5 System architecture

This chapter provides an overview of the system architecture. The first section presents a superficial explanation of the entire systems architecture. In section two, architectural design decisions the team made is discussed. In the last two sections, a detailed description of the android and the server-side architecture is presented.

5.1 Architecture overview

The system has a client-server architecture with an Android client, and a central server. An overview of the architectural components is given in figure 5.1. This outline shows the primary components of the final solution. The Android app communicates with the central server through the Internet. The server handles functionality for managing and storing data in a database.

![Architecture overview diagram](image)

Figure 5.1: Architecture overview
5.2 Architectural design decisions

Selecting the architecture is an important part of any software development project. To create the best possible solution, a preliminary study was performed and a plan on how the different components would interact, was created.

The team members had little to no experience with developing a server according to the chosen design. Development and design was therefore done interchangeably and iteratively in the early stages of the project. Changes in the architectural design were made because many of the requirements in the first draft were discarded due to time constraints.

The customer was initially skeptical to use a dedicated server to store data as this adds a responsibility of up-time and maintenance. The customer suggested alternative solutions, like Dropbox and Google Drive. These were incompatible with several of the functional requirements, including tips (FR3), and comparison functionality (FR5.4). In addition, this would limit further development.

To cover all the functional requirements, storing the data on a central server was necessary. This resulted in a client-server architecture.

5.3 Android app

The app is developed to run on Android versions 4.0 or greater. As of March 2014 this makes up for 79,7% of all Android devices in use [47]. Support for earlier versions would require extra work, and because the app is a proof-of-concept this was not considered a problem.

The best practice guidelines for Android [48] state that one should avoid using a complex architecture. A complex architecture will increase the code base and is unnecessary in most situations. A structured pattern with concise code guidelines is needed to keep the development process as simple, effective, and bug free as possible. This section will give a detailed description of the Android app architecture.

5.3.1 Overall structure

The Activity is the starting point of the app. It is a self contained process with the possibility to display a user interface to the user. To access the different parts of the app it was decided to use a navigation drawer. The sections accessed through the navigation drawer is called a tab. The navigation drawer and all of the tabs are implemented as Fragments. A Fragment is a representation of some behavior or an interface. It is embedded inside an Activity and can be swapped in and out of the Activity. Multiple Fragments can live in the same view. A subset of the Activity life-cycle is implemented in Fragments so it can work as a self-contained module. To read more about the Android life-cycle, see the official Android documentation. [49]
5.3.2 Performance

One of the key issues in mobile app development is keeping the user interface responsive. Android renders the user interface on the main thread. Running long operations on this thread will make the interface unresponsive. Requesting data from external sources, like the server, or Facebook, is time consuming and should not run on the main thread. Handling threading correctly is critical for the overall performance of the app.

5.3.3 Data access

Data access to the underlying database is done using ContentProviders [50]. Data is accessed with URI’s that are uniquely defined. An overview of the data layer implementation is shown in figure 5.2. The view box in the figure represent the different Fragments that use data in the app. Fetching data to the view is done through the LoaderManager [51] interface. The LoaderManager loads data from the ContentProvider, and return the result asynchronously on a callback. ContentProviders used with a LoaderManager results in a view that is automatically updated when data changes, without bothering the user interface thread.

![Figure 5.2: Overview of how data is accessed](image-url)
5.3.4 Data manipulation

Keeping the business logic in one place makes maintaining and updating the code easier. The team decided to use the pattern Model-View-Presenter (MVP). MVP is a Model-View-Controller (MVC) derivative [52]. The MVP pattern separates the models, containing the data, from the presenter, handling the data, and the view, displaying the data. This is illustrated in figure 5.3.

![Figure 5.3: An illustration of the Model-View-Presenter pattern](image.png)

5.3.5 User authentication

The system requires the user to be identified as a unique and authorized user. This authentication is done using the OAuthV2.0 protocol [53]. This needs a third party authenticator, in our case Facebook is used. Other OAuthV2.0 authentication providers can also be used, for instance Twitter or Google. The Android system has its own account architecture. Working with this architecture requires the app to register an AbstractAccountAuthenticator [54] class in the AndroidManifest [55]. When the user logs in, Facebook returns a token, stored in the AbstractAccountAuthenticator object. The token and other related user data (Facebook id) can be accessed, by the application when necessary.

5.3.6 Server communication

The app is designed with the ability to synchronize data to the server. The server, as explained in section 5.4, exposes a restful API were the app can retrieve and store data. Data synchronization is done by using Androids built in SyncAdapter. The SyncAdapter is an external, self-contained process that keeps the data updated. The SyncAdapter is registered to a specific account on the Android system and utilizes the ContentProvider to manipulate data. SyncAdapters can be configured to run when the system has available resources. It can also be forced to run on regular intervals.
5.4 Server architecture

The server is the system back-end and is responsible for permanently storing data and providing it to the user on demand. The team decided that the server should be RESTful in order to make it easy to access. The data-interchange format chosen to transfer data between the app and the server is JSON. This format was chosen for its human readability and it is easily parsed by computer software. In addition, both these properties allow for easy integration with future functionality.

5.4.1 Dropwizard

The server is implemented in Dropwizard which is a Java framework for making RESTful web services. The main class of the server is a service that provides access to the database at run-time. The main structure of how the server uses resources and the database can be found in figure 5.4.

![Figure 5.4: UML Class diagram for main server structure](image)

Resources

A Dropwizard implementation has a set of resource classes. These classes contain methods that map to a URI path. When the server receives a HTTP request it will try to match the path to the paths defined by the resource methods. Then it will call the method found or return a
"not found" error. An example resource class is included in code snippet 5.1. The method in this resource will handle HTTP GET calls to "/user/{user}/sync/usage/{timestamp}" where {user} and {timestamp} matches any text. These variables is then used inside the method to perform actions depending on user and timestamp.

```java
@Path("user/{user}/sync")
@Produces(MediaType.APPLICATION_JSON)
public class SyncResource {

    @GET
    @Path("/usage/{timestamp}")
    public List<DeviceUsage> getUpdatedUsage(
        @PathParam("timestamp") LongParam timestamp,
        @PathParam("user") LongParam userId)
    {
        List<DeviceUsage> usage = db.getUpdatedUsage(
            userId.get(), timestamp.get());
        if(usage != null && !usage.isEmpty())
            return usage;
        else
            throw new WebApplicationException(
                Response.Status.NO_CONTENT);
    }
}
```

Code 5.1: Dropwizard resource example

Database abstraction

Dropwizard uses the JDBC and JDBI libraries to communicate with the database. The JDBC library is Java’s standard interface to databases. MySQL provides a driver for JDBC in order to make communication possible. JDBI is a convenience layer on top of JDBC which makes it possible to write an interface with SQL queries and use that interface as a data access object.
5.4.2 Database

For permanent storage the server uses MySQL. The database design is illustrated in figure 5.5. This diagram represents the data entities stored in the database and the relation between them.

Figure 5.5: ER-Diagram for the database