do list entries and calendar entries cannot be made for any date other than the day they are created. The server side for these functions works as the unit tests are passing however the android application cannot make the request.

Other issues that came up were issues with the messaging. Currently the messaging is very basic on doesn't store any messages on my server side and so the messages are lost upon closing the group.

There are other bits of functionality that were not as high priority and hence fell out scope for this project. The server implementation for this functionality exists but the client application does not avail of these services:

- Change group name
- Remove member from group

Any other future work would revolve around user experience improvement to make the UI more pleasing to use.

References

Spring: <https://spring.io/>

MyBatis: <http://www.mybatis.org/mybatis-3/>

MySQL: <https://www.mysql.com/>

Maven: <https://maven.apache.org/>

Tomcat: <http://tomcat.apache.org/>

Android Docs: <https://developer.android.com/index.html>

External calendar view: <https://github.com/marcohc/RobotoCalendarView>

GCM: <https://developers.google.com/cloud-messaging/>