Friedrich-Alexander University Erlangen-Nurnberg

Faculty of Engineering, Department Computer Science

IORDANIS KOSOGLOU

MASTER THESIS

# Design and Implementation of a Multi-client API for Wahlzeit

Submitted on July 11, 2016

Supervisor: Prof. Dr. Dirk Riehle, Samir Al-Hilank
Professorship of Open Source Computer Science
Department of Computer Science, Faculty of Engineering
Friedrich-Alexander University Erlangen-Nürnberg

## Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, July 11 2016

## License

Erlangen, July 11 2016

# Table of Contents

## List of Figures

# List of Tables

## List of Abbreviations

| | |
|---|---|
| **ADAP** | Advances Design And Programming |
| **API** | Application Programming Interface |
| **CORBA** | Common Object Request Broker Architecture |
| **CRUD** | Create Read Update Delete |
| **FTP** | File Transfer Protocol |
| **GAE** | Google App Engine |
| **GCE** | Google Cloud Endpoints |
| **GUI** | Graphical User Interface |
| **HATEOAS** | Hypermedia As The Engine Of Application State |
| **HTML** | Hyper Text Markup Language |
| **HTTP** | Hyper Text Transfer Protocol |
| **IDE** | Integrated Development Environment |
| **JAX-RS** | Java API for RESTful Web Services |
| **JSON** | JavaScript Object Notation |
| **MBaaS** | Mobile Backend as a Service |
| **MVC** | Model View Controller |
| **OOP** | Object-oriented Programing |
| **OS** | Operation System |
| **POJO** | Plain Old Java Object |
| **REST** | Representational State Transfer |
| **ROA** | Resource Oriented Architecture |
| **RPC** | Remote Procedure Call |
| **SDK** | Software Development Kit |
| **SGML** | Standard Generalized Markup Language |
| **SMTP** | Simple Mail Transfer Protocol |

| | |
|---|---|
| **SOA** | Software Oriented Architecture |
| **SOAP** | Simple Object Access Protocol |
| **SQL** | Structured Query Language |
| **TCP** | Transmission Control Protocol |
| **UDP** | User Datagram Protocol |
| **UML** | Unified Modeling Language |
| **URI** | Unique Resource Identifier |
| **URL** | Uniform Resource Locator |
| **UUID** | Universal Unique Identifier |
| **WSDL** | Web Service Description Language |
| **XML** | eXtensible Markup Language |

# Acknowledgments

During my studies I had the chance to meet people that helped me expand my knowledge and improve my skills. Hereby, I would like to express my gratitude to Prof. Riehle for the idea of this thesis and his commitment to all of his students. Furthermore, I would like to deeply thank Samir Al-Hilank for the time he invested in supervising and guiding me all the way through this thesis.

I would also like to thank all of my friends for their support and especially Argiris Kitis for proofreading my thesis, and Zisis Kitsikoudis for helping me design and draw diagrams. Words are powerless to express gratitude to my parents who encouraged and supported my studies all these years.

## Abstract

Wahlzeit is a photo sharing and rating web application used for teaching at FAU. This thesis defines an API to Wahlzeit as a headless service which provides all necessary information to accommodate a wide array of possible clients. To achieve this, the design decisions to be made when connecting clients to the service are analyzed and a resulting design is defined. Appropriate technology is chosen for its implementation and the existing codebase is adjusted accordingly. As a proof of concept, an Android mobile client is implemented to interact with the service.

# 1 Introduction

During the past few decades, there have been several releases of photo rating applications that gave individuals the opportunity to submit and share pictures, either for recreational or scientific purposes. Wahlzeit as such, is an open source web application that allows users to upload their pictures and rate those of other users and in this way share their opinion, socialize and have fun. As a well designed Java web application that reflects the principles and best practices of software engineering, Wahlzeit has initially been developed to support teaching of Advanced Design and Programming (ADAP) at the Professorship of Open Source Software at the Friedrich-Alexander-University of Erlangen-Nürnberg.

Wahlzeit is a web application which interface runs in all major web browsers. However, application development might have never seen so many rapid changes and trends as in the last few years. The recent boom in the mobile and computer industry, gave users the opportunity to enjoy the freedom of mobile devices and interact with web applications as if they were using their desktop. However, converting legacy or desktop applications into workable mobile apps and simultaneously maintaining a pleasant user experience is a challenging task. Mobile devices have limited internet bandwidth while their screen sizes differ. Various operating systems have different implementations and there are several protocols through which client and server may communicate with each other.

This thesis aims to extend Wahlzeit by adding a headless service that will allow interoperability with third parties. Headless in this context, means there will be no Graphical User Interface (GUI). By analyzing and discussing the existing technologies and decisions that have to be made when connecting clients to the service, the most appropriate will be pointed out and implemented, extending the existing codebase. Moreover, a mobile client will be implemented to interact with the extended version of the application.

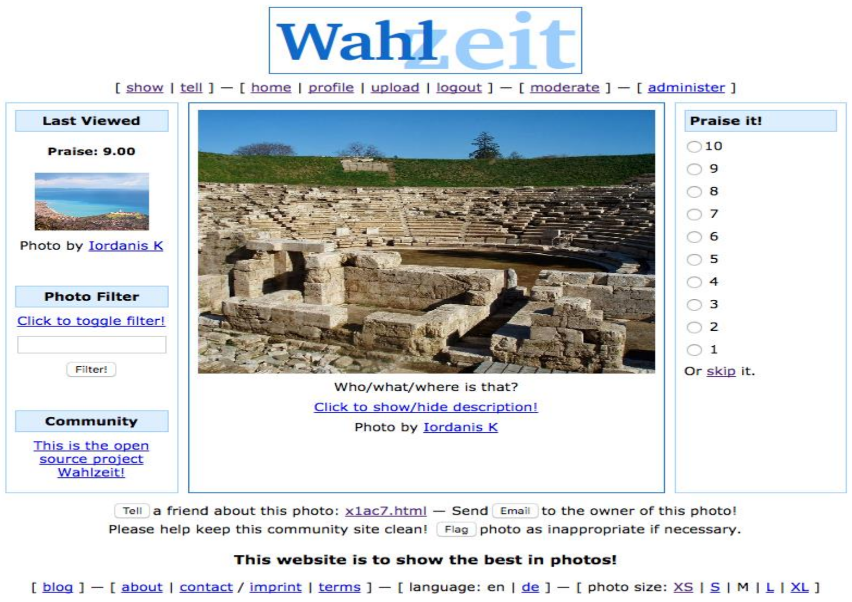
## 1.1 Wahlzeit's description

Wahlzeit is an example application used for teaching principles and practices of advanced object-oriented programming (OOP). During the ADAP lecture, students are assigned to fork the project from Github[1], and extend the software with additional functionality by recognizing and analyzing patterns and applying advanced OOP concepts. Instances of the application can be deployed either locally on their own computers using the Jetty Server or directly into the cloud.

[1] see https://github.com/dirkriehle/wahlzeit

Every instance of Wahlzeit is meant to serve particular user groups. Hence, it is organized around specific topics. For example, a region's citizens may use the application to share pictures of their city, whereas astronomers may publish photos of outer space objects (Figure 1.1). These pictures might be related to additional information such as the location, camera settings or the spectral range. This information can be associated with each picture using tags. Epigrammatically speaking, Wahlzeit allows users to upload, share and rate photos from a scale of 1 to 10. Users are able to perform Create, Read, Update and Delete (CRUD) operations and administrate their uploads and their relevant metadata. For irrelevant pictures, users can flag photos and create cases which are reviewed by the administrators. In this way, a widely accessible photo gallery could be created.

The application not only supports teaching, but also intends to fulfill its own vision. The long term vision of Wahlzeit, is becoming a product that users can rent and adjust to their needs (Hahmann, 2015). Other photo sharing applications like Flickr or Pinterest, are hard to customize and meet individual needs. On the other hand, Wahlzeit could be offered as a service, where developers could make capital out of adjusting the application according to various requirements and deploy an instance on a server. Another vision, is providing a solid codebase, including the respective infrastructure, to support people who intend to develop their own photo rating application. Wahlzeit is accessible under the GNU Affero Public License[2], which embraces contribution from the open source community, and saves time and effort to those who want to build a photo rating application.


Wahlzeit
[ show | tell ] — [ home | profile | upload | logout ] — [ moderate ] — [ administer ]
Last Viewed
Praise: 9.00
Photo by Iordanis K
Photo Filter
Click to toggle filter!
Filter!
Community
This is the open source project Wahlzeit!
Who/what/where is that?
Click to show/hide description!
Photo by Iordanis K
Praise it!
10
9
8
7
6
5
4
3
2
1
Or skip it.
Tell
Email
Flag
[ blog ] — [ about | contact / imprint | terms ] — [ language: en | de ] — [ photo size: XS | S | M | L | XL ]

Tell a friend about this photo: x1ac7.html — Send Email to the owner of this photo!
Please help keep this community site clean! Flag photo as inappropriate if necessary.

This website is to show the best in photos!

Figure 1.1: Wahlzeit's main page, displaying a random picture.

[2] see https://www.gnu.org/licenses/agpl-3.0.html

Until recently, Wahlzeit's 1.2 version was heavily dependent on numerous technologies. There have been dependencies to the development environment, whereas an SQL-based database was the underlying infrastructure to save all relevant information and pictures. The whole application stack could have been deployed locally on a Tomcat server while students have been using the department's server to make the application accessible over the internet.

Starting from 2.0, Wahlzeit has been migrated to become a Google App Engine (GAE) application, that can be deployed either locally or to GAE's cloud service without any expense. Its relational database infrastructure has been migrated into a non-relational, to comply with the standards of the GAE platform. All relevant information ranging from user information until photos and their metadata, have been migrated from PostgreSQL and the file system to Google Datastore and Google Cloud Storage respectively. Integrated development environment (IDE) dependencies have been removed and project's structure has been adjusted to conform to Gradle's default layout. As a result, Gradle building tool is used to manage dependencies and deploy the application either locally or straight into the cloud.

## 1.2 Motivation

During the past few years the mobile industry and application development have been thriving, forcing the existing applications to keep up with the upcoming trends. Many client devices have been receiving much attention, from smartphones and tablets to promising wearable and Internet of Things gadgets. Recently, fixed internet access via desktop was overtaken by mobile clients (Comscore, 2014).

In the era of smartphones and tablets, mobile applications are providing added value to several industries including transport, ecommerce, travel, retail and others. (Dalmasso et al., 2013). However, the diversity of mobile platforms and the variety of Software Development Kits (SDKs) and other tools impose unique challenges. Mobile applications in native code depend on the platform on which they are focused and the terminals where these applications work. Apple's iOS platform is based on Objective C and Swift programming languages. Android has Java and Windows Phones .NET. Those platforms are the most popular and posses the dominant market share. Additionally, there are cross platform development tools which allow developers release applications that are compatible with multiple platforms. Despite the development costs application vendors are offering mobile solutions to better reach their audience, personalize the audience's user experience and reinforce brand loyalty (Subsplash, 2015).

This fact, indicates the need to extend and enforce the existing web applications, to allow cross platform interoperability. Application Programming Interfaces (APIs) consist of a set of functions built into an application, which can be used by other applications to interact with it. To keep pace with the actual trends in application development and prepare the Wahlzeit for future development, this paper analyzes the decision to be made in order to implement an API. As a result, technological and business benefits are expected, some of which include platform independence, data integration, code reusability and cost savings (Petcu et al., 2011). Furthermore, as a proof of concept an Android client will be developed that will allow users to access the application from their mobile devices. The following diagram illustrates the extended environment of Wahlzeit, after implementing an API and an Android mobile client (see Figure 1.2).

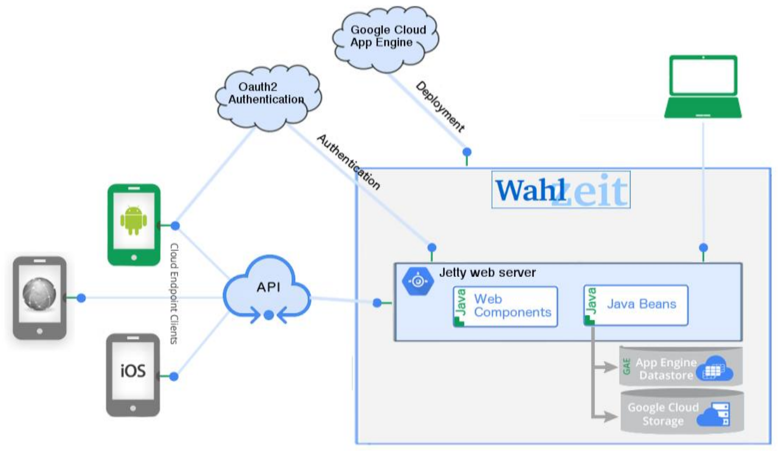


Figure 1.2: Wahlzeit's system overview

## 1.3 Thesis conventions and outline

This thesis follows formal conventions and a concrete structure. ***Important definitions*** are marked italic and bold, and for the *links* and *source code* the courier font is used. The next chapter (Chapter 2) determines the requirements, Chapter 3 gives an overview of the related literature, Chapter 4 describes the design decisions and the resulting design of the API and Chapter 5 presents the implementation effort and details for developing the API as well as the mobile client. The last section (Chapter 6), provides a critical overview and summarizes the results of the thesis and makes suggestions for future work.

# 2 Requirements

In order to achieve Wahlzeit's long term vision (see 1.1) a lot of effort is required. Except from the current browser client, the application shall accommodate mobile clients and be prepared for future changes. In this context, the following goals are set for this thesis.

**Goal 1: Analyze the design decisions to be made when connecting clients to the service.**

APIs are specified by website and have the form of web services, remote procedure calls or message passing and rely on application dependent protocols (like FTP, SNMP) (Petcu et al., 2011). Making the correct decision between the different styles depends on the size and nature of each application. Choosing the appropriate style upfront and designing the API according to its guidelines is crucial for developing an efficient and robust API for Wahlzeit.

Wahlzeit is already designed according to the Hyper Text Transfer Protocol (HTTP) protocol, and provides HTML responses to browser clients. In order to design a multi-client API and accommodate more clients, there are decisions to be made based on the interface infrastructure technique, the message representation format and the communication protocol over which the participating parties will communicate with each other.

**Goal 2: Extend the existing codebase by adding a headless service.**

After having decided which architecture style and communication technique better suit Wahlzeit's needs, the relevant guidelines will be followed to design a headless service. Based on the existing infrastructure, an appropriate framework should be chosen to implement this service. After the implementation is complete, the service should fulfill the following requirements.

1. Accommodate as many clients as possible.
2. Expose the features and functionalities of the existing desktop application.
3. Comply with the existing infrastructure and technologies.
4. Keep the interface that runs on web browsers fully functional.
5. Be intuitive so other students/developers may use it to implement further clients.

Designing the API according the best practices suggested by each style is crucial in order to facilitate future efforts and adjustments.

**Goal 3: Implement a mobile client to interact with the service**

Wahlzeit needs a mobile user interface, so that users will be able to interact with the application from their mobile devices. This mobile client should prove the usefulness and accessibility of the API. In the end, the desktop and the mobile clients should share the same information and work in parallel. The requirements that the application should fulfil are summarized below:

1. Have the same features and functionalities as the browser application
2. Enhance user experience by using mobile built in features. (Optional goal)

The mobile client will be developed as a native Android application. Since Android is a major mobile platform, the client is expected to serve a considerable amount of mobile users. Furthermore, the Android platform allows the installed applications to interact with each other and share information. Hence, built-in applications like camera and email can be accessed and used to improve user experience.

# 3 Related work

A good web API makes it easier to develop a program, by providing all the building blocks which are then put together by the programmer (Beal, 2006). This chapter gives an overview of the most contemporary interface structure techniques (3.1), data infrastructure formats (3.2) and communications protocols (3.3). Furthermore, the Android mobile OS is described (3.4).

## 3.1 Interface Structure Techniques

From an engineering point of view there are two API categories exposing operations of a certain resource (Petcu et al., 2011). In-Process APIs are what an average developer uses on a daily basis and refer to a combination of objects, methods, functions depending on each programming language, abstracting the resources in terms of memory usage, and pieces of machine executable code. This saves developers time and provides the functionality that otherwise should be developed from scratch. On the other hand, remote APIs are used to surpass a single process border and bridge applications. Web applications providers offer remote APIs for their applications, and in particular ***web services*** to interact with clients such as mobile devices. Web service are based on the request-response model where a client party invokes a service implemented by a provider party. In return, the provider processes the request and sends a response.

In general, the ***client-server architecture*** consists of the providers of a resource or service called servers, and the requesters called clients. According to the industry term, clients and servers usually have different implementation characteristics. In this context, multiple clients need to connect to a database on a central server, while the client side software mainly contains data access and presentation logic.

Web services also comply with the paradigm of ***distributed systems*** which refers to systems that are located at networked computers, whose components communicate and coordinate their actions only by passing messages (Coulouris et al., 2012). In this context, component-based applications can be deployed more easily and provide a better alternative to the costly and cumbersome two-tier architecture, by reducing concurrent usage of the database server (Abdaldhem & Fuhrer, 2009). A third layer usually puts something between the presentation and data store layer such as a business logic or service layer. This layer, moves and processes data between the two surrounding layers and coordinates the application by processing commands and making logical decisions. This approach provides several key benefits for software developers such as isolation for future changes, reduces learning curve, development tools independence, optimized security etc. (Microsoft, 2016).

Web services are self-contained, modular, distributed, dynamic applications that can be described, published, located, or invoked over the network to create products, processes, and supply chains (IBM, 2000). The core motivation to consider using a web service, includes decoupling of service interfaces from implementations and platform considerations as well as the ability to move closer to cross-language and platform interoperability (Ferris & Farrell, 2003). Some prominent examples of web service providers are for instance eBay (auction and shopping), Amazon (retail) or FedEx (logistics). The two prevailing architectures for web service provisioning are ***Service Oriented Architecture*** (SOA) and ***Resource Oriented Architecture*** (ROA).

### 3.1.1 Service oriented architecture

SOA is component-based software architecture to create distributed applications where basic services can be published, discovered and bound together to build more complex composed services (Abdaldhem & Fuhrer, 2009). A service is a collection of methods used for implementation of enterprise solutions. The service's functionality is defined by a service interface which gives the ability to the service to interact with any implementation contract (Papazoglou, 2008).

There are three major roles within the web service architecture, namely the service provider, the service requestor and the service registry whose interface should be independent from any platform, operating system and programming language (see Figure 3.1). Thus, applications become more flexible due to their ability to interact with other services. In more detail:

- Service provider: creates a web service and publishes the service description in the service registry.
- Service registry: enables online service discovery
- Service requestor: finds the service by querying the service registry. The requestor then retrieves the service description, uses it to bind to the service implementation, and begins interacting with it.

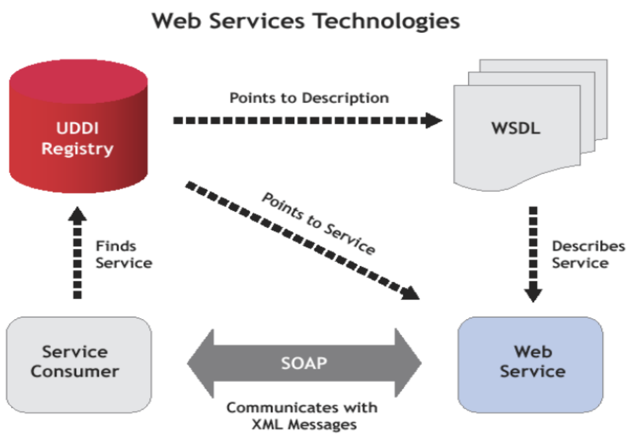


Figure 3.1: Web service technologies (Abdaldhem & Fuhrer, 2009)

As illustrated above (Figure 3.1), the Web Service Definition Language (WSDL) is an eXtensible Markup Language (XML) format that is designed to describe network services as a set of endpoints (W3C, 2001). WSDL describes those endpoints regardless of which message formats and protocols are used to communicate. However, WSDL is used most of the times in conjunction with Simple Object Access Protocol (SOAP) which internally uses XML and is used by the service to transfer messages of rigid structure (for more details about SOAP, see 3.3).

### 3.1.2 Resource oriented architecture and Representational State Transfer

Opposed to SOA, ROA premises that the whole system is made up of resources. A resource can be defined as a directly-accessible component, that is handled through a standard common interface which makes resources handling possible (Lucchi et al., 2008). A resource can also relate to other resources and provide a reference to them. In practice, a resource is similar to an object, but with a predefined interface semantic.

Based on ROA architecture, Fielding first introduced the architectural software design called ***Representational State Transfer*** (REST) as a way to organize resources and the operations to perform with them. REST is a hybrid style that derives from several of the network based architecture styles and combines additional constrains to define a uniform connector interface. This style is applied to components, connectors and data views within a distributed hypermedia system (Fielding, 2000).

RESTful web services may also be considered as an approach for using REST purely as a communication technology to build SOA. In this way, services are defined using SOA style decomposition and REST-based web services are leveraged as a transport. Henceforth, HTTP is used to make calls between machines instead of choosing more complex mechanisms such as SOAP (Abdaldhem & Fuhrer, 2009).

For an HTTP based REST application, the concept might be minimized to the following 5 core principles (Tilkov, 2011):

1. *Resources with unique identification*: Each web resource or set of resources that is relevant for the user gets a unique identifier, namely a **Unique Resource Identifier** (URI). A URI represents a global namespace and may be accessed regardless of client or session.

2. *Links and hypermedia*: The control of the application flow and the connection of resources is based on links and their description. This way the server informs the client what actions are available.

3. *Standard metho*ds: Every resource and each URI implement the same interface which refers to the standard HTTP methods. These methods or verbs, guarantee a particular behavior. As a result, the functionality of an application that conforms to the REST style is ambiguous and usable for other applications.

4. *Different representations*: Every client that requires or generates a particular data format, might interact with a resource that supports this format. The service provides different representations for each resource, whereas a client might use HTTP content negotiation to request a resource in various formats.

5. *Communication without status*: The server manages only resource states, no client specific sessions. This enhances scalability and loose coupling between the server and the client, and furthermore enables workload distribution across servers.

A system is called RESTful to the extend it conforms to the constrains implied by the principles mentioned above. In the next subsections the REST concepts will be discussed in more detail.

#### 3.1.2.1 Resource design and URIs

Resources are the central concept of REST and are uniquely identified by a URI. In the context of an RESTful API resources might be conceived as an abstract idea. A client might rather request and see the representations of resources. Those representations either comprise an internal resource that is a storable element in the database, or some instance calculated out of others (Tilkov, 2011). In this context, a resource is made out of a set of various representation whereas both are identified by a common identifier (URI).

Identifying resources and their associated representation is a hard, yet important task when designing an API. When designing an API for an existing application, the core features and software components are the main candidates in becoming resources. Those resources are oftentimes called ***primary resources*** (Tilkov 2011, p.35). Resources that are part of another resource, are usually called ***sub-resources***, and may be designed either as a separate resource entity or be embedded in the representation of another resource. Both primary and sub-resources are identified by unique URIs and will conform to the HTTP operations interface.

For almost every primary resource there are usually ***list-resources*** that allow handing resources as collections (Tilkov 2011). In many applications those collections might contain a significant amount of other resources. In this case it is quite a common requirement for the client to be able to retrieve only a subset of the available resources. To achieve this, REST suggests using the following techniques:

1. ***Filtering***: This technique allows client to apply a filter criterion to a given collection of resources. The server searches and serves only the resources that comply to the given arguments.

2. ***Pagination***: This mechanism is supported by the majority of modern database technologies and also applies to the case a client only needs a restricted amount of information. Longer list collections are divided into result sets of a limited

resource amount. The correct approach for the API would be to provide the client with mechanisms to control some aspects of a list call. In particular, the user should be able to specify the number of results per API call (similar to results per page) and to make subsequent calls to specify the particular page or location from where to return the next n results. This behavior can be implemented either by using the range header on a GET request with a resource range unit or by providing query parameters to a resource.

3. ***Projection***: The projection mechanism is meaningful when a client interacts with only a subset of the information a resource offers. In this way, the amount of information to be transferred may be significantly reduced, reducing network overhead. Projection is meaningful and may be applied both in list or individual resources.

4. ***Aggregation***. In contrast to the projection contract, aggregation offers a way to aggregate the attributes of a primary or a list resource. As a result, the required client-server interactions might be minimized.

Except from the static resources mentioned above there are also resources that allow interaction with functions and features of a web application. Those are called ***activity resources*** and enable activities to be represented as resources. An activity is a description of an action performed by the user.

#### 3.1.2.2 Standard HTTP methods

The main idea behind REST is to use the same operation over all resources. As a result, every resource should implement exactly the same interface which will reflect a set of specified operations. Although Fielding does not state which interface this should be, HTTP defines a concrete set of operations that resources should implement (Tilkov, 2011). This is the reason why REST is almost always combined with HTTP.

Those eight methods are specified since HTTP 1.1 and are the following: GET, HEAD, PUT, POST, DELETE, OPTIONS, TRACE and CONNECT. Some methods are characterized ***safe***, meaning that the implementers should be aware not to allow the requesting party to perform any action which may have an impact to themselves or others (W3C, 1999a). In the case of REST this might be roughly translated into the fact that safe HTTP methods do not modify resources, even if they might change things on a server (like for example number of total resource request) or resource but definitely not on its representation. In particular, GET and HEAD should only be used for retrieval. Safe methods also have the particularity of being ***cachable***, which means that they might be cached and pre-fetched without having any impact on the resource.

Methods can also have the property of ***idempotence*** in that the side-effects of multiple identical requests are the same as for a single request (W3C, 1999a). In the scope of the REST style, this applies to the fact that calling the same method on a given resource more than ones, should have the same impact on the representation of the resource.

However, the resource might still be affected like for instance a timestamp parameter that is not included in the representation of the resource. The methods GET, HEAD, PUT and DELETE share this property. An overview of all methods and their properties is given below (Table 3.1).

| | GET | HEAD | PUT | POST | DELETE | OPTIONS |
|---|---|---|---|---|---|---|
| Safe | ✓ | ✓ | - | - | - | ✓ |
| Idempotent | ✓ | ✓ | ✓ | - | ✓ | ✓ |
| Cachable | ✓ | ✓ | ✓ | - | ✓ | ✓ |

Table 3.1: Overview of the main HTTP methods and their properties.

The behavior of some of the most frequent operations is described below:

1. ***GET***: It is usually referred as the most important HTTP method and is used to provide a representation of the resource. It is the most frequent operation over the web, whereas the whole web is optimized around it (Tilkov 2011). Along with the URI for identifying a resource and its relevant query parameters, a client is able to send parameters over the header in order to specify the representation format or even limit the amount of the resources to be retrieved. The operation is safe, idempotent and might be cached.

2. ***PUT***: Is usually used to modify an existing resource or create one if there is none. The client uses the body entity of an HTTP request to describe the status of the resource and the header content type to inform the server about the format of the transferred information. A client expects the server to handle its request, although he may ignore or supplement information related to the resource. PUT is idempotent as multiple execution will have the same impact on a resource, although it is not safe nor cachable.

3. ***POST***: Similar to PUT it is basically used to create a new resource, or may be abused to make some action that is not supported by other methods (Tilkov 2011). The main difference between POST and PUT is reflected in the different meaning of the request URI. The URI in a POST request, identifies the resource that will handle the enclosed entity, whereas the URI in a PUT identifies the enclosed entity itself (W3C, 1999a). Usually the URI in a POST request is a list-resource where the enclosed entity should be added to. The method is neither cachable, nor idempotent or safe.

4. ***DELETE***: As expected the operation shall remove a resource. It is idempotent and is more related to some deletion logic and attributes some delete parameter to the resource representation, like for example cancelled or deleted. It is not deleting a resource from the persistence layer.

#### 3.1.2.3 Links and Hypermedia

There are a lot of mixed opinions as to whether an API consumer should create links to navigate through the resources paths, or whether links should be provided by the API. Fielding used the phrase "***hypermedia as the engine of application state***" (HATEOAS) to state that interaction with an endpoint should be defined within metadata that comes with the output representation and not based on out-of-band information.

This way the client navigates through the resources using the links provided by resource representations. At a particular time, every resource has a specific status which is expressed by the links included in its representation. In this way the whole application moves from a status to another based on hypermedia information (Tilkov, 2011, p.83). Furthermore, a client only interacts with a link and has no insight to the server's implementation and infrastructure.

To conform to the HATEOAS principles an application's API has to fulfill several prerequisites. For example, the amount of links that are delivered over a resource representation should determine the application's status and reflect the navigation possibilities of a client, so that the client should not require any external description in order to use an API, whereas the server should serve representations that the client understands (Tilkov, 2011). The advantages of abiding to those rules include amongst others: loose coupling between the client and the server, eliminating the need of using external documentation etc.

#### 3.1.2.4 Representation Formats

APIs offer different representation formats for displaying resources' representations. Some allow the client to choose between more than one formats, while other offer only a particular format. The process of requesting a particular format for a representation when there are multiple representations available is called ***Content Negotiation*** (W3C, 1999c). Applications might either use a standardized, well known format, or invent a new format and interact respectively (Tilkov 2011, p.83). More details about representations formats are discussed below (see 3.2).

#### 3.1.2.5 Stateless communication and scalability

Stateless systems can be seen as a black box, where request are treated independently and are not related to previous requests. This type of communication is called ***stateless***. However, there are mechanisms that allow tracking a client's progress and relate its requests. ***Sessions*** for example, are defined as a series of related browser requests that come from the same client during a certain time period (Oracle, 2016). This type of communication where a system has a memory is called ***stateful***.

In general, it is a better practice to avoid stateful communication, and enable each individual request to enclose all information a server needs to process it (Tilkov 2011, p

108). This approach brings the several advantages especially when the number of clients is high. The communication is more explicit and requests are easier to interpret, whereas the server requires fewer resources to process a request (Tilkov 2011, p.108).

Avoiding stateful communication according to the REST guidelines can be achieved by storing the status either on the resources or in the client application. Stateless communication is considered to have a significant impact on the API's ***scalability***. Scalability describes the capacity of a system to handle greater amounts of load (Tilkov 2011). Scalability can refer to how much additional traffic a system can handle, how easy is it to add more storage capacity or how many transactions can be processed.

## 3.2 Data infrastructure

Smartphones and tablets devices vary in terms of system platforms and application programming environments (Android, Blackberry, Ubuntu linux, iOS, Amazon's FireOS etc). Therefore, systems use specified message formats to communicate with each other. This variety has made data serialization format increasingly important.

Data serialization refers to the process of writing the state of an object to a stream and rebuilding the stream back into an object (Sumaray & Makki, 2012). The two most common modern data serialization formats are XML and JavaScript Object Notation (JSON). Although both are well established and documented, they were developed before the onset of smartphones. More recently, binary serialization formats have been gaining popularity due to the effective way they can compress data. Some popular binary data serialization formats are Google's Protocol Buffers[3] and Apache Thrift[4]. These formats were developed to address shortcomings in XML and JSON. In more detail:

- ***XML*** is a format of Standard Generalized Markup Language (SGML) that was developed by members of the W3C. It was designed to describe data, supports a wide variety of applications (W3C, 2008). XML is used to describe data and uses tags similar to HTML although they are not predefined. A document may be associated with an optional description of its grammar, which defines the structure of the document and can be used for validation. Moreover, a document has to have a correct syntax and be well-formed. It has a large user base, and is used widely in various web services. SOAP protocol for example is fully based on XML. There is also a variety of XML-based languages available.

- ***JSON*** was developed by Douglas Crockford, and is a data-interchange format that is often referred as lightweight, easily human readable and easy for machines to parse and generate (JSON, 2016). It is built on two structures,

[3] see https://developers.google.com/protocol-buffers/
[4] https://thrift.apache.org/

namely arrays (list of values) and objects (name/value pairs), and its values support most of the primitive data structures. The technology is pretty mature and many languages offer out of the box solutions to support JSON operations like parsing and encoding. However, JSON has a number of shortcomings, such as extensibility drawbacks, lack of namespace support and input validation (Nurseitov, 2008).

- ***Binary formats*** were designed to be extremely lightweight, and fast to serialize and deserialize. Messages are serialized into a compact binary-wire format. However, this format is not self describing and therefore ***positional binding*** is necessary***.*** According to this approach the name part of the name-value pairs should be kept in a separate file (Sumaray & Makki, 2012) which should be acquainted and stored in the client. For instance, .proto and .thrift files should be generated for ProtocolBuffers and Thrift respectively which are then used natively in client implementations. These files do not have to be sent over the Internet, which can greatly decrease the size of the data to be communicated.

In many applications, clients may express their preference for a certain representation format (see 3.1.2.4). According to the capacities of the corresponding API multiple formats can be supported. Different formats are suitable for different implementations, whereas each format brings its own advantages and disadvantages. Figure 3.2 provides a comparison of some major formats based on the byte size and average serialization and deserialization speeds, when transferring a text heavy book and a video object which is mainly maid of numbers (Sumaray & Makki, 2012).

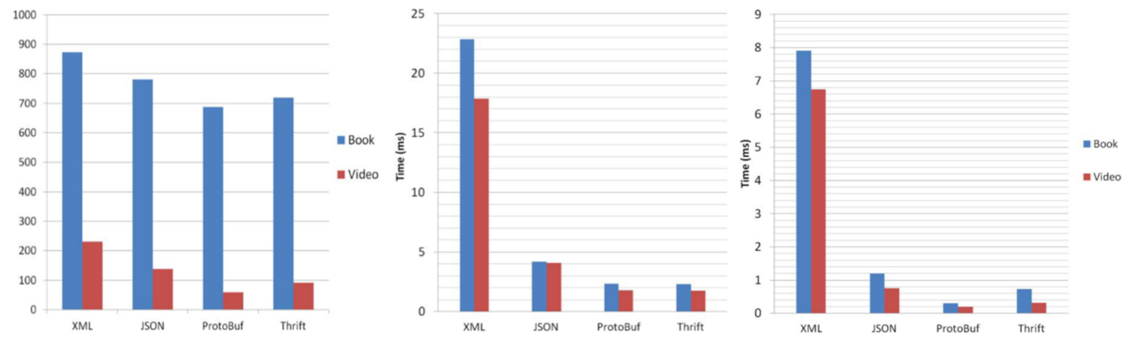


*Figure 3.2: Comparison of data serialization formats(XML, JSON, Protobuf, Thrift) based on byte size (left), average serialization (middle) and deserialization(right) time.* (Sumaray & Makki, 2012)

## 3.3 Communication protocols

In the vaguest sense, a protocol refers to a standard procedure for defining and regulating communication. Communication between components requires reliable transport, a common understanding of the data being exchanged and an understanding of the sequence of exchanges (Zahir et al., 2005). Regardless of the interface structure techniques, clients that wish to make use of a service must know the valid sequence of message exchange expected by the provider.

Since the number of web APIs increased rapidly, agents are required to work through these APIs and may need to discover conversations dynamically rather than based on prior knowledge. Reaching a consensus regarding standards in a heterogeneous environment favors interoperability between agents and might be realized by using technologies such as HTTP, SOAP, HTTP-polling, websockets etc. At this point it is important to point out that in this papers other internet protocols, such as User Datagram Protocol (UDP) are not considered, because they are designed for real-time communication like streaming and phone calls which allow outdated information to be dropped.

***HTTP*** is an application protocol for distributed, collaborative, hypermedia information systems which (IETF, 1999) uses hypertext and logical links between nodes containing text. HTTP is stateless and assumes very little about a particular system and communication between a host and a client occurs via a request/response pair. The requests messages are sent to the host via Uniform Resource Locators (URL) and are associated with verbs that define the action that should be performed (see 3.1.2.2). In the end the server responds with status codes and message payloads according to the requested representation formats. It is the protocol for which the internet is optimized and the foundation of data communication.

Another well known message transfer protocol is ***SOAP,*** which is frequently used in SOA architectures (see 3.1). SOAP is designed to enable distributed computing platforms to exchange structured information and specifies exactly how to encode an HTTP header and an XML file so that a program in one computer can call a program in another computer and pass along information (W3C, 2007). In fact most of the times SOAP uses the ***Remote Procedure Call*** (***RPC***) communication style where services are used as remote object on the client side which makes a procedural call without understanding network details and the server (Yildirim, 2014). Hence, every operation is mapped to some implementation in the backend. There are many open technologies implementing the RPC paradigm, like for example Common Object Request Broker Architecture (CORBA), XML-RPC, JSON-RPC etc. Despite its frequent pairing with HTTP, SOAP was designed to take advantage of many different types of data transport layers including synchronous HTTP/HTTPS, asynchronous queues and even over email (such as SMTP).  This capability render SOAP to be a single solution for many heterogeneous interconnectivity problems.

Web API also based communication on HTTP variations, like for example ***HTTP polling and long polling*** (Pimentel & Nickerson, 2012). According to this protocol the client sends a request to the server and the server replies either with a new message or with an empty response if there is none. After a certain amount of time, called polling interval the client polls the server again to see if any new messages are available. This is known as HTTP polling, whereas HTTP long polling avoids unnecessary requests from the client in which the server only responds with a new message or a timeout occurs. Although HTTP is a synchronous protocol where there is a response for each request, these techniques allow for some asynchronous communication in case the client is not able or eager to wait until the server responds.

A more recent protocol is the ***Websocket Protocol*** which provides full-duplex, bidirectional communication over a single socket (TCP connection). Its bi-directional and full-duplex character differentiates from the request/response paradigm and allows servers to push messages on the other side, whereas both sides communicate asynchronously. This eliminates the need of repetitive HTTP headers in each request from the client and each response from the server (IETF, 2011), and in this way reduces communication overhead.

## 3.4 Android mobile OS

In this subsection the Android OS will be introduced. However, describing Android's whole set of capabilities goes far beyond the scope of this paper.  Therefore, only the most eminent components and the ones that are used in the implementation of the client application are reviewed.

Android is a free open source mobile OS, based on the Linux kernel and designed for touchscreen mobile devices. Android was initially developed by the Android Inc, which was acquired by Google in 2005 (Elgin, 2005) and was officially introduced in 2007. Its open source character is meant to enable various developers to create mobile applications that take full advantage of all components a handset mobile device has to offer, whereas its Linux Kernel makes the OS flexible to be adjusted to any hardware platform (OHA, 2016).

### 3.4.1 Android's architecture

Android's system architecture is subdivided into five (Brähler, 2010): the kernel layer and low level tools, native libraries, the Android runtime, the framework layer and on top the applications layer (see Figure 3.3).  Android's basis is a Linux 2.6 kernel series, which bridges the hardware with the other architecture layers containing hardware drivers, and is modified for special needs in power management, memory management and the runtime environment.

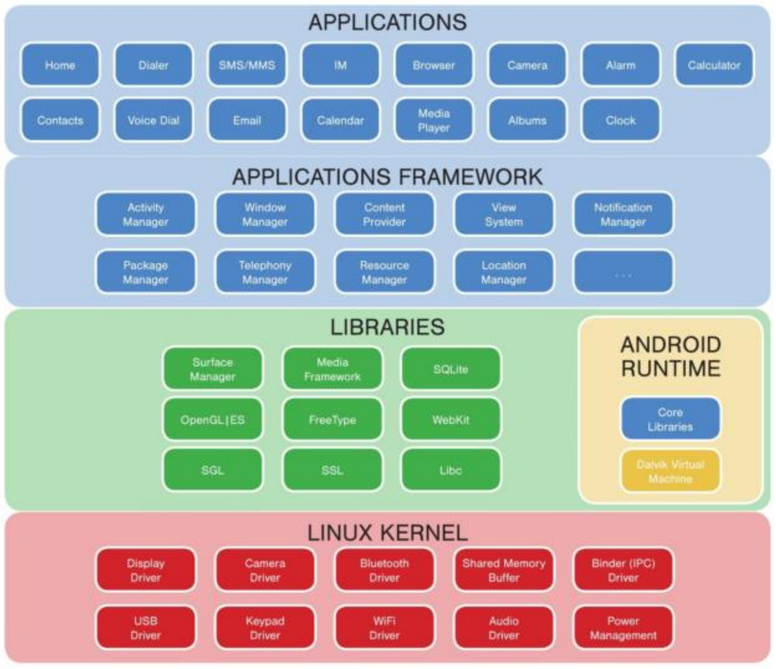


Figure 3.3: Android System Architecture (Neumann, 2012).

Right above the kernel runs a series of libraries that are compiled to device native code (C and C++) for performance reasons. Those libraries provide access to core features like the surface manager, the media framework, 3D libraries etc. The Android runtime consists of the Dalvik Virtual Machine (VM) along with the Java core libraries. Dalvik is responsible for interpreting core libraries written in Java and compiling them to native code. Furthermore, it is developed to efficiently execute virtual machines in parallel, and uses low-level memory management and the optimized threading mechanisms of the kernel.

The application framework layer is used for development and guarantees the consistent architecture of applications. The target is to force developing applications according to the given guidelines and in this way facilitate integrating and reusing applications. This layer allows execution of background tasks and information sharing between applications, and provides access to hardware modules and sensors. Of course, access to these components is requested upon installation. In this way, Android applications run in their own sandbox Dalvik VM and may consist of multiple components: Activities, Fragments, Broadcast Receivers and Content Providers.

### 3.4.2 Android components

Android provides several means of different layers to compose, execute and manage applications. This subsection gives an overview of several Application Framework layer components (see Figure 3.3), that are used to develop Wahlzeit's mobile client.

### 3.4.2.1 Android Manifest, Intents and Broadcasts

All Android applications running in the Dalvik VM environment need a ***manifest file*** to obtain administrative rights and organize the application's information (Android, 2016a). Amongst other things a manifest specifies an application's name, organizes its components and obtains permission for accessing core libraries and launching intents and broadcasts. For security reasons, every time an application is installed the user is asked to grant permission according to specifications of the manifest file.

***Intents*** in this context, refer to asynchronously sent message objects used to request an action from another app component (Android, 2016b). Intents are used to start other activities and convey information. Apart from that, intents are used to start other applications or send action broadcasts, that are received from all installed applications. Those applications provide their resources to perform the requested action whereas the user is able to choose his preferred application (Android, 2016c). This is known as broadcast service. On the other hand, broadcast receivers are used within the scope of the same application to perform some supporting operations and background tasks and get invalidated after performing the requested action.

### 3.4.2.2 Activity

An ***activity*** is as a single screen of an application that contains visual elements to present data or allow user interaction. An application consists of many related activities (Android, 2016d). Developers define the hierarchy and the navigation possibilities between the activities whereas the transition between the different activities is initiated via intents.

Activities are organized in a ***back stack*** that manages the lifecycle of all activities and is based on the "last in, first out" concept. Every newly created activity receives focus and is placed on top of the stack, pausing the activity that was previously on top. When a user finished interacting with an activity and leaves the activity using the back button, the activity is ended and removed from the stack so that the preceding activity is resumed and receives focus. For developing a flexible and organized application using the suggested callback methods is crucial. This concept is needed for handling multitasking and helps dealing with low memory situations (Android, 2016d).

### 3.4.2.3 Fragments

***Fragments*** represent modular components within an activity, which have their own lifecycle, receive their own input and can be added or removed while the activity is running (Android, 2016e). Multiple fragments can be embedded in one activity. As a result, fragments can be used to optimally fill the space of each screen and may be reused in multiple activities.

Similar to activities fragments are organized in a back stack and have their own lifecycle and status. An activity uses a ***fragment manager*** instance to add, replace, remove and perform other actions with fragments using so called transactions. Each transaction can be saved to the back stack, allowing the user to navigate through fragment changes. Fragments are usually used to adjust the UI components to various screen sizes.

#### 3.4.2.4 Tasks

Android is a multi-threading platform that allows each application to run on its own thread (Android, 2016f). Every application is decoupled from every other running application and the OS. This way, in case a crush occurs no other application is affected (known as ***sandbox principle***). Every application is running on its own main thread (also known as UI-Thread) which is accessible by all application's components.

Several tasks require more time than others, like for example fetching data from the server or from a database. To avoid blocking the main thread with tasks like these, Android suggests executing ***asynchronous tasks*** (Android, 2016g). Those tasks are triggered from the main thread and run on the background. When a task is complete, a callback method on the main thread is executed which can update the UI accordingly. This way, the UI-Thread is not overwhelmed and delays are avoided.

#### 3.4.2.5 Resources

In Android's environment resources refer to reusable data components that are used to specify and construct screen layouts (Android, 2016h). Resources are either drawables or XML files. ***Drawables*** refer to all graphical elements, from the icon that is used to start the application to search and menu icons. XML files can be either ***layout*** files that define the structure of each screen, ***values*** that are used to store information related to colors, text representations etc. or ***menu*** files that that are used to extract the options of menus within the application. All these resource files are usually part of the applications source code, and can be easily accessed from the application's Java components.

# 4 Design

The following section discusses and analyzes the design decision to be made in order to design an API for Wahlzeit that will serve mobile clients (section 4.1). Section 4.2 presents in detail the design process according to specifications extracted from the current web application and by following the guidelines of the architecture style decided upon.

## 4.1 Design decisions

There are many different **qualities** that are desirable for APIs. Some of them are related to its **usability**. For example, how easy an API is to learn and how productive are programmers using it. Others are related to its **power**, in term of expressiveness (the sorts of programs it can create), extensibility and evolution potential, performance (speed, memory and other resources consumption), security and robustness (Stylos & Myers, 2007). Although a good design does not guarantee the fulfilment of all these qualities, it is a crucial steps towards developing a good API.

There are several decisions to be made before designing a service to connect clients to an application such as the interface structure technique (see 4.1.1), the serialization format (4.1.2) and the communication protocol (4.1.3). Those decisions are discussed in the following sections and are made so that Wahlzeit's API will fulfil as many qualities as possible from the ones mentioned above.

### 4.1.1 Interface structure technique design decisions

Wahlzeit's API structure should be optimized to connect mobile clients to the application. The most common photo sharing applications over the internet (Flickr, Pinterest, Instagram), offer APIs that are based on REST or SOAP or both. It is important to clarify that SOAP is not a software architecture style like REST but a protocol. However, the term is frequently used to refer to the implementation of SOAP over HTTP in SOA architectures (see 3.1.1). This section analyzes the differences between REST and SOAP and offers an approach that is oriented to the challenges involved in building mobile applications for multiple platforms.

SOAP is a well-known technology that was designed before the explosion of mobile technology. It introduces an abstraction layer that can be build on top of any transport layer (see 3.3), which on the one hand offers extensibility and portability but on the other hand is considered over-engineering and might not provide any real value (Wagh & Thool, 2012). Hence, generating SOAP client code from WSDL interfaces can be quite complex and re-coding the complex interface several times for several platforms can be both time-consuming and error-prone. Moreover, the fact that the whole procedure is based on parsing and generating XML messages can be notably resource-consuming for mobile devices, it terms of power, bandwidth and computing power. Another problem imposed by SOAP is the fact that changing services often means complicated code change on the client side. This problem is magnified for mobile devices where the problem of application updates dissemination arises.

On the other hand, REST was designed to operate with thin clients, and is almost always build on top of HTTP. Hence, its infrastructure is considered more friendly to developers as it is similar to the majority of web traffic on the internet and since web traffic is already optimized for HTTP it is also optimized for REST. REST provides flexibility regarding the data types returned, which has impact on the resources needed to parse and generate messages. Furthermore, REST offers mechanisms for caching requests, which might reduce the number of necessary requests from the client and is designed to allow stateless communication which facilitates scalability (see 3.1.2.5).

For the reasons presented above and having in mind the qualities of a good API (see 4.1), REST is considered a more suitable technique for designing Wahlzeit's API and accommodating mobile clients. In terms of simplicity and developer productivity, REST does not add another abstraction layer and when used alongside HTTP, it uses common internet concepts like operations and status codes.

The fact that there is a possibility to choose between multiple serialization formats has an impact on overall API performance and on resource usage - clearly important to mobile devices where battery and network data usage are valuable resources. (see 4.1.2). This fact also affects the flexibility of the API. Wahlzeit's future clients might require unpredictable representation formats. Hence, an API structure that allows selecting message formats gives the opportunity to future developers to extend or adjust the representation infrastructure if necessary.

If designed properly, a RESTful API might also take advantage of caching mechanisms and stateless communication which will have impact on its extensibility and evolution potential. Last but not least, assuming that the transport will be over HTTP, the security mechanisms that are built-in the protocol will be available. Table 4.1 summarizes the qualities of a good API and the reasons REST is considered a better choice for APIs designed to accommodate mobile clients.

| API qualities | Reasons for choosing REST to accommodate mobile devices |
|---|---|
| Usability (learning curve, programmers productivity) | Does not require any additional tools and knowledge of the WSDL interfaces and is easier to use as it uses common networking concepts (URI, HTTP operations, status codes). |
| Performance (speed, power, bandwidth consumption) | Allows choosing efficient serialization formats. Provides caching mechanisms. |
| Extensibility and evolution potential | When designed correctly it provides loosely coupled client-server interaction allowing stateless and scalable communication. Allows adjusting the data infrastructure to communicate with unpredictable future clients. Changing services in web provisioning does not require changes and updates in mobile clients. |
| Robustness | Does not add an additional abstraction layer on top of the transport layer and thus makes development in both server and client side less error-prone. |
| Security | Uses HTTP Security. |

Table 4.1 API qualities and the reasons why REST is considered a better choice for APIs designed to accommodate mobile clients.

### 4.1.2 Representation format design decisions

Message serialization formats allow systems with different implementations to exchange information. Wahlzeit's API should accommodate as many clients as possible (according to Goal 2, Requirement 1), therefore choosing a well-established serialization format that is supported by the native environments of most mobile platforms is important and will facilitate the development process in the client's side. Furthermore, the API should allow transferring both text as well as binary data like for example images. As described in the previous section serialization formats have an impact on the mobile client's bandwidth, power and resource consumption and affect the overall performance of the API. Since the comparison of the data serialization formats is beyond the scope of this paper, the serialization formats will be considered according to the relevant literature (see 3.2).

Despite its widespread use, XML is considered extremely verbose and not well suited for mobile environments. Parsing and generating XML messages is not a trivial task and requires a serious amount of computing power. Furthermore, XML payloads have a larger size compared to other formats which increases bandwidth consumption.

On the other hand, JSON is considered as the most suitable format for mobile devices and RESTful web services. All mobile platforms offer tools to facilitate processing JSON messages. It requires less resources and time to be processed and its payload are significantly smaller than its XML counterparts.

Binary formats like Google's Protobufs and Thrift, are superior compared to XML and perform slightly better than JSON messages (see 3.2). As a result, binary formats would be a good choice in order to optimize performance. However, the transferred binary files have to be re-compiled from the client application to assign the correct name-value pair (see 3.2). As a result, using binary formats assumes that the mobile platform offers libraries to parsed these messages. This fact could make the development process much harder in case such libraries are not offered by the platform. Furthermore, changing the message structure, would require updating the mobile client applications according to the new name-value pairs.

For the reasons discussed above, JSON is considered to be the most suitable serialization format for Wahlzeit in order to accommodate multiple clients, facilitate development, and optimize performance. Furthermore, there are a lot lot of ways to exchange binary data with a REST API by merely using JSON messages (details are discussed in the implementation chapter, see 5.3.4). Hence, for simplicity reasons, content negotiation is not necessary and JSON can be the only serialization format which the API has to support.

### 4.1.3 Communication protocol design decisions

Wahlzeit is a web application that uses HTTP, to define the sequence of exchanges to reliably transfer text and messages with web browser clients. This section, analyzes whether it would be appropriate to use another protocol (see 3.2) for designing Wahlzeit's API to transfer messages and communicate with mobile clients.

For the same reasons discussed above (see 4.1.1) SOAP uses XML message encoding which runs on top of other communication layers. However, for Wahlzeit there is no need of using other data transport layer for transferring its context (text and images). In order to keep things simple and be flexible in choosing the messages representation formats, SOAP is not considered for designing Wahlzeit's API.

WebSockets is a protocol which addresses some shortcoming of HTTP, by reducing overhead and allowing serves push messages any time (see 3.2). This is useful for real-time data transmission. Wahlzeit however, is not a real-time application (e.g. online gaming, chat) that requires a persistent connection and receives updates every second. Furthermore, WebSocket is a stateful protocol that might affect scalability, whereas mechanisms that come along HTTP like caching, routing and compression have to be defined on top WebSockets. Hence, the overhead produced by the requests Wahlzeit's case is negligible, and using WebSockets shall not bring any notable benefits.

HTTP polling and long polling is another mechanism to address the fact that there is no mechanism for the server to send data to the client without the client asking first (see 3.2). It allows asynchronous communication similar to WebSockets, though it is not a protocol and uses known HTTP concepts. As a result, in favor of simplicity any asynchronous communication necessary in the scope of Wahlzeit's API could be implemented using these techniques.

In this context, it can be inferred that Wahlzeit's aspects do not suggest that using another protocol than HTTP is necessary. Hence, the API's design is expected to be simple since there are no additional layers added on top of others and allow flexibility in choosing message serialization formats. Last but not least, asynchronous communication requirements could be fulfilled by using HTTP mechanisms.

## 4.2 API Design

After analyzing the design decisions, this section presents the process of designing the API according to REST guidelines  (see 3.1.2) and the HTTP protocol, as it has been decided upon in the section above. The APIs message infrastructure will be discussed later, when selecting the technology that will be used to implement the API (see 5.1).

For the design process, features and aspects of the existing web application have to be taken into account. For this reason, user stories have been extracted by examining the GUI and the features of the web browser client. Table 4.2 summarizes all those user stories which are defined in the form: The API should offer a way to <do something>, so that client applications can <achieve something>.

| API US # | The API should offer a way to | So that client applications can |
|---|---|---|
| API US-1 | Create client objects with different access rights  (User, Moderator). | Allow users to securely login or navigate through the application's content as guests. |
| API US-2 | Modify client objects. | Allow users update their profile settings (name, gender, language, notification settings) |
| API US-3 | Create new photos | Allow users to upload photos |
| API US-4 | Retrieve the applications photos, either all of them or only subset. | Retrieve the application's photos. |
| API US-5 | Retrieve photos that belong to a particular client. | Allow a user to review his own uploads. |
| API US-6 | Retrieve all image sizes associated with a photo, or only images of some particular size (Medium, Large etc.) | Actually display the photos of a particular size. |
| API US-7 | Modify a client's photos | Update the tags and visibility status of a photo. |

| | | |
|---|---|---|
| API US-8 | Delete some client's photos | Allow users to delete their uploads. |
| API US-9 | Modify the praising value of photos | Allow users to actually rate photos. |
| API US-10 | Create photo cases | Allow users to flag photos. |
| API US-11 | Administrators to retrieve all photo cases | Display all photo cases to the administrator . |
| API US-12 | Administrators to modify photo cases | Allow administrators decide on the action to be performed in each photo case. |

Table 4.2: API user stories as extracted from the current web application.

### 4.2.1 Resource identification and URIs construction

In this subsection the logic of defining resources from the extracted user stories (Table 4.2) will be presented. Furthermore, identifiers (URIs) have to be assigned to each resource. The logic and naming conventions for constructing these URIs will also be discussed. It is assumed that Wahlzeit's API is accessible under the hypothetic path http://www.wahlzeit.org/restApi/.

The main concepts of the web application are the main candidates for becoming primary resources (see 3.1.2.1). Wahlzeit is a photo rating application, so ***photos*** are identified as a primary resource. Other primary resources are the ***clients*** and the ***photo cases***. Primary resources are also designed as list-resources in order to manage them as collections. For the URI of those resource plural is used, so they can be identified as collections. An individual resource can be accessed by appending its Universal Unique Identifier (UUID) (for example clientId, photoId etc.) to the end of each path. The following URIs represent the paths for accessing and performing operations to the photos' collection and an individual photo respectively.

http://www.wahlzeit.org/restApi/photos

http://www.wahlzeit.org/restApi/photos/{id}

Collection resources like photos or photo cases might contain a large amount of resources whereas client applications might need to retrieve only a subset. For instance, according to US 5 client applications may need to retrieve only the photos that belong to a particular user. For retrieving subsets of primary resource, REST offers mechanisms like filtering and pagination (see 3.1.2.1). There are many ways to construct URIs that will reflect those mechanisms. For this design, query parameters are preferred for their expressiveness and simplicity. For instance, an API user can either omit query parameters and retrieve the whole list or use query parameters to define a subset. The following URIs represent the links for accessing the photos of a particular client (specified using his clientId) and using pagination to limit the amount of retrieved photos (limit) on each request. For pagination, an indicator parameter (cursor) is required to help browsing through results.

http://www.wahlzeit.org/restApi/photos/?clientId={id}

http://www.wahlzeit.org/restApi/photos/?limit={resultsPerPage}& cursor={nextPageToken}

Resources can be also related to each other. Each photo for instance has various image sizes, according to the US 6. At this point it is important to distinguish the difference between ***photos*** and ***images*** in the scope of Wahlzeit. Images refer to the actual binary image file that is stored in various sizes. Photos refer to all metadata and information that make up a photo object (tags, praising, visibility status etc.). Hence, there is a one-to-many relationship between photos and images. Images are defined as sub-resources (see 3.1.2.1) and for this design they will be embedded in the representation of photo resources. The reason for doing this, is to allow client applications perform operations on photos (e.g. updating tags, praising) without having to transfer data related to the image. The relationship between these resources is depicted in the relevant URI.

http://www.wahlzeit.org/restApi/photos/{id}/images

It is also likely that Wahlzeit's clients might only need to work with particular image sizes, that fit to their screen's size. Rather than accessing the full set of fields offered by a resource, it is possible to be very precise as to which fields are included or excluded. For this case, REST offers the projection mechanism (as described in 3.1.2.1). For this design, resource projection will be also represented by the URI and the relevant query parameters. The following path provides access to images of the specified image size.

http://www.wahlzeit.org/restApi/photos/{id}/images?imageSize={size}

Except from static resources, there are also resources that are associated with actions. According to US 10 for instance, clients should be able to rate photos. Hence, rating is identified as an activity-resource which is not associated with a model in the application's persistence layer. However, this action is executed frequently and dedicating a resource to it makes the task more explicit. The URIs for such resources are constructed by adding a verb to describe the action. The following URI is used to represent the rating action related to an individual photo.

http://www.wahlzeit.org/restApi/photos/{id}/rating

Following a similar process, resources are identified and URIs are constructed for the other primary resources, namely the clients and photo cases. The following section describes the process of assigning operations to the defined resources.

### 4.2.2 Assigning operations to the defined resources

Unlike other approaches, REST does not define verbs or methods to perform specific functions. Nevertheless, HTTP operations define the type of actions to be performed by each request. This section, will present the process of allocating HTTP operations (as described in 3.1.2.2) to the identified resources, according to the APIs user stories (Table 4.2). The photos' primary resource as well as its sub and activity resources will be used as running examples.

According to US 4, client applications should be able to retrieve the applications photos. The /photos list-resource contains all photo resources. Hence making a GET operation to this resource should return the entire collection. By using query parameters and defining the URIs for filtering and projection, and thereafter performing a GET, client applications should retrieve only the photos that belong to a particular user (see US 5) or limit the amount of returned resources respectively. Likewise, making a GET request to sub-resources like images (/photos/{id}/images) should either retrieve all images associated with a photo, or the image sizes specified by the request parameters (US 6).

For allowing client applications to upload photos, according to US 3, making a POST request to /photos should create a new resource along with its associated images and add it to the collection (see 4.2.3, for explaining this decision). Client applications might also need to perform actions that can not be matched by any HTTP operation. Such an action for example is rating photos (US 9). For such cases, POST is used. Hence, by making a POST request in the rating activity-resource (photos/{id}/rating), a "rating task" is delegated to the API. It is also important to understand that since POST is not idempotent, performing the same request multiple times will create identical photo resources or praise a photo multiple times with the same praising value. Last but not least, the response to a POST request will also contain the created resource's representation.

According to US 7 and US 8 client application should be able to update (tags, visibility status) and delete individual photos. This can be achieved by interacting with the individual photo's URI (/photos/{id}). In this case, making a PUT request should update the relevant photo's content, whilst a DELETE request should delete the resource from the server. Both operations are idempotent, therefore trying to modify or delete the same resource multiple times will not have any side effects. The response in a PUT request, should also contain the updated resource's representation.

Following a similar approach, HTTP operations are allocated to the other identified resources. Table 4.3 gives an overview of all identified resources along with their assigned HTTP operations.

| Resource | URI | Operations | Usage |
|---|---|---|---|
| List of all clients | */clients* | POST | Login |
| Client | /clients/{clientId} | PUT | Change profile settings |
| List of all photos | /photos | GET<br>POST | Retrieve all photos<br>Upload photo |
| Filtered photo list | /photos/?clientId | GET | Retrieve client's photos |
| Paginated photo list | /photos/?limit&cursor | GET | Retrieve a subset of all photos |
| Photo | /photos/{photoId} | GET<br>DELETE<br>PUT | Retrieve a single photo<br>Delete a photo<br>Change tags and visibility status |
| List of all images | /photos/{photoId}/images | GET | Retrieve all images for a specific photo |
| Projected image list | /photos/{photoId}/images ?imageSize | GET | Retrieve images according to the specified image size |
| Praise | /photos/{photoId}/praising | POST | Rate a particulatrphoto |
| List of all photos cases | /photocases/ | POST<br>GET | Flag photos<br>Retrieve all photo cases |
| Photo case | /photocases/{photocaseId} | PUT | Decide on photo cases |

Table 4.3: Overview of Wahlzeit's resources

### 4.2.3 Achieving stateless communication and defining navigation possibilities

REST principles suggest avoiding stateful communication and enclosing all necessary information with each request, enabling the server to process them independently (see 3.1.2.5). Furthermore, the HATEOAS concept suggest that client agents should be able to determine what actions can be performed based on the current application's state (see 3.1.2.3). This section discusses how these principles shall be fulfilled by the API.

Wahlzeit 2.0 extensively uses session instances to store various information and relate the requests that come from a particular client. When a user uploads a photo for example, the session stores the tags and the file's location which are used later when processing the request, for constructing and saving the photo object. Moreover, sessions keep track of the photos a user has praised to avoid having the same photo praised multiple times by the same user. To achieve stateless communication, information about tags and praised photos will be included to the photo and the client resources respectively. This allows both the server as well as client applications to interpret and process the information enclosed with a resource, without having to associate any requests or use sessions.

To conform to the HATEOAS concept, resources have to enclose all the links that reflect the navigation possibilities. Hence, client application can perform actions based on these links. However, decisions on what requests will be sent are made when the client applications that integrate the API are written. As a result, client applications would not be able to handle significant API changes without breaking. Therefore, although HATEOAS is promising, Wahlzeit's API will not be fully implementing this concept. Nevertheless, resources will contain resource identifiers and some navigation possibilities in case standards and tooling will be defined around this concept in the future. For instance, individual a photo resource should contain a link to the resource itself (identifier) as well as links to its images and rating resources. The same applies for the client and photo case resources.

The following diagram (Figure 4.1) describes the design of Wahlzeit's RESTful API, by illustrating resources as well as the relationships and paths between them. It is a hybrid web API/information model diagram, based on Unified Modeling Language (UML) and inspired by IBM guidelines (IBM, 2011). Classes are identifiable as resources by writing <<Resource-Type>> above the class name. In order to model URIs, ***Path Dependencies*** are defined by writing <<Path>> and the relevant URI next to each arrow. Path dependencies show the direction of the relation between two resources. Each resource's URI can be inferred by following the path dependencies from the application class to the relevant resource.

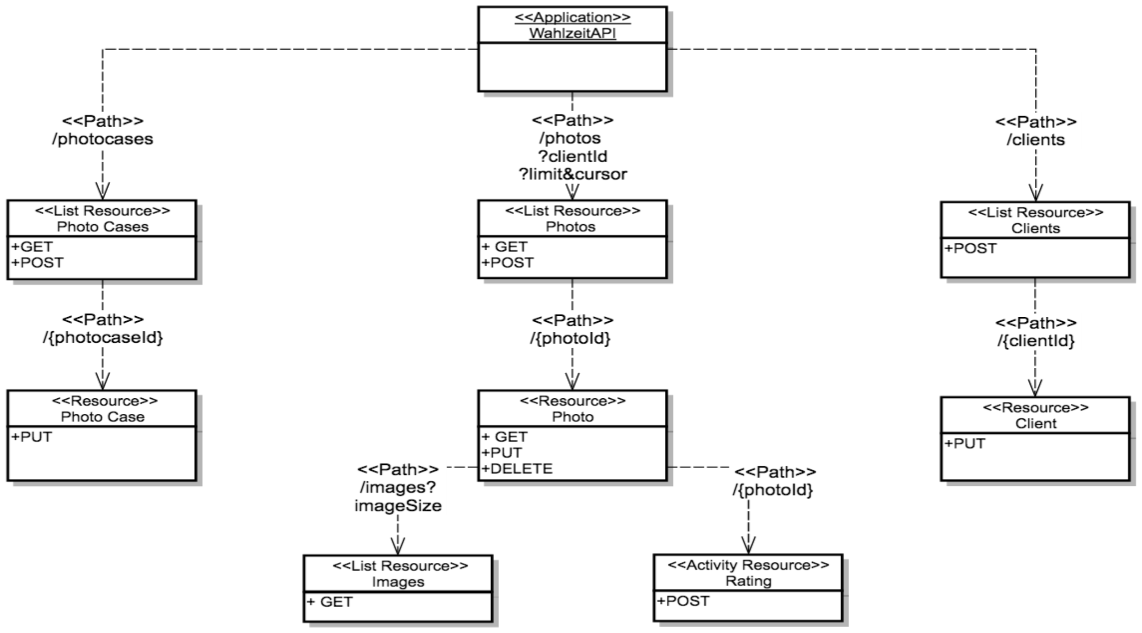


Figure 4.1: Class diagram describing resources and their relevant URIs

# 5 Implementation

Having defined a design for Wahlzeit's API, this chapter chooses a technology for its implementation (5.1). Thereafter, the chosen technology is described (5.2) and the process for extending and adapting the existing codebase to the defined design is presented (5.3). Last but not least, a client application running on Android is designed and implemented (5.4).

## 5.1 Selecting a technology to implement Wahlzeit's API

APIs can be developed either by simply using a web browser and some HTTP library or one of the numerous development frameworks available for any programming language (Bouguettaya et al., 2013). Development frameworks incorporate many libraries in a set of cooperating classes that make up a reusable design, and thus facilitate the development process significantly. This section extracts decision criteria for selecting an appropriate framework to implement Wahlzeit's API, by taking into account the design decisions made in the previous chapter (4.1) and the goals set for this thesis (Goal 2). Afterwards, frameworks that fulfill these criteria are detected and the most suitable is selected.

First of all, according to requirements 3 and 4 of the thesis' Goal 2, the implemented service has to comply with the existing infrastructure whereas the current interface should remain functional. Wahlzeit 2.0 is a Java web application that conforms to GAE's standards and is deployed on top of Google's infrastructure. Thus, the selected framework should ***support Java*** programming language and ***be compatible with GAE's*** infrastructure. This is defined as the first selection criterion.

Secondly, according to the decision made upon the representation format (see 4.1.2), the API has to ***support at least the JSON format***. Any additional format supported by the selected framework might be useful for unpredictable future clients. However, supporting JSON is mandatory. This makes up the second selection criterion.

An important aspect that defines the quality of an API is its simplicity in being integrated by client applications. The 5th requirement of the thesis' Goal 2, suggests that the API should be easy to use in order to develop future clients. As a result, the selected framework should be ***adequately documented and include examples*** of complete request/response examples. Moreover, several frameworks allow generating client stubs and in this way further facilitate client-side development. In this regard, frameworks that support ***client-libraries generation,*** for the main mobile providers will be preferred. Last but not least, frameworks that offer ***additional mechanism*** to secure the API and support asynchronous communication will be favored. Hence, adequate documentation, generating client stubs for mobile platforms and providing additional mechanisms make up the 3rd, 4th and the 5th selection criteria respectively.

An overview and comparison of all framework vendors goes far beyond the scope of this paper. For this reason, the most popular frameworks that more or less fulfill the criteria specified above are considered as candidates. Such frameworks are Jersey, RESTeasy, Restlet and the solution offered by GAE called Google Cloud Endpoints (GCE). All these frameworks implement the Java API for RESTful Web Services (JAX-RS) which is a framework that provides support for creating web services in Java, by defining how to publish Java code as a RESTful API using annotations (Hadley, 2009).

According to the first criterion, all frameworks mentioned above support Java. However, GCE is considered a better option for fulfilling this criterion as it is perfectly compatible with the GAE. Any potential change in the GAE infrastructure is expected to be included and reflected by its feature (GCE). Furthermore, according to the second criterion each of these frameworks supports JSON representations, whereas RESTeasy supports the largest variety of representation formats (XML, JSON, Atom etc.) compared to the others. According to the 3rd criterion, all candidate frameworks provide adequate documentation and examples over the internet. Hence, it is impossible to identify a better option in this case. Moreover, according to the 4th criterion, GCE offers a feature for generating client libraries for Android, iOS and Javascript clients, whereas other frameworks would require using some third party library for this purpose. Finally, all frameworks offer additional mechanism to support security and asynchronous communication, specified in the 5th criterion.

For the reasons discussed above, GCE is considered the most suitable technology for implementing Wahlzeit's API. Table 5.1, summarizes the defined criteria and shortly describes how and whether or not they are fulfilled by GCE.

| Criteria | Google Cloud Endpoints | Fulfilled |
|---|---|---|
| 1. Java support and GAE compatibility | Supports Java and is actually a feature of GAE | ✓ |
| 2. JSON support | For now, only JSON format is supported | ✓ |
| 3. Adequate documentation and examples | Google's official documentation and running examples are accessible over the internet. | ✓ |
| 4. Auto-genereted client stubs | Offers libraries to generate client code for Android, iOS and Javascript. | ✓ |
| 5. Additional built-in mechanisms | Offers built-in OAuth2.0 support and Push Notifications for asynchronous communication | ✓ |

Table 5.1: An overview of how Google Cloud Endpoints fulfills the specified selection criteria

## 5.2 Description of Google Cloud Endpoints

Google Cloud Endpoints, consists of tools, libraries and capabilities that are used to develop APIs from an App Engine application (Google, 2016a). It is a Mobile Backend as a Service (MBaaS) solution offered as a GAE feature to simplify API management and development process by offering an extensive list of features. This section describes GCE basics and the features that have been used to implement Wahlzeit's API and the Android client. These features facilitate the process of converting the existing GAE code into a backend API.

Simple ***annotations*** are used to decorate native code. As long as they are used properly, GCE will recognize parts of the application as parts of the API. For example, the **@*Api*** annotation is used in Java classes to expose all their public, not-static and non-bridging methods to the public API (Google, 2016b) and define generic API configurations. The **@ApiMethod** annotation is used to expose a single method and define the HTTP operation and the path of the relevant request to be executed in order to call this method. In this way, resources can be mapped into methods and query parameters into method parameters. GCE supports Java's primitive data types, collections and Plain Old Java Objects (POJOs) and recognizes them as query parameters. Annotations **@*Named*** and **@*Nullable*** are written before method parameters to name a query parameter and indicate whether the parameter is optional. Last but not least, **@*ApiResourceProperty*** annotations are used to decorate class variables and methods, and provide more control over the way resource properties are exposed to the API.

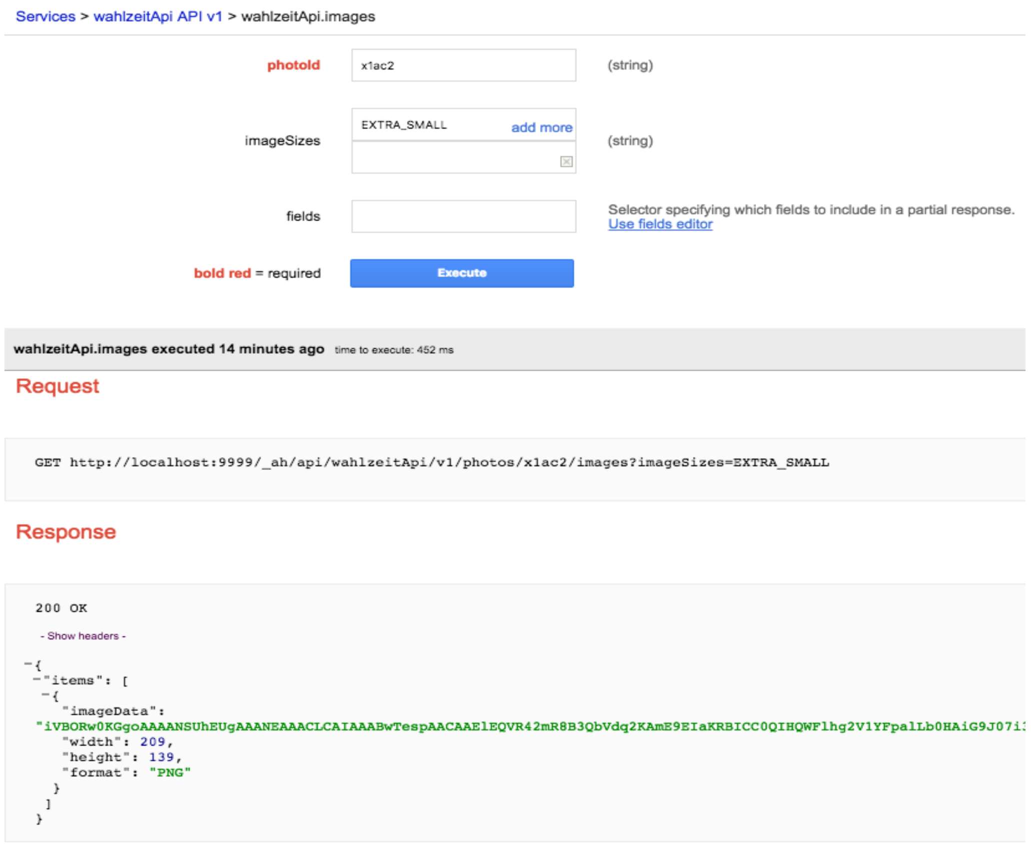
Assuming that the backend API is annotated properly, GCE can build Android, iOS and Javascript ***client libraries*** as well as ***discovery documents***, so each client application can use similar libraries. Thus, GCE facilitates development across multiple platforms. Client libraries contain service objects that help constructing and executing requests. Client libraries also contain all necessary POJO files that are used in the backend's methods, either as method parameters or as return values. Hence, clients avoid constructing requests using JSON directly, and focus on their native development language. Furthermore, discovery documents describe the surface of a particular version of an API and provide information about its resource schemata, authentication scopes and methods (Google, 2016c).

Another feature offered by GCE, is its built-in ***OAuth2.0 support*** that secures the API and restricts all or parts of it to be accessible only by authorized applications. By using the Google Cloud Platform Console[5], authorized clients are specified by generating client identifiers (clientIds) using client secrets. For example, an SHA1 fingerprint is used as a sercet to register Android clients. At runtime, a client application is granted the authorization token it needs to send requests to the API backend if its client secret

[5] https://console.developers.google.com

matches a client ID within the backend's client ID whitelist (Google, 2016d). By adding a User parameter to the API’s backend method, GCE automatically authenticates the user by looking up its client secret in the relevant client ID list. Hence, the development effort that would otherwise be needed to secure the API is eliminated.

To conveniently navigate and explore an API, GCE offers ***Google API Explorer*** that allows developers read docs and execute requests against an API from a web browser. This tool runs automatically by adding the _ah/api/explorer suffix to the website’s URL, and works either locally or on a deployed version over the internet. Its GUI helps constructing authorized or unauthorized request and specify the relevant query parameters. Figure 5.1 shows the response on a request made to some photo’s images resource (/photos/{photoId}/images?imageSize) in Wahlzeit’s API.



Services > wahlzeitApi API v1 > wahlzeitApi.images

| | | |
|---|---|---|
| **photoId** | x1ac2 | (string) |
| imageSizes | EXTRA_SMALL | (string) |
| fields | | Selector specifying which fields to include in a partial response. Use fields editor |
| **bold red** = required | Execute | |

wahlzeitApi.images executed 14 minutes ago time to execute: 452 ms

Request

```
GET http://localhost:9999/_ah/api/wahlzeitApi/v1/photos/x1ac2/images?imageSizes=EXTRA_SMALL
```

Response

```
200 OK

 - Show headers -

-{
 -"items": [
  -{
    "imageData":
 "iVBORw0KGgoAAAANSUhEUgAAANEAAACLCAIAAABwTespAACAAElEQVR42mR8B3QbVdq2KAmE9EIaKRBICC0QIHQWFlhg2V1YFpalLb0HAiG9J07i
    "width": 209,
    "height": 139,
    "format": "PNG"
   }
  ]
 }
```

Figure 5.1: Using Google’s API Explorer to retrieve the images of some photo in Wahlzeit’s API.

## 5.3 Essential API implementation steps

Hereby, the API's implementation process is described. The following sections reflect phases and important steps of the development process. The first two sections (5.3.1 and 5.3.2) portray the procedure of setting up the environment and writing the API. Sections 5.3.3 and 5.3.4 describe the process of adjusting Wahlzeit's model classes and transferring images over the API respectively. Finally, section 5.3.5 outlines the migration effort for adjusting the authorization method to OAuth2.0 standards as necessary.

### 5.3.1 Setting up the environment

Wahlzeit can either run locally or be deployed to GAE for broader accessibility. Gradle building tool downloads all necessary libraries and manages dependencies for the application to build and run properly, and is responsible for deploying the application to the cloud. This section outlines the additional adjustments that are required to write and deploy Wahlzeit's API and take advantage of GCE's features in order to secure the API and generate client libraries (as described in 5.2).

To ***write and deploy the API***, the Appengine Local Endpoints 1.9.8 library has been added as a dependency to Wahlzeit's build.gradle file. By synchronizing and building the project, the relevant GCE libraries are added to the project. These libraries contain all necessary GCE classes needed to write annotations and define API related configurations.

For ***generating client libraries***, the endpoints closure has been added within the appengine closure and the getClientLibsOnBuild and getDiscoveryDocsOnBuild attributes have been specified. These attributes, when set to true, download the client libraries and the discovery documents before the war task is being called. These libraries will be later used by the Android client to make requests to the API.

To run the API a new servlet is needed to ***handle the requests sent to the API***. In this regard, the SystemServiceServlet has been added to the web.xml file. This file manages and distributes the ingoing requests to the appropriate servlets. For API calls, applications send their request to the _ah/api path and the backend handles these requests at the _ah/spi path. Behind the scenes, each request sent to an endpoint is mapped by the SystemServiceServlet to a request in the endpoints service provider (endpoints classes as described below). Hence, all API classes have to be explicitly defined in this servlet, in order for the API requests to succeed.

To ***make requests and secure the API,*** configuring the Google Cloud Project in the Google Cloud Console Platform is necessary. Wahlzeit 2.0, requires a cloud project to be able to deploy an application instance to the cloud and manage its database entries, keep track of its logs etc. For accessing the API, configuring this cloud project was necessary. Specifically, clientIds have been created to define Android and Web applications that are authorized to make API request (as described in 5.2, OAuth2.0 support). For obtaining an Android client ID, the cloud project requires the SHA1 key to register the Android client's package name to the allowed clientIds whitelist. This, client ID is later used for making authorized requests from Wahlzeit's Android client.

### 5.3.2 Using Google Cloud Endpoints to write the API

Endpoint libraries do a lot of work behind the scenes to map API requests to Java code. This section describes the process of writing Wahlzeit's API as designed above (see 4.2). The design is implemented by writing Java classes and methods and annotating them appropriately.

According to the defined design, three endpoint classes are written, one for each URI path (refer to Figure 4.1). These classes contain methods to handle the requests made to each resource. In order to map incoming API requests to be handled by these endpoint classes, the name of each class is declared in the SystemServiceServlet so it can be identified as a service provider from the backend.

Using the correct annotations allows converting the Java classes to Cloud Endpoint classes. These classes are decorated with the *@Api* annotation to specify generic API configurations. Hence, a name, a description and a version are specified. The clientIds attribute is used to specify which clients are authorized to execute requests. In addition, the audiences attribute is required in case the API supports requests from Android clients (Google, 2016b). Moreover, the scope attribute is declared to specify which OAuth2.0 scopes are required to be granted, in order to access this method. Thus, client applications request the user to grant access according to the specified scopes so they can access the API. The following snippet depicts the *@Api* annotation with its specified attributes, as used in PhotosEndpoint.class (Figure 5.2).

```
@Api(name = "wahlzeitApi",
     version = "v1",
     description = "A multiclient API for Whalzeit",
     clientIds = {
        Constants.WEB_CLIENT_ID,
        Constants.ANDROID_CLIENT_ID,
        API_EXPLORER_CLIENT_ID },
     audiences = {
        Constants.WEB_CLIENT_ID,
        Constants.ANDROID_CLIENT_ID },
     scopes = {
        https://www.googleapis.com/auth/userinfo.email }
    )
public class PhotosEndpoint { ... }
```

Figure 5.2: API-scoped annotation to convert a Java class into an API class.

Identical annotations are used to decorate the other two endpoint classes (PhotosCasesEndpoint, ClientsEndpoint). Moreover, a Constants.java class is used to store all client IDs generated in the Google Cloud Console Platform, as shown in the snippet above.

Method-scoped annotations (see, Figure 5.3) allow mapping Java methods to API methods (resources). The @ApiMethod annotation is used in each method to configure the request's parameters. The annotation's path attribute is used to shape the URIs of each resource to fit the defined URIs of the design (Google, 2016b). In some methods the httpMethod attribute is specified to declare the HTTP operation that should be enclosed with the request in order to call this method. When this attribute is omitted GCE does some clever work behind the scenes and uses the method's name to assign the correct HTTP operation. Moreover, method names are declared which are then defined in the generated client libraries and are used by the Android client to construct requests.

Additionally, method parameters can be annotated to match request parameters. The @Named and @Nullable are used to map query parameters to method parameters and specify the optional character of a parameter respectively. Moreover, the injected types User and HttpSerlvetRequest method parameters are specified, as required by GCE to secure API methods and determine information about the request on runtime.

The snippet below (Figure 5.3) shows the signature of the API method written to handle handle the task of retrieving some photo's images based on the specified images sizes (/photos/{photoId}/images?imageSize). An authorized request on this method using API Explorer is illustrated in Figure 5.1.

```
@ApiMethod(name = "images",
           httpMethod = HttpMethod.GET,
           path = "photos/{photoId}/images")
public Collection<Image> listAllImages (
                        User user,
                        @Named("photoId") String photoIdAsString,
                        @Nullable @Named("imageSizes") PhotoSize[] sizes
                         ) throws UnauthorizedException { ... }
```

Figure 5.3: Method-scoped annotations for mapping a Java class to an API class

The API methods defined in each Endpoint class instantiate and use objects of classes that belong to the web application, to perform several tasks. For example, the images API method depicted above accesses the single object of the PhotoManager class, to fetch the relevant photo from the database and henceforth retrieve its images. Other, API methods use objects in a similar way to perform CRUD operations, without directly accessing the persistence layer. An overview of how API methods and the application's classes are related to each other is given in the diagram below (Figure 5.4), which extends the API's resource diagram presented above.

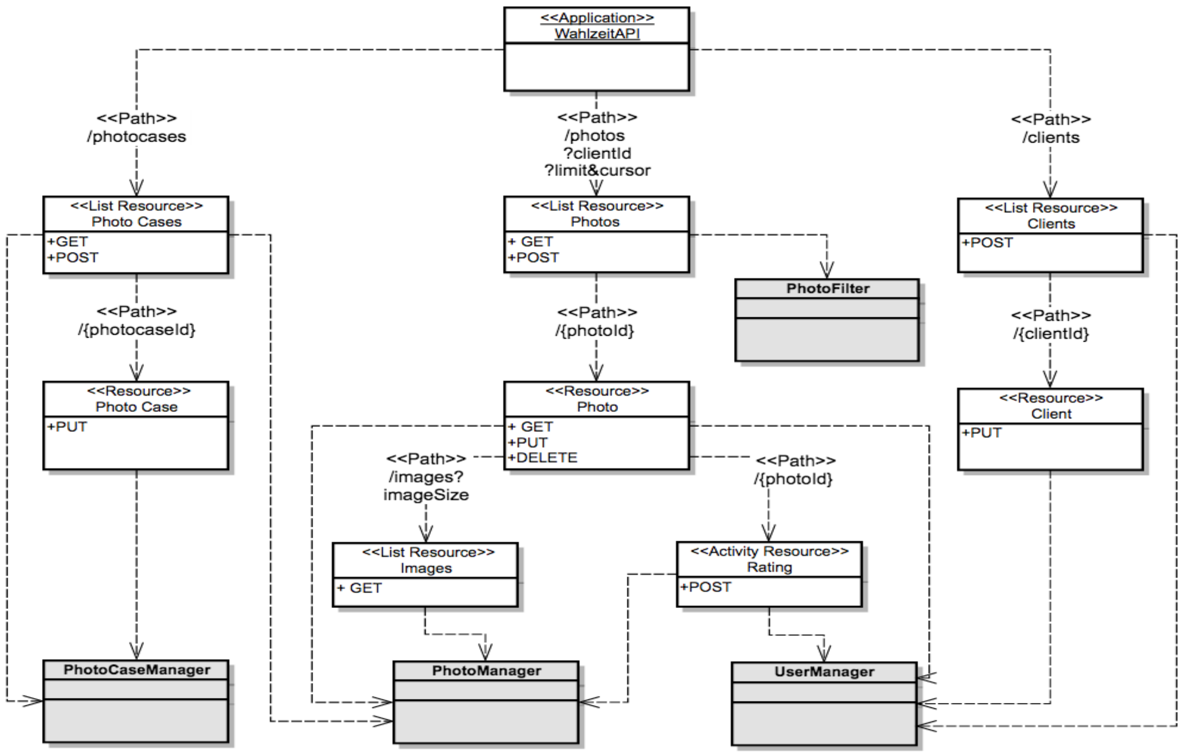


Figure 5.4: Class diagram describing Wahlzeit's resources and their relation to classes of the web application.

### 5.3.3 Adjusting Wahlzeit's model classes

All API methods, return objects or object collections that are converted by GCE libraries into JSON representations. Wahlzeit's backend services uses the POJO classes located in org.wahlzeit.model package for this purpose. However, these objects were not designed to be converted and used as messages. Hence, adjusting these objects according to the API's and Android client's needs is necessary.

Wahlzeit's model classes contain information irrelevant to the client applications. For example, a Client object provides getter methods for its writeCount which is relevant to the persistence layer. Furthermore, it exposes its httpSessionId and language configuration which are relevant to the browser's user session and UI template respectively. This redundant information could lead to unnecessary overhead and increase response times. For this case, the annotation @ApiResourceProperty is used to decorate either class variables or their getter methods, in order to omit these properties from a resource's representation. This annotation is extensively used in other objects such as Photos and PhotoCases, and their related classes.

Often Wahlzeit's model objects contain less information than needed to perform specific tasks. As discussed during the design phase (see 4.2.3), to achieve stateless communication and avoid the need of relating requests to each other, resource representations have to contain all relevant information needed to perform specific operations. For example, Photo's representation does not provide any information about the rating value and the praising client. Thus, it would be impossible to determine this information, which in the web application is stored and retrieved from the user session. To solve this issue, relevant class variables and getter methods have been defined in the Photo, Client and PhotoCases classes.

### 5.3.4 Transferring photos and images appropriately

As discussed during the resource design (see 4.2.1) photos refer to objects in the scope of Wahlzeit which along with other metadata (tags etc.) are associated with image of various sizes (binary data). As mentioned during the representation format design decisions (see 4.1.2) there are many ways to transfer images that are associated with a photo, without having to support content negotiation. To achieve this, images and photos can either be sent in separate request or together in one request.

For ***sending photos and image files in separate requests***, the server should be configured accordingly. For example, when a client would like to make an upload, he should initially send a request with the photo. For the relevant photo, the server should generate URLs to which images could be uploaded and send these URLs in response. Thereafter the client could post images to the specified URLs. Oppositely, when a client would like to download a photo's images, he should send a request with the photo and the server should send the image's URLs in the response, which the client could use to download the image files. This approach would have an impact on stateless communication as discussed during the design phase (see 4.2.3), and would require the client and the server to associate requests.

According to the second approach ***images and photos could be sent in the same request*** right away. The easiest and most compatible way to send all data in one request would be to serialize a photo's images into some binary serialization format and include them to the resource as properties. A common binary serialization format would be for example Base64, which is also used to save images to Google Cloud Storage. This would allow photo resources to enclose all relevant information. However, transforming binary data to text representation increases the size of the transferred message.

Wahlzeit's API uses a combination of both approaches to upload and retrieve images. When a client application makes an upload, an image is enclosed in the photo's resource representation, serialized in Base64 format. When the server receives the request he is responsible for generating and saving the images in various sizes. This way, the burden for processing and generating images is removed from the client and an upload task can be handled by a single request. The overhead of transferring only one image in Base64 format should be negligible.

To download a photo, clients have to initially retrieve the photo's resource and thereafter request and retrieve an image of their preferred size. Although, this requires the client to associate the requests it allows retrieving only the relevant image size.

### 5.3.5 Adjusting the authorization method

Wahlzeit 2.0 is an application that uses the ClientLogin API to allow user to login the application, by redirecting them to the appropriate login page. However, the technology was deprecated and permanently disabled in May 2016. As a result, although not relevant to the API, the login process had to be adjusted according to the suggested OAuth2.0 standards and in this way, an outage in Wahlzeit's ability to access the App Engine services has been prevented.

For Wahlzeit to access Google's Oauth2 API and retrieve information about the user (like name, gender etc.) a standard process has to be followed (depicted in Figure 5.5). First of all, the application should obtain client credentials from the Google Developers Console Platform. Henceforth, by using these credentials the application would be able to request an access token from Google's authorization server. In case a user logs in with his Google account and consents to grant access to the requested scopes, the authorization server replies with an ***authorization code***. Thereafter, this authorization code has to be exchanged with an access token. Finally, the access token can be send to the preferred Google's API (Google+, UserInfoPlus) and in this way query information about the user's profile. The implementation of this process is described below in more detail.

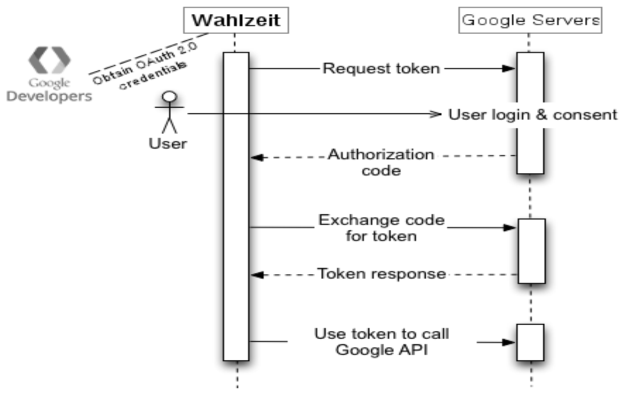


Figure 5.5: Overview of the authorization flow (Adjusted from Google, 2016e)

***Obtaining OAuth 2.0 client credentials*** from the Google Developer Console allows Google to identify Wahlzeit as an authorized application and is required to allow the application to ***send requests to the authorization server.*** For convenience, these credentials have been extracted into a JSON file named client_secret, which has been saved in the application's source code. Furthermore, an AuthenticationUtil class has been written to read the client secrets from the file and further support the authentication process.

Google's authorization server requires Wahlzeit's users to ***log in*** their account and thereafter a consent screen is displayed. This ***consent screen*** asks users whether Wahlzeit is permitted to have access to specific scopes. To define which scopes will be used in the consent screen, the relevant scope values are defined in the AuthenticationUtil class. Hence userinfor/email and userinfor/profile are defined so that Wahlzeit will request access to users' email addresses and profile information respectively. Those scopes are preferred over others since they do not require access to sensitive information like for example list of circled people, list of installed Android apps etc. In this way, it is less likely that suspicious users will decline permissions to Wahlzeit.

In case a user consents to allow Wahlzeit access the defined scopes the ***authorization server replies with a callback,*** which contains the ***authorization code***. The server uses the web page's URL and the /oauth2callback suffix to send the callback to Wahlzeit. Thus, a new AuthenticationCallbackServlet class has been written to handle this callback and further support the authorization flow.

This class is responsible for ***exchanging the authorization code with an access token,*** which is thereafter used to make a ***request to the User Info Plus API.*** In this way, information about a user's Google profile are retrieved and used to register a new Wahlzeit client and allow him to access the restricted features of the application, which are available only to authorized users. Since the access token has a limited lifetime for about an hour, ***refresh tokens*** can be used to obtain new access tokens in case additional request need to be sent.

## 5.4 Essential Android client implementation steps

This section gives an overview of the process that has been followed to implement Wahlzeit's Android client. For code editing, debugging and an instant deployment environment, the official IDE suggested by Google (Android Studio 1.5.1) and the Android SDK 6.0 have been used. User stories were extracted from the web application's features (see Appendix A) and emphasis was given in interacting with the API and helping users quickly and intuitively discover the application's content.

### 5.4.1 Defining the main navigation pattern and designing screens

Applications on mobile devices are much different than web applications. Screens are smaller and users use touch gestures instead of pointers to interact with the application. Hence, an efficient GUI design has a major impact on the development process and the overall performance of the application (Wasserman, 2010). By taking into account the web application's user interface, this section chooses an appropriate main navigation pattern. Furthermore, the necessary number of screens is determined, and for each screen mobile UI components are selected to match the relevant components used in the web application. The resulting screen diagram is depicted below (see Figure 5.6).

The ***main navigation pattern*** is expected to facilitate users navigate and discover the applications content. Wahlzeit's web application uses a single level navigation bar menu which populates when a user logs in according to his access rights. Since the number of menu categories might be more than six (in case the user has administrative rights) and there is no need to display the navigation options to the user constantly, a transient navigation pattern is recommended (Neil, 2004). Transient in this context, means that the navigation options are hidden and can be revealed with a tap or gesture whenever a user needs to navigate to a different screen. Hence, a vertical sidebar (known as ***navigation drawer***) will be used. This sidebar will accommodate the same navigation option as the web application, and populate accordingly.

By considering the navigation options of the web application a series of screens is identified (***show, tell, home, profile, upload, moderate***). These screens should be accessible through the navigation drawer, hence they are depicted in the same raw in the screen diagram (see Figure 5.6). On top of these screens, a ***login screen*** is defined as a welcome screen, to allow the users to login or continue as guests. In this way, the access rights will be determined and thereupon the navigation options. Similar to the web application new screens will be used when a user chooses to ***flag*** or ***edit*** a photo. Finally, to share, take or choose existing photos the devices ***e-mail***, ***camera*** and ***gallery*** applications, installed in the device, will be used respectively.

The following diagram (Figure 5.6), outlines the hierarchy of the defined screens. The arrows imply that one screen is directly reachable through the other.

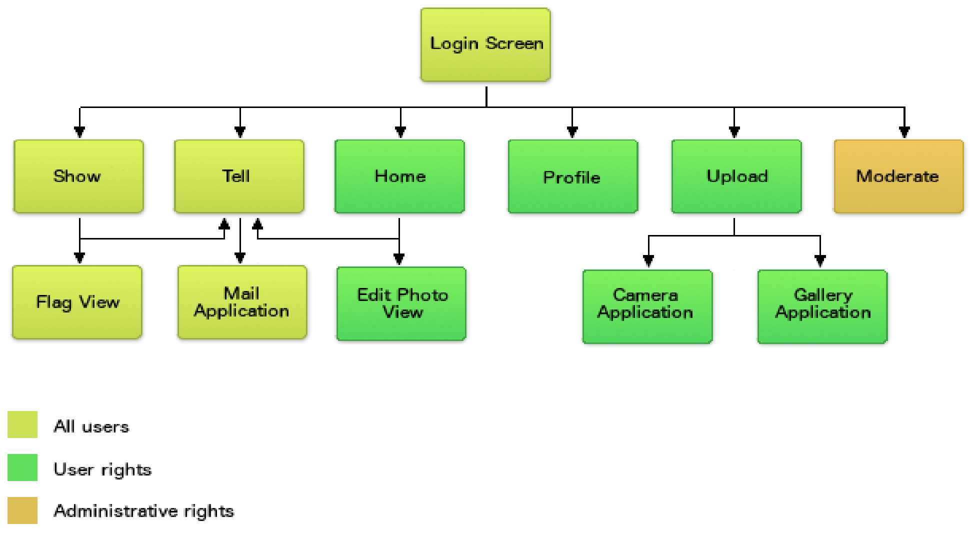


Figure 5.6: Screen diagram for users with different access rights

Android provides a variety of pre-built UI components that match the GUI components used in the web application. As a result, ***editable text views***, ***text views***, and ***spinners*** (dropdown boxes) will be used (Android, 2016i). Furthermore, the radio buttons used for rating will be replaced with a ten-star ***rating bar***, and instead of checkboxes ***switch buttons*** will be used throughout the application.

Further mobile UI patterns have to be used in order to implement other features of the web application. For example, the show page shows all uploaded photos one at a time, and allows users to rate or perform actions on each photo individually. To achieve this on the mobile client, a ***card stack*** will be used. Each card will display a single photo, whereas only the card on top of the stack will be visible. This way, skipping will be possible by swiping the cards off the screen towards any direction.

Last but not least, the web application provides constant access to some global settings. In this way, users may for example modify the language settings (English, German) from any page. For the mobile client language options will also be accessible from each screen by using the application's ***action bar***. Hence, users will be able to change the language, and the whole application will be reloaded accordingly.

### 5.4.2 Writing Wahlzeit's Android client

Having defined the number of the application's screens and how they relate to each other, this section describes how Android components (described in the literature review, see 3.4.2) have been used, to implement the defined screen design.

In order to allow users to login the application or continue as guests a LoginActivity class has been defined and is the first ***activity*** (see 3.4.2.2) that is launched when the application is started. This activity is responsible for triggering ***asynchronous requests*** (see 3.4.2.4) to Google's authorization server and Wahlzeit's backend API. Having determined the user's access rights, the activity uses an ***intent*** to launch the MainActivity.

The MainActivity is responsible for creating and populating the navigation drawer (sidebar) according to the relevant access rights. Android guidelines suggest using ***fragments*** (see 3.4.2.3) to switch between screen layouts into a content area (Android, 2016j). Hence, all the screens of the second row of the defined screen hierarchy (show, tell, home etc.), will be composed using fragments. The navigation drawer manages the lifecycle of those fragments and displays them on top of the MainActivity's layout. Each of those fragments is displayed when the relevant option is selected. Moreover, the factory pattern (Gamma, 1995) is used to create instances of Wahlzeit's fragments, and in this way the fragments' creation logic is hidden from the MainActivity. Figure 5.7 gives an overview of this implementation.

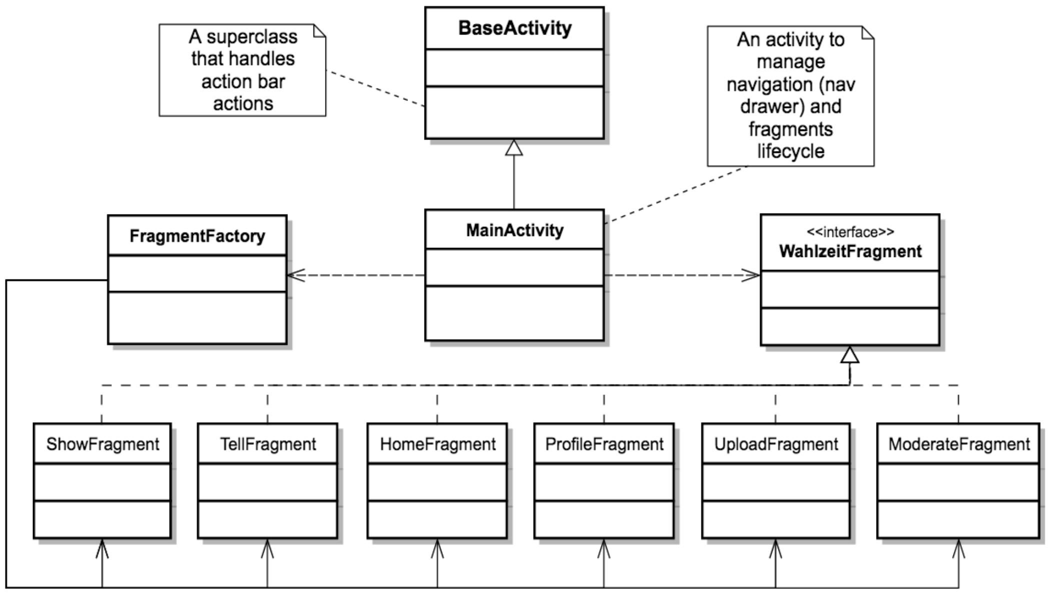


Figure 5.7: Class diagram that depicts the factory pattern used to create fragment objects by the MainActivity.

Activities are also used to compose the screens for editing and flagging a single photo. All activities defined are subclasses of the BaseActivity abstract class (as shown in Figure 5.7 for the MainActivity). This class is responsible for adding options to the action bar and performing actions when a user makes a choice. Hence, the action bar is available throughout the application's screen and can be used to switch the application's language anytime.

All activities and fragments are related to layout files in the application's ***resources*** directory (res) (see 3.4.2.4), which are used to specify and construct the layout of each screen. The resources directory also contains drawables that are used to display Wahlzeit's logo in the login screen or decorate the navigation drawer's options. Moreover, in order to support both English and German, text representations in both languages have been created.

As described above, Wahlzeit's mobile client will use the device's installed applications to access mobile built-in features. To achieve this, TellFragment and UploadFragment will send ***action broadcasts*** to delegate tasks to the e-mail, camera and gallery application. Those applications receive the broadcasts and provide their resources to perform the assigned tasks. Wahlzeit itself also acts as a ***broadcast receiver*** to perform operations in response.

Last but not least, to allow the application to send broadcasts, intents, and determine which permissions are required, in order to access the internet and protected parts of the OS, all relevant information is declared in the application's ***Android manifest*** file. This file is also responsible for defining the application's components structure. In this way the LoginActivity is defined as the first activity to start the application and parent activities are specified to navigate appropriately when the back button is tapped.

### 5.4.3 Using the generated client stubs to interact with Wahlzeit's API

Having defined the application's components and how they connect to each other, this section explains how the application uses GCE generated client libraries to authorize and interact with the backend.

The very first time the application is started, the login activity launches an AccountPicker instance that prompts the user to choose from a list of his accounts, and thereafter stores the account to a native file called SharedPreferences. This account will be used to allow the application access Wahlzeit's backend. Hence, the login activity is responsible for constructing a Google account credential, that is used to make authorized API request. If a user chooses to login, a request is sent to OAuth2.0 server to retrieve information about the user' profile. This information is used to construct a Client object, which is then posted to the backend asynchronously (see Wahlzeit's /clients resource in Table 4.3).

To interact with the backend, the Android client uses the auto-generated client libraries in Java (see 5.2). The library's service objects are named according to the specified method names, declared when annotating the API's methods (see 5.3.2). To construct a request, the client application refers to those names. The service objects are also used to specify the required query parameters. Hence, the Android client can interact with the API by using native code only.

By using the client libraries, synchronous requests are sent to the server, which blocks the main thread and waits until the server responds. Since Android does not allow blocking its main thread (see 3.4.2.4) all API calls will be executed inside asynchronous tasks. After receiving the parsed response, the relevant information will be stored dynamically in a utility class and the registered application component will be notified accordingly (see 5.4.4, when describing how the singleton and observer patterns have been used).

Figure 5.8 shows the snippets of code used in an asynchronous execution to retrieve some specific image sizes of a photo (refer to the API methods, see Figure 5.3). To achieve this the credential property constructed from the chosen account is used to construct an API service handler. This handler is used to construct a Wahlzeit's API request object (getImagesRequest) and pass the relevant query parameters, namely the photo's UUID (photoId) and the relevant images sizes (large, medium etc). When the execute() method is called, the request is sent to the server and the parsed response is received.

```
                              /* API method */

@ApiMethod(name = "images",
           httpMethod = HttpMethod.GET,
           path = "photos/{photoId}/images")
public Collection<Image> listAllImages (
                        User user,
                        @Named("photoId") String photoIdAsString,
            @Nullable @Named("imageSizes") PhotoSize[] sizes
                        ) throws UnauthorizedException { ... }



                 /* Snippets used in the Android client */

WahlzeitApi wahlzeitApi = new WahlzeitApi.Builder(HTTP_TRANSPORT,
                                                  GSON_FACTORY,
                                                  credential).build();

WahlzeitApi.Images getImagesRequest = wahlzeitApi.images(photoId)
                                      .setImageSizes(Arrays.asList("LARGE”);

ImageCollection getImagesResponse = getImagesRequest.execute();
```

Figure 5.8: Using endpoints generated libraries to make an authorized request to Wahlzeit's API Images resource.

### 5.4.4 Abiding to OOP and Android's design principles.

Wahlzeit's web application, serves as a solid codebase to facilitate developers build their own photo sharing applications. Similar to the web application, the mobile client is also designed according to OOP principles. This section describes how Wahlzeit abides to the Android platform's design principles and uses design patters. This way the application can be understood and extended by future developers.

All mobile application platforms organize their structures according to the ***Model View Controller*** (MVC) behavioral pattern (Campos et al., 2015). Wahlzeit's activities and fragments (controllers) use XML layouts (view) to specify their screens' structure. Moreover, the generated client stubs provide POJO classes that are necessary to construct requests and parse responses. For the model component of the MVC pattern, mainly these classes are used. In addition, more POJO classes are composed to store information temporarily in order to construct GUI elements. For instance, the classes PhotoListItem and NavDrawerItem are used to populate the user's photos list in the home screen and the navigation drawer respectively.

For pulling information out of collection resources and displaying the collection on the screen, Android principles suggest using the ***Adapter pattern*** (Gamma, 1995). In Wahlzeit's mobile client, adapters are used extensively for creating customized views like the cards stack, the user's photo list etc. For the user's photos list for example, the PhotoListAdapter is used to adapt the Photo objects and convert them into PhotoListItems that can be placed into the list. Furthermore, it provides all necessary methods to retrieve or add photos to the list. In this way, when a user retrieves photos from the photos resource (/photos) using the API's pagination feature (see 4.2.1), the list can display the newly retrieved photos and populate dynamically.

At this moment the mobile client does not use a database to permanently store information. However, during a running application session, all information retrieved from the server by executing asynchronous tasks is stored temporarily into a class called ModelManager. This class is used from controller classes (activities and fragments) across the application to populate their views. To ensure that there is only one instance across the application with a global access point, the ***Singleton pattern*** is used (Gamma, 1995). The same pattern is used for the CommunicationManager that manages connection details.

For sending notification across the application and notifying components about state changes according to the ***Observer pattern*** (Gamma, 1995), Android suggests using the LocalBroadcastManager instance. This utility is extensively used throughout the mobile client's implementation for operations of indeterminate time, such as reactions to API calls. It is also used to respond to user input. For instance, the user's photo list uses the API's pagination feature to make subsequent calls to retrieve parts of the photos collection. Hence, every time a user pulls down the list, a new request is sent to the server and the home fragment subscribes for the response (using the

LocalBroadcastManager). When the server replies, the fragment is notified and the list is populated accordingly.

### 5.4.5 Organizing code structure and conventions

Mobile application should be neatly organized with a clear folder structure and naming conventions to ensure that the code is clean and maintainable. In addition, using proper naming conventions for components and variables is equally important. This section describes the package's structure and the naming conventions used for developing the Wahlzeit's Android client.

By default, Android Studio displays project files organized by modules and file types to simplify navigation and hides certain files that are not commonly used. Hence, the project is separated to Gradle Scripts which contain all the files related to managing the whole project and the app module which further splits to manifest (see 3.4.2.1), java and resource files (see 3.4.2.5). The java directory contains the auto-generated client stubs from the backend and the source code. The client stubs should be re-generated every time changes are made to the API.

Java packages are usually organized by layer, reflecting the various application layers, or by feature, reflecting the feature sets. For organizing the java source code directory, emphasis is given to the application's component and thus packages have been organized by layer. Although this structure is blamed for low cohesion within the packages and high coupling between them, keeping the folders organized like this makes the code look logically organized and scanning through the code is a pleasant experience. The following table (see Table 5.2) outlines all packages and describes the corresponding components. (In Appendix B, packages are illustrated in a package diagram).

| Packages | Components |
|---|---|
| com.wahlzeit.mobile.activities | Activities |
| com.wahlzeit.mobile.adapters | Lists and card stack adapters |
| com.wahlzeit.mobile.fragments | Fragments and fragment factory |
| com.wahlzeit.mobile.listeners | Contains all event listeners |
| com.wahlzeit.mobile.model | Data models used in the UI |
| com.wahlzeit.mobile.network | Asynchronous tasks |

Table 5.2: Java packages organized by layer.

Android components should follow particular naming conventions to indicate their purpose. Activity and fragment class names are written using the camel case convention and specify their type in their suffix, for example TellFragment. The same applies to fragments, adapters, event listeners etc. Furthermore, layout files are written using the snake case convention and use the reverse order, for example fragment_tell.

# 6 Conclusion

In this thesis, Wahlzeit has been extended by implementing an API that provides all necessary information to accommodate multiple clients. To achieve this, the thesis went through the necessary design decisions to be made when designing and implementing a web API. Cloud Endpoints has been chosen as the most appropriate technology to implement a REST API for Wahlzeit. The API has been implemented successfully, and allows system to interoperate by exchanging JSON messages via the HTTP protocol. As a proof of concept an Android client has been developed to interact with the web application.

## 6.1 Goals overview

In Chapter 2 the goals and requirements of this thesis have been defined. This section reviews the means that have been used to achieve those goals, and argues whether or not they have been accomplished.

***Goal 1*** concerns analyzing the design decisions to be made when connecting clients to the service. To achieve this, Chapter 3 thoroughly reviewed contemporary interface structure techniques, data infrastructure formats and communications protocols. Thereafter, subsections 4.1.1, 4.1.2 and 4.1.3 analyzed the decisions in order to define a resulting design, by taking into account the aspects of the existing application. According to the first design decision, REST was indicated as the most appropriate style to design the service's interface. Furthermore, regarding the appropriate data infrastructure and communication protocol, JSON representations and the HTTP protocol have been chosen respectively. Thereby, the first goal is considered fulfilled.

***Goal 2*** concerns extending the existing application by adding a headless service. Therefore, in section 4.2 a design for Wahlzeit's API has been defined. To implement this design, section 5.1 has defined selection criteria to choose an API development framework, by taking into account the design decision (see 4.1) and the requirements specified in Goal 2 and Goal 3. Cloud Endpoints has been chosen as the most suitable technology according to those criteria and its features have been described in section 5.2. Furthermore, section 5.3 describes the process followed to implement the defined API.

To answer whether or not this goal has been fulfilled, the API requirements that have been set in this goal have to be reviewed. Table 6.1, gives an overview of those requirements and describes how they have been fulfilled by the thesis.

| # | Requirement | Solution | Fulfilled? |
|---|---|---|---|
| 1 | Accommodate as many clients as possible | Allow clients to communicating with JSON message sent over the HTTP protocol. | ✓ |
| 2 | Expose the features and the functionalities of the existing web application | Features and functionalities have been extracted, from the web application (see 4.2) and the API has been designed accordingly. | ✓ |
| 3 | Cohere with the existing infrastructure and technologies | Choose Google Cloud Endpoints framework which is a trusted feature for Google App Engine applications. | ✓ |
| 4 | Keep the interface that runs on web browsers fully functional | The web application's is working flawlessly in parallel with the API. | ✓ |
| 5 | Facilitate future students to implement additional features or client applications | Google API Explorer can be used to discover the API by sending interactive requests, there is adequate documentation accessible over the internet, and Auto-generated client stubs for iOS and Javascript clients are available | ✓ |

Table 6.1: Checking whether the requirements set in Goal 2 have been fulfilled.

***Goal 3*** concerns implementing an Android client to interact with Wahlzeit's API. According to goal's first requirement, the mobile application has to offer the same features as the web application. To achieve this, user stories (Appendix A) have been extracted based on the web application's features to ensure that the mobile client will offer the same features and functionalities. The process followed to develop the Android client (see 5.4) is based on those user stories to define the application's screens and thereafter compose the application. Hence the first requirement is considered to be fulfilled. The second optional requirement, aspires to enhance the user experience by using the devices build-in features. According to the defined screen design (see 5.4.1), the application's screens should be able to interact with the device's installed camera, gallery and email applications. Section 5.4.2, outlines how the application uses Android's broadcast actions to delegate tasks to other installed applications and in this way use the devices built-in features. In this regard, the second requirement is also considered fulfilled.

## 6.2 Future work

Extending the application by adding a multi-client API and implementing a mobile client are important steps to achieve Wahlzeit's vision (see 1.1). However, there are a lot of additional features required to make the application more attractive and more effort needs to be invested so it can be offered as a standalone solution for rating and sharing photos.

First of all, regarding the web application, renovating the UI is recommended. To achieve this, modifying the application's CSS and Javascript files is necessary. Henceforth, the handler classes which are responsible for populating the application's UI have to be adjusted as well. Emphasis should be given to make the web site responsive to fit various screen sizes in order to be optimally used from web browsers running on mobile or tablet devices. To facilitate this process, frontend development frameworks like Twitter Bootstrap, Zurb Foundation etc. should be considered.

Secondly, relating the application to other social media platform is suggested. To achieve this, as first step would be including sharing options that will allow users to share photos in the most popular media platforms (Facebook, Twitter, Instagram). At the moment the only way to share photos is via email, which is rather outdated. Furthermore, adjusting the authorization method, to sign in using social media platforms' accounts should also be considered. Henceforth, the users might be given an option to share photos from their social media accounts. As a result, populating their Wahlzeit account would be much easier and the application would be quickly filled up with more content.

A minor fix that is suggested, is showing another user's profile page, when a user clicks on the name of the uploader. At this moment the application behaves unpredictably, by adjusting the photo filter and using the user's name as a filter parameter.

Apart from the additional features and fixes suggested, developers may use Wahlzeit's API to make the application more accessible. Similar to the Android application, the generated client stubs may help developers build iOS or Javascript. Although native applications are more reliable and usually perform better, using a cross development platform like Apache Cordova is recommended, to make the application reach all different mobile platforms including Windows Phones.

Finally, regarding the API and the implemented Android client there are additional features beyond the scope of this paper that have to be considered. For example, the current client does not have a persistence layer to store the information retrieved from the server.

Finally, there are features in the Android client and the Wahlzeit's API, that have not been addressed by this thesis. For instance, the Wahlzeit mobile does not use a local database to save the photos and client's details. Furthermore, Cloud Endpoints push notifications feature should be considered to notify clients about state changes in the backend and other relevant information. Last but not least, adjustmenting the OAuth2.0 authentication process is necessary, so that administrators' access rights will be assigned only to the user that deployed the application.

# Appendix A Wahlzeit mobile user stories

The following table summarizes the user stories that have been used to develop the mobile client. User stories are organized in sections based on the application's features. Each user story is defined in the form: I want to <do something> so that I can <achieve something>, to be verified by the following acceptance test <T-1: Test description>, <T-2: Test description>.

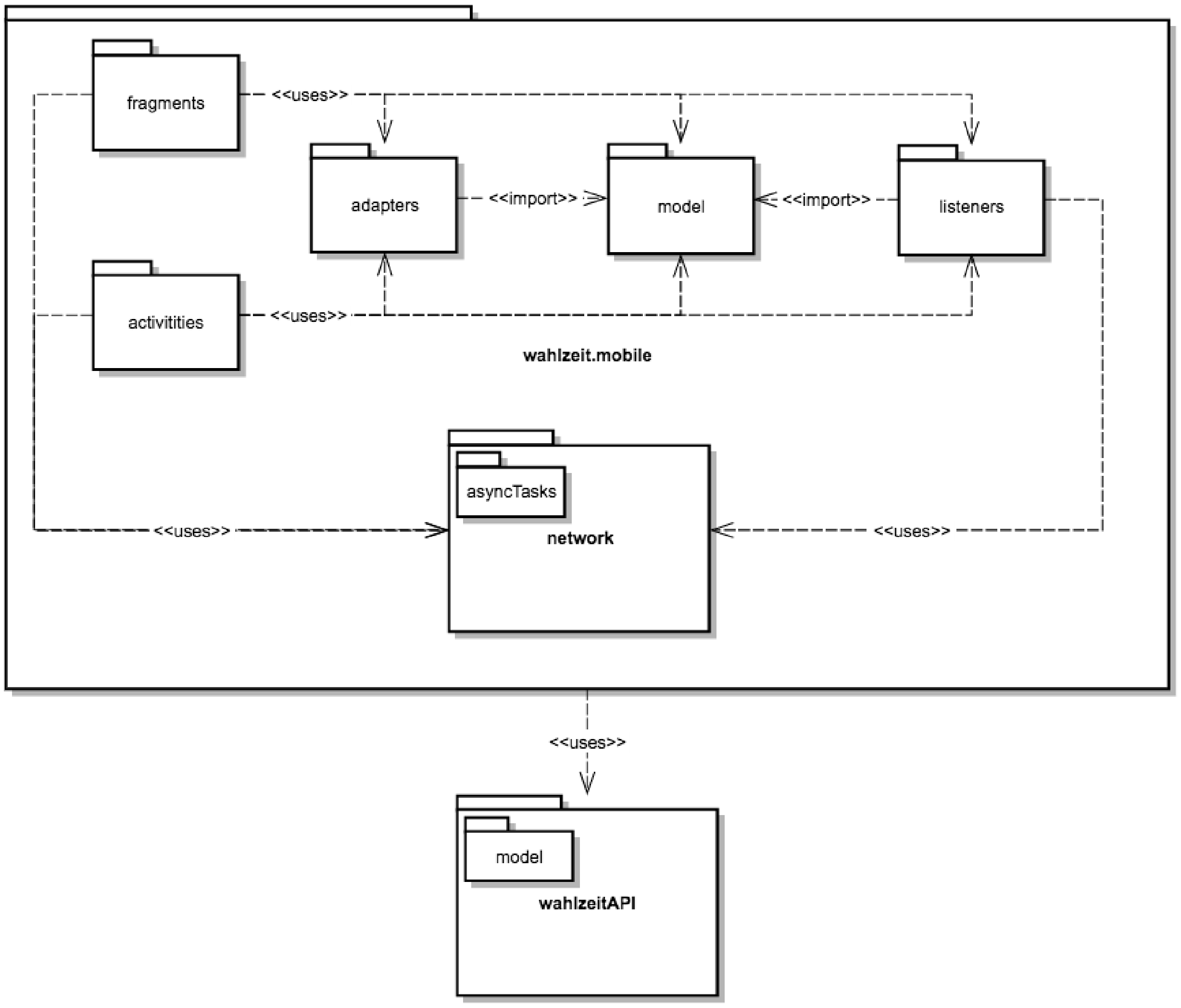
| # | I want to | So that I can | To be verified by the following acceptance test |
|---|---|---|---|
| | | Login | |
| US-1 Pr. H ☑ | Choose an email account from Google | Provide the application a mean to verify my identity | T-1: The app shows a list of all available emails and let the uses choose which email to use. |
| US-2 Pr. H ☑ | Sign in as user or guest | Choose whether I want the application to know my identity or not | T-2: The login screen has two buttons (user-guest) the user can press to sign in, after having chosen an account. |
| US-3 Pr. M ☑ | Save my credentials and language preferences | Avoid signing in every time I open the application | T-3: The app implicitly saves the credentials and language preferences, and retrieves them when launched. |
| US-4 Pr. L ☐ | Sign in using another account instead (Twitter, Facebook) | Easily signing in indirectly | T-4: The login screen has a button to allow signing in via fb or twitter, etc. |
| | | Show | |
| US-5 Pr. H ☑ | View each photo once at a time | Evaluate each photo individually | T-5: The show fragment is made of a **card stack** that allows performing action at one photo at a time. |
| US-6 Pr. H ☑ | Rate a photo from the scale of 1 to 10 | Express my opinion about a photo | T-6: A rating bar is displayed underneath each photo. |
| US-7 Pr. H ☑ | Skip a photo | Evaluate it later | T-7: A card might be skipped by dragging it in any direction. |
| US-8 Pr. H ☑ | Flag a photo | Express myself in case the photo is innapropriate (copyright violation, offensive etc.) | T-8: The card is clickable and launched a popup to choose an action from.<br>T-9: In case flagging action is chosen a new view is displayed where the user can enter more details(explanation, reason, address). |
| US-9 Pr. H ☑ | Mail a friend or the owner about the photo | Directly contact my friends or the owner in private | T-10: In case "mail a friend" or contact the "owner" is chosen when the photo is clicked, the app redirects to the tell fragment. |
| US-10 Pr.H ☑ | Paginate through the whole photos' collection | | T-11: The application retrieves 10 photos at a time and reloads after the user has reacted on those 10 photos. If |

| | | | |
|---|---|---|---|
| | | | the collection is empty a relevant message is displayed. |
| US-11 Pr. H ☑ | Filter photos according to their tags | Evaluate photos of a particular category | T-12: The show fragment has a text box which regulates the photos that are being displayed. |
| **Tell** | | | |
| US-12 Pr. H ☑ | Be able to send emails | Share a photo or embrace users to use the application | T-13: The tell view consist of a optional image field, and textboxes (address, subject, message).<br>T-14: A tell button launches the preffered email service, conveying the relevant information for the email as well as the attachment of the picture. |
| **Home** | | | |
| US-13 Pr. H ☑ | See all my uploaded photos and profile information | So that I can administrate my uploads | T-15: A view displays information about the actual user (user photo, mail, name, gender).<br>T-16: The home view includes a list of all photos that belong to the user and displays the information about each photo(praise, status, creaton time, tags). |
| US-14 Pr.H ☑ | Perform actions to administrate each photo | | T-17: The user is able to click each photo in the list and a popup dialog with the relevant actions is being launched (edit, tell, select, delete). |
| US-15 Pr. H ☑ | Edit a photo | Correct the photo if something is wrong or outdater, or provide additional information. | T-18: A new view is launched allowing the user to edit the tags and visibility of the photo. |
| US-16 Pr.H ☑ | Delete a photo | | T-19: The photo status changes to deleted and the is no more visible by other users |
| US-17 Pr.H ☑ | Select my profile picture | | T-20: The select action replaces the profile picture of the user. |
| **Profile** | | | |
| US-18 Pr.H ☑ | See my profile details | Administrate and edit them | T-21: The home view contains information about the user's name, gander, language, email and notify about praise, which the user can edit.<br>T-22: An edit button is used to update the settings throughout the application and in the server side. |
| **Upload** | | | |
| US-19 Pr.H ☑ | Have an upload-like view | Upload photo entities | T-23: The upload view should contain an image view for the photo to be uploaded and editable text view to allow user to write tags. |

| | | | |
|---|---|---|---|
| US-20 Pr.H ☑ | Choose images from the gallery | So that I can share my gallery photos using Wahlzeit | T-24: The application allows the user to navigate through the gallery and chose a photo. Thereafter the photo is displayed in the upload view. |
| US-21 Pr.M ☑ | Capture a photo from my camera. | Instantly add it to Wahlzeit's collection | T-25: The application launches the camera and allows the user to capture a picture which is thereafter displayed in the upload view. |
| US-22 Pr.H ☑ | Upload the photo entity. | Save the photo permanently and share it with other users. | T-26: An upload button handles the filled out upload form and upload the entity to the backend. |
| | | Moderate | |
| US-23 Pr.H ☑ | See all flagged photos, if I have been granted moderator rights. | Manage each photo individually | T-27: The moderate view should be comprised by a list view that displays all flagged photos and their relevant information (explanation, reason, flagger, tags, description). |
| US-24 Pr.H ☑ | Perform administrative actions on each flagged photo | Keep the content of the photo collection clean and fair | T-28: Each photo in the list is clickable and allows a popup, allows performing actions (moderate, unflag). |
| | | Miscellaneous | |
| US-25 Pr.H ☑ | Sign out and return to the login screen | Sign in again using a different account | T-29: The app offers an option to sign out |
| US-26 Pr.H ☑ | Switch language options between | Get the application's context in English and German | T-30: The app offers an option to switch language from any point |

# Appendix B Android client's package structure

A package diagram, which depicts the packages located in the java directory of the Android project and their relationship.



## References

Abdaldhem & Fuhrer. (2009). Web Services Technologies: State of the Art, 43. Retrieved from: http://diuf.unifr.ch/drupal/softeng/sites/diuf.unifr.ch.drupal.softeng/files/file/publications/internal/WP09-04.pdf

Android. (2016a). API-Guides: App Manifest. Retrieved from: https://developer.android.com/guide/topics/manifest/manifest-intro.html

Android. (2016b). API-Guides: Intents and Intent Filters. Retrieved from: https://developer.android.com/guide/components/intents-filters.html

Android. (2016c). API-Guides: Broadcast Receiver. Retrieved from: https://developer.android.com/reference/android/content/BroadcastReceiver.html

Android. (2016d). API-Guides: Activities. Retrieved from: https://developer.android.com/guide/components/activities.html

Android. (2016e). API-Guides: Fragments. Retrieved from: https://developer.android.com/guide/components/fragments.html

Android. (2016f). API-Guides: Tasks and Back Stack. Retrieved from: https://developer.android.com/guide/components/tasks-and-back-stack.html

Android. (2016g). Android APIs: Asyncronous Task. Retrieved from: https://developer.android.com/reference/android/os/AsyncTask.html

Android. (2016h). API-Guides: App Resources. Retrieved from: https://developer.android.com/guide/topics/resources/index.html

Android. (2016i). API-Guides: User Interface. Retrieved from:https://developer.android.com/guide/topics/ui/index.html

Android. (2016j). Training: Creating a Navigation Drawer. Retrieved from: https://developer.android.com/training/implementing-navigation/nav-drawer.html

Beal. (2006). API- Application Program Interface. Retreived from: http://www.webopedia.com/TERM/A/API.html

Bouguettaya, Sheng & Daniel. (2013). Web services foundations. *Web Services Foundations*, *9781461475*, 1–739. http://doi.org/10.1007/978-1-4614-7518-7

Brähler (2010). Analysis of the Android Architecture. Os.Ibds.Kit.Edu, 52. Retrieved from: http://os.ibds.kit.edu/downloads/sa_2010_braehler-stefan_android-architecture.pdf

Campos, Kulesza, Coelho, Bonifácio, & Mariano (2015). Unveiling the Architecture and Design of Android Applications - An Exploratory Study. *Proceedings of the 17th International Conference on Enterprise Information Systems*, 201–211. http://doi.org/10.5220/0005398902010211

Comscore. (2014). The U.S. Mobile App Report. Retreived from:

http://www.comscore.com/Insights/Presentations-and-Whitepapers/2014/The-US-Mobile-App-Report

Coulouris, Dollimore, & Kindberg (2012). *Distributed Systems: Concepts and Design*. *Computer* (Vol. 4). Retrieved from http://www.amazon.com/dp/0321263545

Dalmasso, Datta, Bonnet, & Nikaein (2013). Survey, comparison and evaluation of cross platform mobile application development tools. *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, 323–328. http://doi.org/10.1109/IWCMC.2013.6583580

Elgin. (2005). Google Buys Android for Its Mobile Arsenal. Retrieved from: http://www.webcitation.org/5wk7sIvVb

Ferris & Farrell (2003). What are Web Services? *Communications of the ACM*, *46*(6), 31.

Fielding (2000). Architectural Styles and the Design of Network-based Software Architectures. *Building*, *54*, 162. http://doi.org/10.1.1.91.2433

Gamma, E. (1995). Design patterns: Elements of reusable object-oriented software. Reading, Mass: Addison-Wesley.

Google. (2016a). Overview of Cloud Endpoints. Retrieved from: https://cloud.google.com/appengine/docs/java/endpoints/

Google. (2016b). Endpoint Annotations and Syntax. Retrieved from: https://cloud.google.com/appengine/docs/java/endpoints/annotations

Google. (2016c). Generating Libraries and Discovery Documents with the Endpoints tool. Retrieved from: https://cloud.google.com/appengine/docs/python/endpoints/endpoints_tool

Google. (2016d). Adding authorization to the API Backend. Retrieved from: https://cloud.google.com/appengine/docs/java/endpoints/add-authorization-backend#understanding_the_process https://developers.google.com/+/web/api/rest/oauth#incremental-auth

Google. (2016e). Using OAuth 2.0 to Access Google APIs. Retrieved from: https://developers.google.com/identity/protocols/OAuth2

Hadley & Sandoz. (2009). JAX-RS: Java API for RESTful WebServices (version 1.1) Retrieved from: https://jcp.org/en/jsr/detail?id=311

Hahmann. (2015). Migrating Code Into The Cloud. Retrieved from: https://osr.cs.fau.de/wp-content/uploads/2015/10/hahmann_2015_arbeit.pdf

IBM. (2000). Web Services Architecture Overview. Retrieved from: http://www.ibm.com/developerworks/library/w-ovr/

IBM. (2011). Design and Implement RESTful web services with Rational Software Architect. Retrieved from: https://www.ibm.com/developerworks/rational/library/design-implement-restful-web-services/

IETF. (1999) Hypertext Transfer Protocol. Retrieved from: https://tools.ietf.org/html/rfc2616

IETF. (2011). The WebSocket Protocol, p.10. Retrieved from: https://tools.ietf.org/html/rfc6455#section-1.7

JSON. (2016). Introducing JSON. Retrieved from: http://www.json.org/

Lucchi, Millot & Elfers (2008). Resource Oriented Architecture and REST. *Assessment of Impact and Advantages on INSPIRE Ispra European Communities*, 16. http://doi.org/10.2788/80035

Microsoft. (2016). Web service benefits. Retrieved from: https://msdn.microsoft.com/en-us/library/cc508708.aspx

Neil. (2014). Mobile Design Pattern Gallery, Second Edition. O' Reilly

Neumann (2012). Entwicklung einer Android-App zur Erkennung und Übersetzung von Worten in Kamerabildern. Retrieved from:

Nurseitov, N., Paulson, M., Reynolds, R., & Izurieta, C. (2009). Comparison of JSON and XML Data Interchange Formats: A Case Study. *Scenario*, *59715*, 157–162. Retrieved from http://www.cs.montana.edu/izurieta/pubs/caine2009.pdf

OAuth 2.0. (2016). OAuth 2.0. Retrieved from: http://oauth.net/2/

Open Handset Alliance. (2016) Android Overview. Retrieved from: http://www.openhandsetalliance.com/android_overview.html

Oracle. (2016). Using Sessions and Session Persistence in Web Applications. Retrieved from: https://docs.oracle.com/cd/E13222_01/wls/docs81/ webapp/sessions.html

Papazoglou. (2008). Web Services: Principles and Technology. Retrieved from: http://icsoc2008.servtech.info/summer_school/soa_analysis.pdf

Petcu, Craciun & Rak (2011). Towards a Cross Platform Cloud API - Components for Cloud Federation. *Closer*, 166–169. Retrieved from: http://dblp.uni-trier.de/db/conf/closer/closer2011.html#PetcuCR11

Pimentel & Nickerson (2012). Communicating and displaying real-time data with WebSocket. IEEE Internet Computing, 16(4), 45–53. http://doi.org/10.1109/MIC.2012.64

Stylos & Myers (2007). Mapping the space of API design decisions. Proceedings - IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2007, 50–57. http://doi.org/10.1109/VLHCC.2007.36

Subsplash (2015). 6 Benefits of going mobile. Retreived from: http://www.subsplash.com/blog/category/6-benefits-of-going-mobile

Sumaray & Makki (2012). A comparison of data serialization formats for optimal efficiency on a mobile platform. *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication - ICUIMC '12*, (0851912), 1. http://doi.org/10.1145/2184751.2184810

Tilkov (2011). Rest und HTTP. DPunkt Verlag.

Wagh & Thool (2012). A Comparative Study of SOAP Vs REST Web Services Provisioning Techniques for Mobile Host. *Journal of Information Engineering and Applications*, *2*(5), 12–16. Retrieved from http://www.iiste.org/Journals/index.php/JIEA/article/view/2063

Wasserman (2010). Software Engineering Issues for Mobile Application Development. *ACM Transactions on Information Systems, 1–4*. http://doi.org/10.1145/1882362.1882443

Yildirim (2014). Distributed Systems. Retrieved from http://pages.cs.wisc.edu/~remzi/OSTEP/dist-intro.pdf

W3C. (1999a). Hypertext Tranfer Protocol 1.1, 9. Method Definitions. Retrieved from: www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

W3C. (1999b). The Content-Type Header Field. Retrieved from: https://www.w3.org/Protocols/rfc1341/4_Content-Type.html

W3C. (1999c). Content Negotiation. Retrieved from https://www.w3.org/Protocols/rfc2616/rfc2616-sec12.html

W3C. (2001). Web Service Description Language 1.1. Retrieved from: https://www.w3.org/TR/wsdl

W3C. (2007). SOAP Version 1.2 Part 1: Messaging Framework. Retrieved from: https://www.w3.org/TR/soap12

W3C. (2008). Extensible Markup Language (XML) 1.0 (Fifth Edition). Retrieved from: https://www.w3.org/TR/REC-xml/

Zahir, Malhotra & Abdelkamel (2005). Toward the Right Communication Protocol for Web Services. *International Journal of Web Services Research, p.19*.