|  |
| --- |
| **Mushroom++ mpp** |

**IoT group: Daria Maria Popa (293087), Natali Munk-Jakobsen (293132) and Mihai Anghelus (293169)**

**Data group: Levente Nagy (293115), Audrius Sauciunas (293156), Samantha Nettesheim (293089), Shaoyu Liu (294404)**

**Android group: Kristóf Lénárd (293110), Bogdan Mezei (293137), Uldis Alksnis (293168)**

**Supervisors: Erland Ketil Larsen, Astrid Hanghøj and Kasper Knop Rasmussen**

**VIA University College**

**[Number of characters]**

**Software Technology Engineering**

**4th Semester**

**4th June 2021**

*This page intentionally left blank*

**Table of content**

[Abstract iii](#_Toc18659739)

[1 Introduction 1](#_Toc18659740)

[2 Analysis 2](#_Toc18659741)

[2.1 Requirements 2](#_Toc18659742)

[2.2 Functional Requirements 2](#_Toc18659743)

[2.3 Non-Functional Requirements 3](#_Toc18659744)

[3 Design 4](#_Toc18659745)

[4 Implementation 5](#_Toc18659746)

[5 Test 6](#_Toc18659747)

[5.1 Test Specifications 6](#_Toc18659748)

[6 Results and Discussion 7](#_Toc18659749)

[7 Conclusions 8](#_Toc18659750)

[8 Project future 9](#_Toc18659751)

[9 Sources of information 10](#_Toc18659752)

[10 Appendices 1](#_Toc18659753)

**List of figures and tables**

Optional

# Abstract

*An abstract is a shortened version of the report and should contain all information necessary for the reader to determine:*

1. *What are the aim and objectives of the project*
2. *What are the main technical choices*
3. *What are the results*

*Frequently, readers of a report will only read the abstract, choosing to read at length those reports that are most interesting to them. For this reason, and because abstracts are frequently made available to engineers by various computer abstracting services, this section should be written carefully and succinctly to have the greatest impact in as few words as possible.*

*Although it appears as the first section in a paper, most report writers write the abstract section last.*

Cf. (Dawson 2009, p.195).

# Introduction

The purpose of the introduction is to provide background information and set the scene for your project. Within which business or organization are you doing the project? Who are the stakeholders and who is the customer?

The background information is adapted from your project description where you have already described the problem domain. Describe the current situation and existing context. Your statements must be supported by references to reliable and relevant sources.

This should lead to why this project is relevant and outline your aim and objectives. Which technical problems and challenges will be presented in this report, again taken from your project description. System illustrations and rich pictures are welcome here.

State delimitations relevant for your project in the introduction. Delimitations include what the project will not cover in relation to your project description, i.e. what could have been expected in your project. Remember that you can only make delimitations to aspects mentioned in the project description and you must argue well for your delimitations.

The last sentences of the introduction should be an overview of the sections to follow. This will be a good transition to the next sections.

Remember: You must ensure a clear connection between sections in the project report, from Project Description, Analysis, Design, Implementation to Test. This means that everything that is implemented can be found in design, everything that is designed is based on the analysis, and anything that is found in analysis has a clear link to requirements, etc.

# Analysis

The purpose of the analysis section is to outline an understanding of the problem domain and specifically WHAT the stakeholders want. Here, you elaborate on your background description.

You identify objects in the problem domain that will be involved in the solution and how these objects cooperate. The result of this analysis is a Domain Model (Larman 2004, chap.9) and other relevant diagrams.

Use the UML standard for all diagrams where relevant.

Note: Remember that all implementation dependent objects are not part of the domain model only conceptual classes related to the requirements and the domain.

## Requirements

The purpose of the requirement section is to define functional and non-functional requirements. Requirements are perceived as a contract with the stakeholders (customer), and are specified to ensure a common understanding.

Identify the users and describe their roles (e.g. actor descriptions, personas and scenarios).

Note: Remember that all requirements must be precise and testable.

Use the SMART principle (YourCoach n.d.) and MoSCoW (Business Analyst Learnings 2013).

Present a numbered and prioritised list of all the requirements of the users, customer and stakeholders for the project.

## Functional Requirements

Functional requirements could be described with Use Cases, Use Case descriptions and Actor descriptions. Use Case descriptions can be detailed with different types of UML diagrams.

## Non-Functional Requirements

The Android group had four non-functional requirements.

First among them was that this part of the system had to be developed using the official Android framework. Platform level 8 (Oreo, API level 26) was used on a Java 1.8 core, which according to official statistics (Android Developers, *Android Studio - Android Platform/API Version Distribution*), corresponds to approximately 60.8% of all Android devices. This was also the lowest API level that contained all functionality that the group wanted to use; therefore, it was picked as target.

The second requirement was that the system must be developed using Java. This was due to the fact that while Kotlin is available as a development language for Android, the team was untrained in its use, which would have further complicated the development.

The third requirement was to follow, whenever possible, the Google Material Design guidelines. This was to ensure that the system is more accessible to users regardless of their locale and abilities.

The fourth requirement was to structure the system using an architectural pattern. This will be detailed in the following section.

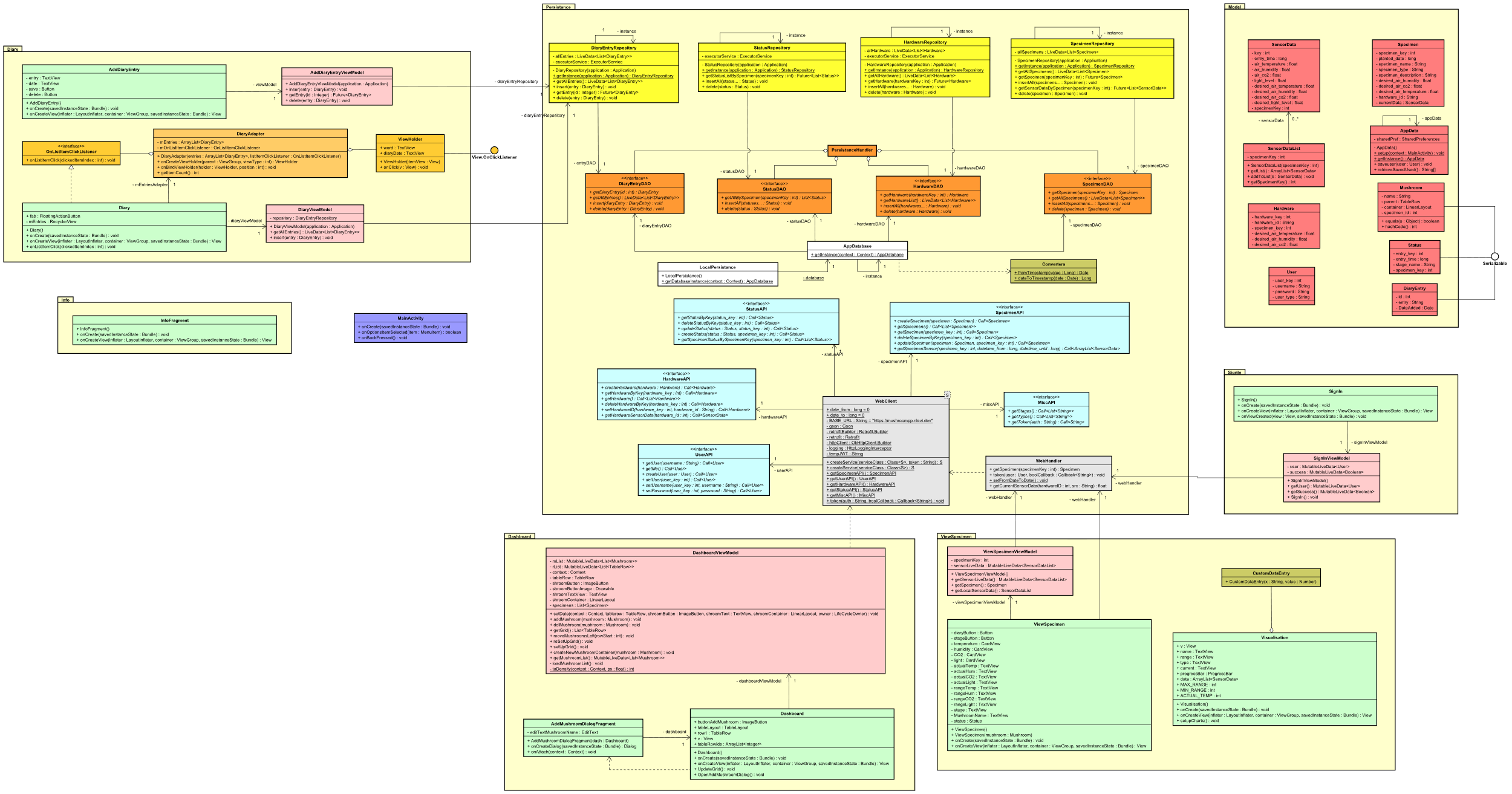
# Design

## Android team

Bogdan Mezei, Uldis Alksnis and Kristóf Lénárd

### General design and architecture.

The Android client application (used as “app” or client hereafter) was designed for only one purpose: to ensure that the user has easy access to those workings of the system that are necessary for them. Multiple architectural patterns were evaluated, however, the officially recommended architectural pattern (Android Developers, *Guide to app architecture*) is MVVM, therefore, this was used. This architecture ensured consistent user experience and appropriate software structure. The following class diagram shows the implementation of this architecture.



### Specific designs.

Within the MVVM architecture, multiple design patterns were used, as seen from the class diagram. These were the Mediator, Observer, Adapter, Singleton and Proxy patterns. Also used as already existing implementations were the Builder and Thread Pool patterns.

The Mediator pattern was used in an architecturally significant location, as the WebHandler class. This class is responsible for the mediation of data between the WebClient class, which directly interacts with the Web API, and all classes that need significant changes in their data before connection. An example of this is the token generator, where the JSON Web Token is requested from the server. This requires a specific authorization; therefore, the Mediator pattern can be used to ensure that only properly formatted data is forwarded.

The Observer pattern was perhaps the most used in the system. This derives from the LiveData class of the Android framework, which provides an observable dataset in the viewmodel for consumption by the view. As MVVM was used, this was a vital component of the design of the views, since this pattern was used to ensure constant updates based on the changes in the data.

The Adapter class was used by the RecyclerView objects, as a mandatory part of the implementation of such objects. These objects require an adapter to be able to properly execute data binding, therefore, a custom implementation was used.

The Singleton pattern was also used in multiple locations, mostly with persistence classes such as the WebClient, which directly consumes the Web API, and the LocalPersistence, which implements the Room library, ensuring the local database’s availability. These classes would normally not be duplicated, since re-generation would not only consume needless resources from the client device, but also from the API (in case of the WebClient), and might even cause data loss, therefore, the singleton pattern was the most proper to use.

The Proxy pattern was also used by the persistence handlers. These classes and interfaces were responsible for substituting commands to both the Web API and the SQLite database exposed by Room, ensuring that the other parts of the system would connect to a compatible placeholder for these external resources.

Other parts of the system mandated the usage of certain patterns as built into used frameworks and libraries. One such pattern was the builder pattern, used chiefly indirectly in the creation of the Retrofit client. This client can only be created by a builder exposed by the Retrofit object; therefore, it could not be avoided.

Another such pattern was the Thread Pool pattern as implemented by the Android operating system. The operating system has a special thread pool such as generating a UI thread, background (also called worker) threads, and possibly others. This is built into the Android system; therefore, it also was used.

### UI design

During UI design, two main goals were defined. First, simplicity, to ensure that no detailed user guides are required to be able to use the system to the fullest. Second, accessibility, to ensure that users with certain disabilities (such as being sight-impaired) are also able to use the system without additional difficulties. Besides these goals, the main objective of course was to deliver a high-quality user experience in such a format.

UI designer Bogdan Mezei has been the principal force behind the initial creation of designs, with all three team members implementing these. These designs were simple yet effective, covering all functionalities that were to be implemented in the release version. After verification and UX testing of these designs, the following designs were adopted:



These were used as a base when implementing the UIs in Android Studio.

# Implementation

## Android team

Kristóf Lénárd

During the implementation of the designs for the Android client app, portability and user experience were prioritised to deliver a client application that is not only adaptable, but also easily usable. To achieve this, the implementation followed the recommended approaches by the Android developers.

Certain non-core libraries were used to facilitate development. These were the Retrofit library for Web communication, the Room library for local persistence, and the Anychart library for visualizations within the user interface.

The implementation of the code began with the development of core classes and interfaces. These were the MainActivity, which is the wrapper class for the UI, the web client connection, to ensure that the system is compatible with the Web API, and the model classes.

On opening the application, besides the mandatory preliminary setup (which is performed by the Android framework), the first method to run is the MainActivity’s onCreate method. This sets up two core parts of the system: a part of the local persistence (the part responsible for storing user data locally), and the intrasystem navigation.

@Override  
protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.*activity\_main*);  
  
 AppData.*setup*(this);  
  
 //Bottom Navigation Bar  
 bottomNavigationView = (BottomNavigationView) findViewById(R.id.*bottomNavigationView*);  
 bottomNavigationView.setVisibility(View.*INVISIBLE*); //To not see navigation bar on Sign In menu  
  
 AppBarConfiguration appBarConfiguration = new AppBarConfiguration.Builder(R.id.*item\_home*, R.id.*dashboard*, R.id.*item\_settings*).build();  
 NavController navController = Navigation.*findNavController*(this, R.id.*fragmentbox*);  
 NavigationUI.*setupActionBarWithNavController*(this, navController, appBarConfiguration);  
 NavigationUI.*setupWithNavController*(bottomNavigationView, navController);  
 getSupportActionBar().setDisplayHomeAsUpEnabled(true);  
 }

After this, the SignIn fragment is loaded. Fragments were used in the implementation to facilitate modularity, as recommended. This fragment is used for the sole purpose of ensuring security. This is done by only enabling system access if the system was able to authenticate the user.

button.setOnClickListener(v -> {  
 signInViewModel.getUser().getValue().setUsername(((EditText) view.findViewById(R.id.*editTextTextPersonName*)).getText().toString());  
 signInViewModel.getUser().getValue().setPassword(((EditText) view.findViewById(R.id.*editTextTextPersonName2*)).getText().toString());  
 signInViewModel.SignIn();  
 final Observer<Boolean> observer = aBoolean -> {  
 if(aBoolean) {  
 System.*out*.println("Observer");  
 LocalPersistence.*getDatabaseInstance*(getActivity().getApplicationContext());  
 BottomNavigationView bottomNavigationView = view.getRootView().findViewById(R.id.*bottomNavigationView*);  
 bottomNavigationView.setVisibility(View.*VISIBLE*); //Turns on Navigation view  
 NavController nav = Navigation.*findNavController*(view);  
 nav.navigate(R.id.*action\_signIn\_to\_dashboard*);  
 }  
 };  
 signInViewModel.getSuccess().observe(getViewLifecycleOwner(), observer);  
});

As seen in the above code, a Boolean variable stored in the viewmodel of this fragment is observed, and if the value of this variable (which is preset to false), is changed to true by the authentication method, the system navigation and access to the Room database are enabled. The application is then navigated to the dashboard, which is the main layout of the app.

public void addMushroom(Mushroom mushroom) {  
 LinearLayout newShroomContainer = createNewMushroomContainer(mushroom);  
 Log.*i*("INFO", mushroom.getName());  
 Log.*i*("INFO", getMushroomList().getValue().size() + "");  
 Log.*i*("INFO", "Current Grid Contains: " + getGrid().size());  
 if (getMushroomList().getValue().size() % 3 != 0) {  
 TableRow rowToAdd = getGrid().get(getGrid().size() - 1);  
 rowToAdd.addView(newShroomContainer);  
 mushroom.setParent(getGrid().get(getGrid().size() - 1));  
 mushroom.setContainer(newShroomContainer);  
 List<Mushroom> list = mList.getValue();  
 list.add(mushroom);  
 mList.setValue(list);  
 } else {  
 TableRow newRow = new TableRow(context);  
 newRow.setId(ViewCompat.*generateViewId*());  
 ViewGroup.LayoutParams tableRowLayout = tableRow.getLayoutParams();  
 newRow.setLayoutParams(tableRowLayout);  
 newRow.addView(newShroomContainer);  
 mushroom.setParent(newRow);  
 mushroom.setContainer(newShroomContainer);  
 getGrid().add(newRow);  
 getMushroomList().getValue().add(mushroom);  
 }  
}

The dashboard consists of two main unique parts: a custom table implementation and an option to add elements to this table. These elements all constitute individual mushroom specimens (with the corresponding hardware), and are retrieved from the Web API when this fragment is loaded. This is done to ensure up-to-date information on all specimens, which is a vital component of this system.

On tapping a mushroom, the navigation transacts to the main informational fragment – the ViewSpecimen. This fragment, acting as a major endpoint for individual specimens, displays sensor data and enables access to other functions relevant to any given specimen.

private void setupCharts(){  
   
 AnyChartView anyChartView1;  
 anyChartView1 = v.findViewById(R.id.*anychart\_temp\_graph*);  
   
   
 Cartesian cartesian = AnyChart.*line*();  
   
 cartesian.animation(true);  
   
 cartesian.padding(10d, 20d, 5d, 20d);  
   
 cartesian.crosshair().enabled(true);  
 cartesian.crosshair()  
 .yLabel(true)  
 // *TODO ystroke* .yStroke((Stroke) null, null, null, (String) null, (String) null);  
   
 cartesian.tooltip().positionMode(TooltipPositionMode.*POINT*);  
 ArrayList<DataEntry> seriesData = new ArrayList<>();  
   
 switch (getArguments().getString("Type"))  
 {  
 case "Temperature":  
 {  
 cartesian.title("Temperature Monitor");  
 cartesian.yAxis(0).title("Degrees C");  
 cartesian.xAxis(0).labels().padding(5d, 5d, 5d, 5d);  
   
 String pattern = "HH:mm";  
 SimpleDateFormat format = new SimpleDateFormat(pattern);  
 for (SensorData data : data)  
 {  
 Date date = new Date(data.getEntry\_time());  
 seriesData.add(new CustomDataEntry(format.format(date), data.getAir\_temperature()));  
 }  
 break;  
 }  
 Set set = Set.*instantiate*();  
 set.data(seriesData);  
 Mapping series1Mapping = set.mapAs("{ x: 'x', value: 'value' }");  
   
 Line series1 = cartesian.line(series1Mapping);  
 series1.name("Mushroom");  
 series1.hovered().markers().enabled(true);  
 series1.hovered().markers()  
 .type(MarkerType.*CIRCLE*)  
 .size(4d);  
 series1.tooltip()  
 .position("right")  
 .anchor(Anchor.*LEFT\_CENTER*)  
 .offsetX(5d)  
 .offsetY(5d);  
 switch (getArguments().getString("Type"))  
 {  
 case "Temperature":  
 {  
 series1.stroke("green");  
 break;  
 }  
 }  
 cartesian.legend().enabled(true);  
 cartesian.legend().fontSize(13d);  
 cartesian.legend().padding(0d, 0d, 10d, 0d);  
   
 anyChartView1.setChart(cartesian);  
 }  
}

One of the main achievements in the system is the visualizer, accessed by tapping a datapoint in the ViewSpecimen fragment for that specimen. This fragment displays current and historic data for this given hardware, with the addition of a graphical interface to enhance user experience. Originally, PowerBI reports were planned to be used, however, those reports cannot filter by specimen, therefore, a custom visualizer was used, derived from the Anychart library.

Besides these fragments, there is another user interface accessible from the navigation bar – the info fragment. This informational fragment is responsible for providing information on fungiculture to end users. This is a display-only fragment, with data added and updated by the system administrator.

These fragments, however, require support. There are two main classes that enable constant access to the data – the WebClient, responsible for the connections to the Web API, and the LocalPersistence, enabling access to the Room SQLite database. These both support the storage and retrieval of the data, to ensure that the most up-to-date information is displayed.

private static AppDatabase *database*;  
  
private LocalPersistence() {  
  
}  
  
public static AppDatabase getDatabaseInstance(Context context) {  
 if (*database* == null) {  
 *database* = Room.*databaseBuilder*(context, AppDatabase.class, "MushroomDatabase").build();  
 }  
 return *database*;  
}

The LocalPersistence is structured as recommended in case of a Room database. It is a Singleton class, with the database instance being retrievable, enabling a common point of connection to the local database. This is necessary to avoid the readers-writers problem, and to ensure that the database contains information that is only overwritten at appropriate changes.

public static <S> S createService(Class<S> serviceClass, final String token) {  
 *httpClient*.interceptors().clear();  
 if (token != null) {  
 *httpClient*.addInterceptor( chain -> {  
 Request original = chain.request();  
 Request.Builder builder1 = original.newBuilder()  
 .header("Authorization", "Bearer " + token);  
 Request request = builder1.build();  
 return chain.proceed(request);  
 });  
 *httpClient*.addInterceptor(*logging*);  
 *retrofitBuilder*.client(*httpClient*.build());  
 *retrofit* = *retrofitBuilder*.build();  
 }  
 return *retrofit*.create(serviceClass);  
}  
  
public static SpecimenAPI getSpecimenAPI()  
{  
 return *specimenAPI*;  
}  
public static UserAPI getUserAPI() {  
 return *userAPI*;  
}  
public static HardwareAPI getHardwareAPI() {  
 return *hardwareAPI*;  
}  
public static StatusAPI getStatusAPI() {  
 return *statusAPI*;  
}  
public static MiscAPI getMiscAPI() {  
 return *miscAPI*;  
}  
public static void token(String auth, final Callback<String> boolCallback)  
{  
 String s = "Basic " + Base64.*getEncoder*().encodeToString(auth.getBytes());  
 Call<String> tokenCall = *getMiscAPI*().getToken(s);  
 tokenCall.enqueue(new Callback<String>() {  
 @Override  
 public void onResponse(Call<String> call, Response<String> response) {  
 if(response.body() != null) {  
 *tempJWT* = response.body();  
 *specimenAPI* = *createService*(SpecimenAPI.class, *tempJWT*);  
 *userAPI* = *createService*(UserAPI.class, *tempJWT*);  
 *hardwareAPI* = *createService*(HardwareAPI.class, *tempJWT*);  
 *statusAPI* = *createService*(StatusAPI.class, *tempJWT*);  
 boolCallback.onResponse(call, response);  
 //callback chained to divert async to MutableLiveData, where we can use postValue,  
 //so we don't have problems with returning values from async  
 }  
 }  
 @Override  
 public void onFailure(Call<String> call, Throwable t) {  
   
 }  
 });  
}

The WebClient serves as the creator of the Retrofit client, and sets up the interfaces that directly connect to the Web API. These are static, to ensure a global point of access to the API. Security is provided here by a JSON Web Token, or JWT for short, as seen above (Auth0 Inc., *JSON Web Tokens*). This is an open standard (RFC 7519), standardizing a compact, URL-safe means of representing claims. Every such token has three components, separated by a ‘.’ character. The first component, also called a header, encodes (in our case, with the Base64 encoder) the encoding algorithm and the token type of JWT. The second component, the payload, encodes information on such elements as the issuer, the expiration time, and the subject of the claims. The third component is the signature, which verifies the contents of the token and protects against tampering. These tokens are attached to the header of the HTTP call automatically, as seen in the createService method.

# Test

## Android team

Uldis Alksnis, Bogdan Mezei and Kristóf Lénárd

For the Android team, three types of testing were used: white-box, gray-box and black-box testing.

During connection development (that is, the development of the code connecting the application and the API), white-box testing was used. This was done to ensure that the connection between the API (developed by the Data team) and the client application (developed by the Android team) matched, and to further verify the feasibility of a common model.

Before deployment, gray-box testing was used. There the running code was connected to a debugger, and various actions were attempted, to be able to evaluate how the current stack and heap changes, and from those, derive whether the system works as intended. Most of this testing was done to attempt to filter any issues that the system may have, especially in areas that would have been difficult or inappropriate to test otherwise.

During deployment and the final pre-release verification, black-box testing was used. This was performed by group members not belonging to the Android team, and were used to ensure that user inputs cannot interfere with system operations. Another use was to verify whether the Android team’s acceptance and functional testing was correct.

# Results and Discussion

Kristóf Lénárd

The Android sub-group was able to achieve a satisfactory result. The system was implemented, with all intended tasks finished. The tasks that remain unfinished are due to external factors, and the finished tasks are meeting the acceptance criteria set during the project planning. The outcome of the project is deemed satisfactory.

The purpose of the results and discussion section is to present the outcome and achieved results of the project.

# Conclusions

The purpose of the conclusion section is to compile the results from each section in the report. What is the conclusion? Did the project fulfil the requirements? Etc.

You can only comment on report contents, no new topics or content can be introduced in this section.

# Project future

Kristóf Lénárd

There are many things that the Android team could do. There are additional features, that were either unachievable (due to external and extenuating circumstances, such as not having enough hardware components for certain requirements), or predetermined to not be included in this release. There are, of course, other things besides this that could be improved.

For example, while the Android part of the project is infinitely scalable, there are quality improvements that could make the software run even better, or perhaps support, with some features missing, some older versions of the Android operating system. All of these, however, are extra features.

# Sources of information

**Note: Use the standard reference method: Harvard Anglia. A very good reference tool is Mendeley** (Mendeley.com 2016), **ask VIA Library if you need help.**

Android Developers, *Android Studio - Android Platform/API Version Distribution*. Available at: <https://developer.android.com/> [Accessed 2 Jun. 2021].

Anon, 2021. *Guide to app architecture* [online] Available at: <https://developer.android.com/jetpack/guide#recommended-app-arch> [Accessed 2 Jun. 2021].

Auth0 Inc., 2021. [online] JSON Web Tokens. Available at: <https://jwt.io/> [Accessed 2 Jun. 2021].

# Appendices

* Appendix A – Project Description
* Appendix B – Hardware and Gateway Installation Guide
* Appendix C1, C2 and C3 – Source code.
* Appendix D1, D2 and D3 – Source code documentation.
* Appendix E1, E2 and E3 – Diagrams with exported images.
* Appendix F – Group Contract.
* Appendix G – Process documentation.

The purpose of your appendices is to provide extra information to the expert reader. List the appendices in order of mention.

Examples of appendices

* Project Description
* User Guide
* Source code – source documentation
* Diagrams
* Data sheets
* Etc.

**Appendix A Project Description**

Insert the original Project Description here