Friedrich-Alexander University Erlangen-Nürnberg

Faculty of Engineering, Department Computer Science

LUKAS HAHMANN
MASTER THESIS

MIGRATING CODE INTO THE CLOUD

MIGRATING A SERVER APPLICATION TO GOOGLE APP ENGINE

Submitted on 28 August 2015

Supervisor: Prof. Dr. Dirk Riehle, M.B.A.
Professorship of Open Source Software
Faculty of Engineering, Department Computer Science
Friedrich-Alexander University Erlangen-Nürnberg

Versicherung

Erlangen, 28 August 2015

License

Erlangen, 28 August 2015

Abstract

Wahlzeit is an open source Java web application that gives you a solid software base to set up your own photo rating website. Besides that, it is used during the lecture Advanced Design and Programming to teach agile methods and open source software development.

In addition to the software, we want to provide Wahlzeit with the according infrastructure for your photo rating website. Google App Engine (GAE) is selected in this thesis as the appropriate cloud service for our needs. Therefore we migrate Wahlzeit from a Tomcat Server application running on our own hardware to GAE that provides Google infrastructure without costs for smaller web projects. To run on GAE we adjusted several parts of Wahlzeit, like the persistence layer, the session management, and the project deployment. Furthermore we created a detailed design, to make Wahlzeit a RESTful service that could serve several clients.

Acknowledgments

In my daily work I have been blessed with a friendly and cheerful group of fellow students. Especially, I want to thank Johannes Bayerl, for accompanying and supporting me during the whole Master, and Tobias Fertig for his useful input and support for this thesis.

I would like to express my deep and sincere gratitude to Christian A. Hochmuth for improving my scientific English, my typesetting, and the demand on me and my work.

My sincere thanks to Prof. Riehle for the idea of this thesis, his suggestions and feedback.

Although computer science is not her favorite topic, my girlfriend was a great supporter for my studying and the main reason why I came to Erlangen.

Last but not least, I owe my loving thanks to my parents for teaching me the value of education. Without their encouragement and support my studying would have been impossible.

Contents

List of Abbreviations

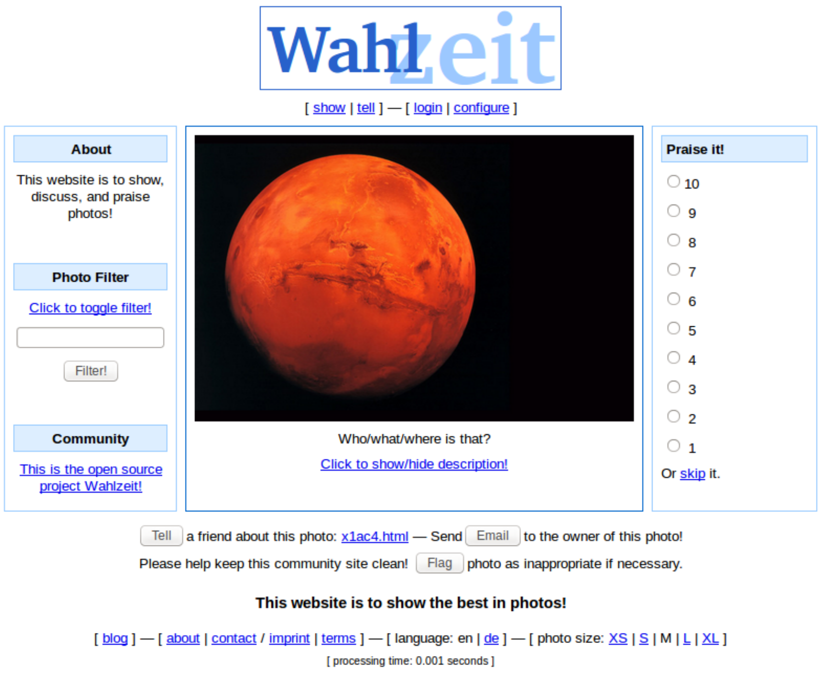
ADAP	Advanced Design and Programming
ALM	Application Lifecycle Management
API	Application Programming Interface
BLOB	Binary Large Object
CSS	Cascading Style Sheets
EC2	Amazon Elastic Compute Cloud
EJB	Enterprise JavaBeans
GAE	Google App Engine
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IDE	Integrated Development Environment
JRE	Java Runtime Environment
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
NIST	National Institute of Standards and Technology
OVF	Open Virtualization Format
PaaS	Platform as a Service
PC	Personal Computer
REST	Representational State Transfer

RRZE	Regional Computing Center Erlangen (German: Regionales Rechenzentrum Erlangen)
SaaS	Software as a Service
SDK	Software Development Kit
SLOC	Source Lines of Code
SQL	Structured Query Language
UI	user interface
URI	Uniform Resource Identifier
WAR	Web Application Archive
WLAN	Wireless Local Area Network

List of Figures

1 Introduction

Wahlzeit is an open source web application. It offers you the possibility to upload and praise photos on a 1–10 scale (see Figure 1.1). To upload a photo, you have to login. In a personal area, you can manage your photos. Wahlzeit is a yet complete Java web application.



Wahlzeit

[show | tell] — [login | configure]

About

This website is to show, discuss, and praise photos!

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!

Praise it!

10
9
8
7
6
5
4
3
2
1

Or skip it.

Tell a friend about this photo: x1ac4.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!

[blog] — [about | contact / imprint | terms] — [language: en | de] — [photo size: XS | S | M | L | XL]

[processing time: 0.001 seconds]

Figure 1.1: Main page of Wahlzeit showing a random photo

The main intend of Wahlzeit is teaching agile methods and open source software development in the course Advanced Design and Programming (ADAP) offered by the Professorship of Open Source Software at the Friedrich-Alexander University Erlangen-Nürnberg. During the lessons students fork Wahlzeit on GitHub[1] to analyze and extend the software with own functionality. The so extended version of Wahlzeit can either be deployed on their own computer via Tomcat Server[2] or on the department server where their Wahlzeit version is accessible in the Internet.

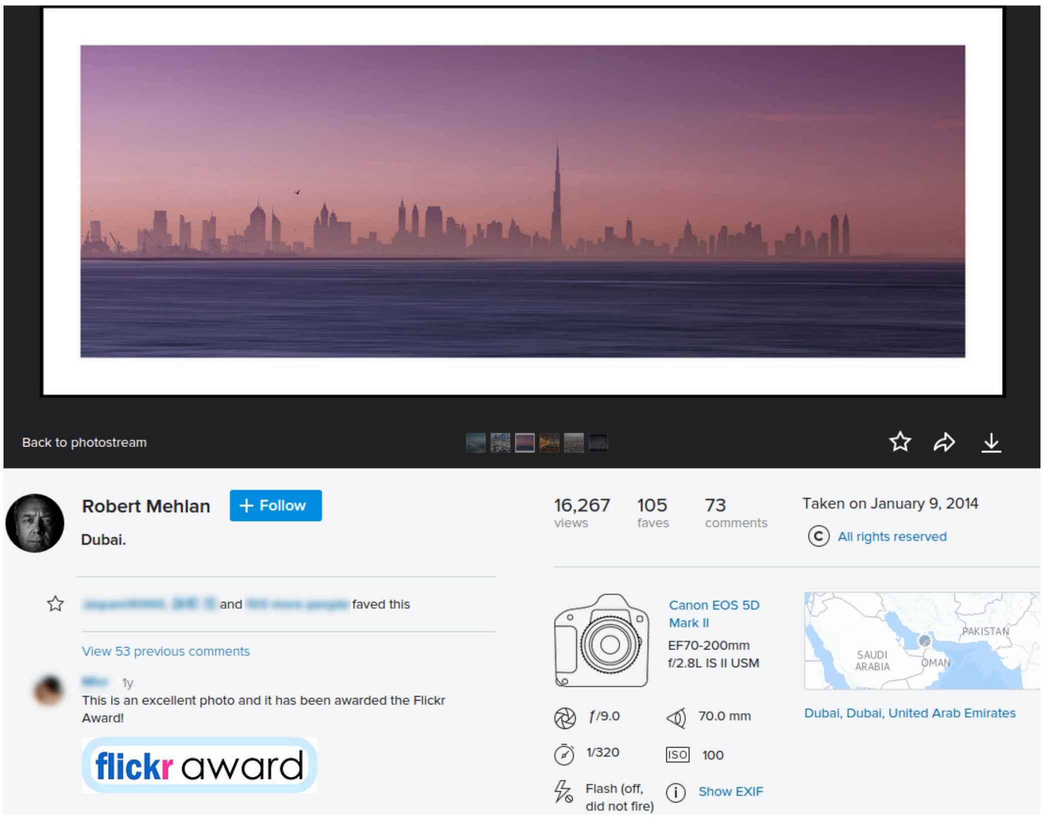
The long-term vision of Wahlzeit is, to deliver it as an own product that you can rent and adjust to your needs. Then you can build up your own photo rating page based on Wahlzeit. In the following we elucidate the motivation behind this vision.

When surfing through the Internet lots of subject areas with very different photos and meta data exist. Most of them are presented on self-build websites that are often technically outdated. This shows that the corresponding website owners do not want to use social networks or predefined homepage building sets for sharing and rating their photos because of special technical needs or personal reasons. Whatever their motivation might be, they decided to create own websites dedicated for their subject area and their photos. This decision is comprehensible when you want special additional photo informations or functions social networks or homepage building sets do not offer. Lets list two examples:

First, **inhabitants of a city** want to see photos of their city and their near environment. Special information of such photos might be the camera and its settings, the location and the name of the photographer. Flickr by this time allows entering such information, but it does not offer a rating function (see Figure 1.2). If you want such a function, adjusting Flickr is not that easy. There exist more subject areas that have very special photo information.

Astronomers are the second example, they work with photos of outer space objects. For them the spectral range of the photo and the position of the outer space object are important informations. As common (earth-) photos are taken in the visible spectral range, nearly none of the social networks or homepage building sets offers adding such information for photos. An even more complicated task is specifying outer space coordinates, because they can be expressed by lots of coordinate systems (Karttunen, Kröger, Oja, Poutanen & Donner, 2007; Majewski, 2014). There even exist libraries[3] to calculate coordinates from one of the many coordinate systems to another. Hence getting the coordinates of all photos only in the favorite coordinate

[1]see https://github.com/dirkriehle/wahlzeit
[2]see https://tomcat.apache.org
[3]see http://aa.usno.navy.mil/software/novas/novas_info.php



Robert Mehlan + Follow

Dubai.

16,267 views 105 faves 73 comments

Taken on January 9, 2014

All rights reserved

[redacted] and [redacted] faved this

View 53 previous comments

[redacted] 1y

This is an excellent photo and it has been awarded the Flickr Award!

flickr award

Canon EOS 5D Mark II

EF70-200mm f/2.8L IS II USM

f/9.0 70.0 mm

1/320 ISO 100

Flash (off, did not fire) Show EXIF

Dubai, Dubai, United Arab Emirates

Figure 1.2: Flickr shows the name of the photographer, the camera with its settings, and the location of the photo.

system, no matter how they have been entered, would be a nice feature for an astronomy website.

Efficiently supporting those people who decide to build an own photo rating webpage is the second target of Wahlzeit. The vision is to deliver Wahlzeit as a base system including the according infrastructure, so you can directly start adapting it to your needs. This would drastically reduce your time to build the photo rating page you want. But there is a lot to do until this vision can become reality.

For this thesis we use the following conventions. **Important words** or **phrases** are marked bold. *Paths* or *names of files* are written italic. For `Sourcecode` we use courier font. The current version of Wahlzeit running on the own infrastructure is Wahlzeit 1.2, the new version running at Google App Engine (GAE) is Wahlzeit 2.0.

With this thesis Wahlzeit will come a step closer to its vision. It is developed from the current version Wahlzeit 1.2 to the new major version Wahlzeit 2.0. Besides that we give an outlook for further development. Chapter 2 determines the requirements, and Chapter 3 gives an overview of the related literature. Chapter 4 describes the design of the changes done to Wahlzeit in order to get closer to its vision, while Chapter 5 summarizes the actual implementation effort. The last chapter, summarizes the activities of this thesis and gives an overview of the next steps towards the Wahlzeit vision.

2 Requirements

As already described in Chapter 1, there is much to do, in order to achieve the Wahlzeit vision. In this chapter the current usage of Wahlzeit during the lesson is described. We want to improve this situation, as well as doing some steps in the direction of the vision. The tasks for the latter are formulated in Section 2.2.

2.1 Technical setup of Wahlzeit 1.2

To understand the necessity of this thesis for the lecture, the current technical setup of Wahlzeit 1.2 has to be described. It is divided into the following 4 activities that arise during the ADAP lessons:

1. students activity: deploy Wahlzeit on their own PC,
2. students activity: deploy Wahlzeit online,
3. students activity: debug Wahlzeit online, and
4. staff activity: maintain Wahlzeit server.

These activities are explained in detail in the following subsections.

2.1.1 Students activity: deploy Wahlzeit on their own PC

Activity 1 involves all activities for you starting with an unprepared computer until you can run Wahlzeit on your local machine. Therefore the following setup tasks are necessary:

1. install git (see `https://git-scm.com`),
2. clone Wahlzeit from GitHub ,

3. open it with your Integrated Development Environment (IDE) and resolve the dependencies,
4. install PostgreSQL (see `http://www.postgresql.org`),
5. create a PostgreSQL user for Wahlzeit and a database,
6. download Tomcat Server and start a server, and
7. run Wahlzeit on Tomcat Server.

Experience from ADAP has shown that these tasks took between 1 hour for experienced developer facing Wahlzeit the first time until 3 or more hours for students that are not very familiar with either git, PostgreSQL, Tomcat, or their IDE. These setup tasks arise every year for twenty or more students who attend ADAP.

When the setup tasks are done, you can start exploring the Wahlzeit source code and extend it. After you have extended and tested Wahlzeit on your local machine, deploy it on the department server to make your Wahlzeit version accessible online. This activity is explained in the following subsection.

2.1.2 Students activity: deploy Wahlzeit online

The main intent of ADAP is to teach good object-oriented design and programming. Wahlzeit helps to achieve both, to show a good object-oriented design, and to offer an extensible platform where students can show what they have learned. To present the results of your adjustments deploy Wahlzeit on the department server that is accessible online (Activity 2). A basic service with a user interface is hosted on that server. There you log in to upload and deploy the Web Application Archive (WAR) of Wahlzeit. Therefore execute the following tasks:

1. Adjust the hard coded Structured Query Language (SQL) user name and password in Wahlzeit to the credentials for the database of the department server.
2. Export Wahlzeit WAR.
3. Open the web interface of the server to upload your Wahlzeit WAR.
4. Deploy it.

In case you have adjusted the credentials of your local SQL database to the one on the department server, the first task can be skipped. The other steps have to be done for each version you want to publish, which are at least ten times per student per semester.

As Wahlzeit might behave a little different when being deployed on the department server, you need to debug Wahlzeit in case of an error. How this is done is explained in the next subsection.

2.1.3 Students activity: debug Wahlzeit online

When an error arises on the department server that did not arise on your local machine you have to do Activity 3, debug Wahlzeit on the server. As a student you have the following two possibilities:

- Analyze error messages generated by Wahlzeit for the user.
- Download and analyze log files from the server.

The user of Wahlzeit is usually not the developer hence the error messages shown to the user in the user interface (UI) are either errors done by the user itself, like entering the wrong password, or fatal errors that prevent Wahlzeit from a regular operation, like inaccessible database. But those messages are usually not helpful to find bugs in your adjustments of Wahlzeit which might be, adding additional attributes to photos. Therefore only the second possibility remains, analyzing the log files.

About twenty students operate on the same department server. Therefore it is quite likely that this server might quit its service during the semester. In such a case the department stuff has to revive the server. This and all the other department maintaining activities are explained in the following subsection.

2.1.4 Staff activity: maintain Wahlzeit server

To deploy Wahlzeit online a dedicated server is maintained by the department stuff. This comes along with the following initial and ongoing maintaining tasks for Activity 4:

1. Rent a server dedicated for Wahlzeit.
2. Install custom Wahlzeit WAR upload service and PostgreSQL.

3. Install Tomcat Server.
4. Create an upload service user account for each student.
5. Create SQL user and database for each student.
6. Monitor server and repair it in case of a savage Wahlzeit instance.

The Tasks 1–3 are initial tasks that usually have to be done only once. The Tasks 4 and 5 have to be done every semester when new students want to deploy Wahlzeit online. Task 6 has to be done during the semester, mostly on demand of the students that could not reach the server.

Beside using Wahlzeit as learning object during lectures, we have a vision to deliver Wahlzeit with the according infrastructure to users that want to build their own photo rating page. The goals to come closer to this vision are explained in the following section.

2.2 Goals

To bring Wahlzeit a step closer to its vision and to reduce configuration overhead that arises during the lecture the following goal is defined:

Mandatory Goal 1: migrate Wahlzeit into the cloud.

Jamshidi, Ahmad and Pahl (2013, p. 150) list general motivations for migrating an application into the cloud: *cost saving*, *scalability*, *efficient resource utilization*, *elasticity to fluctuation*, *interoperability*, and *maintainability*. Cost saving and maintainability are relevant for Wahlzeit, too. The concrete motivation to migrate Wahlzeit into the cloud for teaching during ADAP is to reduce effort for the activities mentioned in Section 2.1:

1. Reduce initial effort to deploy Wahlzeit on students own PC.
2. Reduce effort to deploy Wahlzeit online.
3. Improve debugging possibilities for Wahlzeit online.
4. Reduce maintaining effort for the department staff.

Furthermore Wahlzeit in the cloud would bring it a step closer to its vision (see Chapter 1). Currently Wahlzeit is only a software that has to run on a client server with operating system, database, and other software. If you do not own such infrastructure or do not want to pay for, Wahlzeit will not be your choice to create a custom photo rating page. But if Wahlzeit would run in the cloud, it would no longer be necessary to maintain and pay a server.

For Mandatory Goal 1 to be achieved in the intended way, Wahlzeit in the cloud should furthermore fulfill the following requirements:

5. The infrastructure should be costless for the usage during the lecture.
6. Wahlzeit in the cloud should have at least all the existing functionality.
7. Performance in the cloud should not be significantly slower.

This results in 7 requirements that have to be fulfilled to reach Goal 1. Besides the mandatory goal, there exist two optional goals that can be targeted afterwards:

Optional Goal 1: analyze the application evolution (building, testing, deploying) of a cloud application.

Some cloud services offer a close interaction with source code repositories and allow to monitor and update automatically in case of updated repository (Google, 2015l). The applicability of this process for Wahlzeit should be analyzed.

Furthermore there exists another optional goal that would help all students who want to adjust the UI of Wahlzeit:

Optional Goal 2: UI renovation.

The current Wahlzeit UI is generated out of predefined Hypertext Markup Language (HTML) files. They are combined by Java code and extended with site specific functions. The HTML files are language dependent (currently German and English) which makes changes to the UI more sophisticated than technically necessary. Furthermore adding a new language generates more effort than just translating some text files with the language specific content. A new UI that is created automatically out of Java code, would reduce the effort for adjustments to Wahlzeit and would make the project more attractive for developers.

Mandatory Goal 2: analyze multi-tenant ability for Wahlzeit.

If Wahlzeit is migrated to the cloud it could be offered as an own product. What stays for the user is to register at the cloud service, and to setup an instance of Wahlzeit. To avoid that effort for the user, the cloud service could be rented by the Wahlzeit developers. If an user wants an own Wahlzeit flavor, a new tenant could be created by the developers. They give the credentials to the user. You get a running instance that you can adjust to your needs. If making Wahlzeit a multi-tenant software is a suitable goal for Wahlzeit, should be analyzed in this thesis.

Optional Goal 3: analyze extending Wahlzeit to a framework that can be adjusted by plug-ins to theme specific flavors.

When Wahlzeit runs in the cloud, adjusting it on the fly with custom plug-ins to get your own flavor sounds like a goal worth reaching. To achieve that, it would be necessary to extract a Wahlzeit core that can be adjusted by plug-ins. In this case only one Wahlzeit core would run on a GAE instance, that serves several theme specific sites. Extracting such a core causes new questions: How to secure plug-ins from each other? How to allocate the resources to the tenants? How to encapsulate a plug-in to avoid that it is attacking the Wahlzeit core? Those question indicate that this will get a greater task, that need its own special research.

The basic literature research that is relevant for evaluating those goals is presented in the next chapter.

3 Related work

This chapter gives an overview about the relevant literature and examines the necessary facts, that lead to the design, described in Chapter 4. We start with an introduction into cloud computing.

3.1 Introduction into cloud computing

Cloud computing is a highly discussed buzzword in the recent years (Höfer & Karagiannis, 2011; Tran, Keung, Liu & Fekete, 2011). The Google search trend for *cloud computing* and *cloud service* (see Figure 3.1) shows that the topic of cloud computing arises in the end of 2007 (cloud service also contains meteorologic interests hence it started a little earlier).

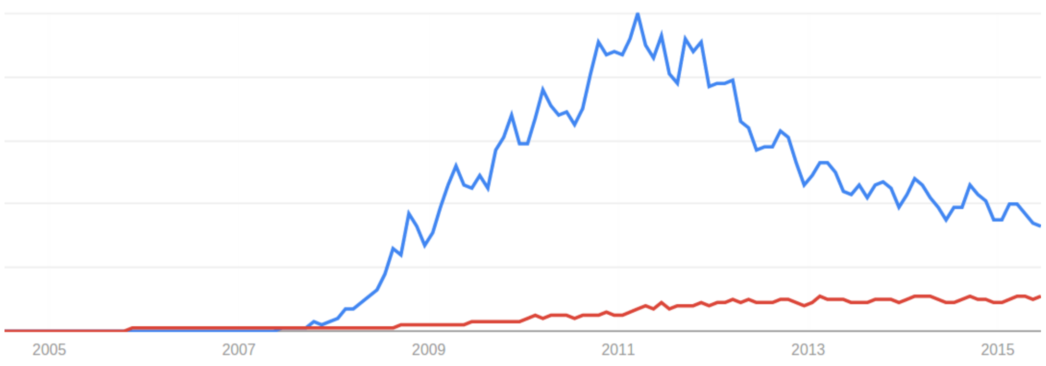


Figure 3.1: Google search trend for the buzzwords *cloud computing* (blue) and *cloud service* (red) since 2005.

In one of the first publications about cloud computing, Hayes (2008, p. 9) describes it as the second incarnation of the service bureaus and time sharing systems that provided access to a mainframe computer for everyone back in the 1960s and 1970s. Although there are some similarities between cloud services and the then service bureaus, a cloud is not a single mainframe computer. Buyya, Yeo, Venugopal, Broberg and Brandic (2009, p. 601) aggregate a definition of the term Cloud out of others:

> "A Cloud is a type of parallel and distributed system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resource(s) based on service-level agreements established through negotiation between the service provider and consumers."

Besides that the National Institute of Standards and Technology (NIST) defines *Cloud Computing* as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (networks, server, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" (Mell & Grance, 2011, p. 6). This describes the process of using the cloud. There are 5 essential characteristics for *cloud computing* (Mell & Grance, 2011, p. 6):

- on-demand self-service,
- broad network access,
- resource pooling,
- rapid elasticity, and
- measured service.

Buyya et al. (2009, p. 601) list additional but not equal terms: *Grid* and *Cluster*. While Clusters denote huge amounts of high end computers mostly used for scientific computing, grids are smaller collections of commodity computers for nearly all purposes. Both Clusters and Grids operate with one operating system, while the Cloud is build on virtual machines with different operating systems. This offers to secure the cloud by the accesses to the virtual machines. Grid and Cluster have to draw on a lower support for privacy. For this theses we only work with the Cloud as Buyya et al. (2009, p. 601) define it.

In the recent years, both, the interest in *cloud computing*, and the amount of cloud computing services has been growing quickly. As it is quite a new field within computer science several classification approaches for cloud services exist (Höfer & Karagiannis, 2011). Nearly all of them divide cloud services into the following 3 categories, which are also defined by the NIST (Mell & Grance, 2011; Höfer & Karagiannis, 2011; Beimborn, Miletzki & Wenzel, 2011):

- *Infrastructure as a Service (IaaS)*,
- *Platform as a Service (PaaS)*, and
- *Software as a Service (SaaS)*.

These categories are described as follows.

IaaS: The core idea of Infrastructure as a Service is to sell computing power and storage. As customer you get (mostly virtualized) server instances that you pay per use of resources. You can rent them as long as you want. On these virtual servers you can deploy own software. Although you have much freedom with this kind of cloud service, you have to maintain all that custom software like operating system and middleware. Usually you monitor your service via a web interface. Popular examples of IaaS are Amazon Elastic Compute Cloud (EC2) (Sadiku, Musa & Momoh, 2014), GoGrid, and the Rackspace Cloud (Höfer & Karagiannis, 2011).

PaaS: Platform as a Service abstracts server instances including hardware and operating system. It provides a software platform to the user. Although it includes IaaS, the infrastructure is completely hidden from the user. The typical users of PaaS offerings develop and run own web applications. The higher abstraction compared to IaaS comes at its price: when you use PaaS, you are limited on programming languages, tools, and runtime environments supported by your provider. But on the other side, the provider cares about hardware, operating system and middleware, which makes you more efficient in developing and deploying your application to the cloud. Popular examples of PaaS are Google App Engine, Microsoft Azure, and Force.com (Höfer & Karagiannis, 2011).

SaaS: The third category Software as a Service puts another abstraction layer above PaaS, it provides complete online applications from web mail to business software. The offered software is already created and runs in the cloud. To run this software in the cloud PaaS offerings are a very popular basis. As a user you are limited in your customization to what the corresponding software offers. This reduced customization ability is the price for accessing a finished software without much effort. Popular examples for SaaS offers are Salesforce.com, Appian Anywhere, and the Google Apps like GMail, Calender, and Contacts (Höfer & Karagiannis, 2011).

Besides the three main cloud service models Höfer and Karagiannis (2011) list two new developments. First, games that are completely hosted in the cloud, and second, dedicated services that support smart phones either with additional cloud storage or processing power. Within this thesis the 3 main categories will be relevant.

For all those cloud services, different business models emerged. They are specified by the NIST as 4 deployment models (Mell & Grance, 2011, p. 7):

- *Private cloud*: one organization offers the cloud exclusively to its customers.
- *Community cloud*: dedicated for a an exclusive community of users, managed by several organizations with different concerns.
- *Public cloud*: provided by one ore more organizations for public use.
- *Hybrid cloud*: a combination of at least two distinct deployment models.

In this section we analyzed the types of cloud services that may be a target platform for Wahlzeit. In the following section the migration process described in the literature is analyzed.

3.2 Taxonomy of cloud migration types

Running software in the cloud offers advantages, because the infrastructure is hidden behind a nice interface. But not every software is cloud ready. To run an application that is not originally designed to be deployed into the cloud, migration tasks have to be done. No matter which cloud computing type you choose — IaaS, PaaS, or SaaS — adjustments for the new environment are necessary (Vu & Asal, 2012, p. 270).

Jamshidi et al. (2013, p.150–151) list 4 types of cloud migration:

1. *Replace*: Data and business (logic) tiers are migrated into the cloud, the presentation tier is kept as it is. Therefore adjustments in the

migrated parts of the software as well as in the collaboration of the components are necessary. This type is rarely used.

2. *Partially migrate*: Move some of the software components to the cloud. As for Type 1 this needs adaption for each migrated component and for the collaboration of components.

3. *Migrate the whole application stack*: Run the software as it is in one or more virtual machines in the cloud (see IaaS). This simulates the old environment and is therefore the easiest cloud migration type. But this comes at its price; it exhausts the advantages of the cloud like elasticity very little.

4. *Cloudify*: Adjust the complete application to the new cloud environment (see PaaS or SaaS). This type is similar to Type 2, but it is not limited to some components. It is considered to generate the most effort, but utilized the cloud service and its advantages best.

These types are specified by the degree of migration and their implementation. But this is not the only categorization in the literature. Binz, Leymann and Schumm (2011) list only 3 types which all describe the migration of the complete application into the cloud. The types are specified only by their implementation:

1. *Standardized format migration*: A software running in a standard format container is migrated either between two instances of the same software that support this standard form or between instances of different software. Examples for such runtime container formats are Open Virtualization Format (OVF), Enterprise JavaBeans (EJB), and Java WAR.

2. *Component format migration*: The format of the corresponding component is transformed into another one, e.g., transforming a virtual machine image, or enable execution of scripting language in PaaS offerings.

3. *Holistic migration*: A software out of multiple components is adjusted to become a cloud application whereby each component is migrated and adapted individually to its new environment.

The overview and comparison of both categorizations is shown in Table 3.1. Binz et al. (2011) describe only complete migrations which means that the first two types of Jamshidi et al. (2013) do not find an equivalent. Jamshidi et al. (2013) does not mention the format explicitly which allows to assign

Migration types of Jamshidi et al. (2013)	Migration types of Binz et al. (2011)
Replace	
Partially migrate	
Migrate the whole application stack	Standardized format migration
Migrate the whole application stack	Component format migration
Cloudify	Holistic migration

Table 3.1: Comparison of the different migration types. The first two types mentioned by Jamshidi et al. (2013) describe partial migrations and hence do not have an equivalent described by Binz et al. (2011). Type 3 of Jamshidi et al. (2013) may be mapped to both the standardized and the component format migration. The least type of both sources describe the same, migration and complete adaption of each component to the cloud.

the first two types of Binz et al. (2011) to *Migrate the whole application stack*. Only the last type of both categorization matches.

3.3 Adapting a software for multi tenants

SaaS provider develop or purchase software to adjust and host it for their customers. In recent time those provider tend to sell more and more multi-tenant software. A tenant is an user, an organization, or a company. Multi-tenant software offers each tenant its own service (copy of the software) in a way that you could think to be the only user (Guo, Sun, Huang, Wang & Gao, 2007, p. 551).

Guo et al. (2007) list two types of multi-tenant software, first, *multiple instances*. Each user of multiple instance software has its own instance of software running on shared hardware. In some cases operating system and middleware is shared, too. The second type is *native multi-tenancy*. Such software supports itself several tenants, so minimal overhead is created per tenant. We focus on the latter type, native multi-tenancy. It supports much more tenants than the first variant. This comes along with massive resource sharing (Guo et al., 2007, p. 551).

Multi-tenant software offers several advantages. Guo et al. (2007) list first, an improved profit margin for providers and second, decreased service costs for clients. This win-win situation is a result of simplified administration and provisioning of tenants, e.g., distribution of updates. But it is always a trade-off between cost efficiency and flexibility for the tenants (Walraven, Truyen & Joosen, 2014, p. 670).

Beside the advantages major challenges arise when adjusting software to support multi tenants: new complexities in application development, deployment and management. Multi-tenant software needs strict isolation between each tenant in almost all parts of the architecture design. Furthermore it should offer tenants customizing options for their own service without disturbing others (Guo et al., 2007, p. 551–552). The term *service* describes the part of a multi-tenant software that is sold to a tenant. Therefore it is important to manage and monitor tenants in a fine-grained way (Walraven et al., 2014, p. 670).

There are 5 multi-tenant specific core features of a software (Guo et al., 2007, p. 553):

- security isolation – secure against other tenants,
- performance isolation – limit resource consumption by service levels,
- availability isolation – avoid that tenant faults harm others,
- administration isolation – adjustments limit to the own service, and
- on the fly customization – offer self service for each tenant.

To build and distribute multi-tenant software PaaS offerings are getting more and more attractive as infrastructure. But not every PaaS offering is a good basis for hosting multi-tenant software (Walraven et al., 2014, p. 671). In the eyes of SaaS developer, there are 3 types of PaaS offerings (Walraven et al., 2014, p. 671–675):

1. offer Application Programming Interfaces (APIs) of popular enterprise application servers an their middleware platforms (e.g. Windows Azure and Red Hat Open Shift),
2. hosting of specific cloud application types (e.g. GAE and Giga Spaces' XAP Elastic Application Platform), and
3. driven by meta data with the focus on SaaS applications (e.g. Force.com and WOLF).

In the eyes of a SaaS provider an attractive PaaS offering should support portability between SaaS and on-premise implementation. A software is considered portable when the costs for porting it are less than the cost for redeveloping a new application (Mooney, 1990, p. 59). Portability is achieved by the usage of standard middleware like Java EE or .NET.

Beside portability, creation and management of multi-tenant applications should be supported. This requires data isolation for databases, the ability to automatically specify the tenant for each user request, tenant specific customization and tenant specific application management facility like billing and metering. Last but not least, a good tool support is essential: an IDE, a local development server with database, and a testing framework (Walraven et al., 2014, p. 671–675).

Defining the ideal PaaS offering by the degree of portability of the according software and the support for multi-tenant SaaS there is yet no ideal offering. While Microsoft Azure (Microsoft, 2015) offers a great portability it totally lacks of support for multi-tenant software. Force.com offers great support for multi-tenant software but the according software could be more portable. GAE (Google, 2015e) offers some kind of compromise between the last two offerings: a quite good portability and some support for multi-tenant SaaS (Walraven et al., 2014, p. 712).

In the next chapter the design of Wahlzeit in the cloud is presented, based on the research of the current chapter. Among others the topic multi-tenancy is discussed for Wahlzeit.

4 Design

Vu and Asal (2012, p. 170) list as first important question ″Is it possible as well as practical to migrate a specific application to the cloud?″. To answer this question, first, we have to check if it is possible at all to migrate Wahlzeit. Difficult obstacles for the cloud migration are specific hardware requirements (Vu & Asal, 2012, p. 271). Wahlzeit has no special requirements to processor or storage, and it does not need a special piece of hardware. There are also no specific privacy or security requirements for Wahlzeit which are some weak points of cloud computing (Tran et al., 2011, p. 27). Hence we can answer the first part of the question with ″yes″.

For answering the practicability part of the question, a specific target platform has to be selected and its individual environment and parameters have to be analyzed. In the Sections 4.1 and 4.2 the target cloud service is selected and described in detail. Section 4.3 then answers the second part of the initial question ″is it practical to migrate Wahlzeit into the cloud instead of redeveloping it for the new environment?″

4.1 Selection of a cloud service for Wahlzeit

Wahlzeit currently runs on the department server, which is provided by the Regional Computing Center Erlangen (German: Regionales Rechenzentrum Erlangen) (RRZE). Hence Wahlzeit is already based on a IaaS. To get rid of infrastructure maintenance mentioned in Section 2.1, the next, more abstract cloud service model is PaaS (see Section 3.1). This answers the next question, ″IaaS or PaaS?″ (Vu & Asal, 2012, p. 271)

We need to find a PaaS offering that limits our degree of freedom only where it not hurts and enables us to be more efficient and scalable while having a more abstract layer on top of the infrastructure (see Figure 4.1). This is also

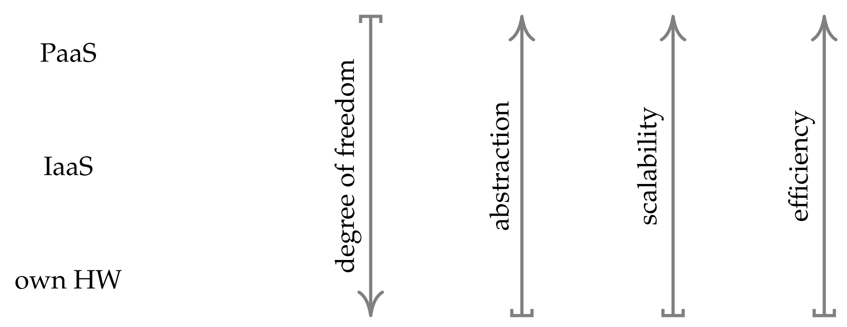


Figure 4.1: Comparison of cloud computing services.

the next step Vu and Asal (2012, p. 271) proposes. As the maintenance of the infrastructure should be outsourced, only a public cloud provider can be selected (see deploy models in Section 3.1). Furthermore Wahlzeit is a Java application, hence only a public cloud provider offering to host Java applications could come into the closer selection. With these parameters in mind 10 popular PaaS offerings have been compared. The result is shown in Table 4.1.

PaaS offering	**Java supported?**	**Free trial** (months)	**Free storage** (GB)
Google App Engine	√	∞	6
OpenShift	√	∞	1
AWS Elastic Beanstalk	√	12	–
Pivotal Cloud Foundry	√	2	–
Microsoft Azure	√	1	–
OutSystems Platform	√	1	–
Jelastic	√	–	–
Oracle Cloud	√	–	–
IBM Bluemix	proprietary	∞	20.5
Salesforce1	proprietary	1	–

Table 4.1: Overview of 10 selected public PaaS offerings hosting Java applications. The importance of the compared properties decreases from left to right.

This list is not an entire market analysis, but it covers a great part of potential candidates that are mentioned in the relevant literature. Google App Engine (Google, 2015e) offers a temporal unlimited trial, among two others: OpenShift and IBM Bluemix (redhat, 2015; IBM, 2015). Some sort of costless trial is necessary because of Requirement 5. Therefore Jelastic and the Oracle Cloud are not suitable candidates, because they do not offer a trial at all (Jelastic, 2015; Oracle, 2015). The offerings Pivotal Cloud Foundry, Microsoft Azure, OutSystems Platform, and Salesforce1 also do not match this requirement, although they offer a free trial (Pivotal, 2015; Microsoft, 2015; OutSystems, 2015; Salesforce.com, 2015). Using a PaaS for ADAP requires at least a free trial for 4 months. Furthermore an unlimited trial is preferable, because students can continue their work after the lecture is over and this costless infrastructure is offered to customers of Wahlzeit.

After that only 3 offerings are shortlisted. To avoid unnecessary adjustments specific to one offering, only providers matter that host standard Java applications which is not the case for IBM Bluemix. The decision between the 2 remaining ones Google App Engine and OpenShift is made by the amount of free storage that is necessary for all the photos and other data. As the result of this analysis Google App Engine is selected as target platform for Wahlzeit. This service is described in detail in the following section.

4.2 Description of Google App Engine

As already mentioned, Google App Engine is a platform for web applications classified as PaaS. You can use it to develop and run own web applications on top of Google's infrastructure (Ciurana, 2009, p. 1). It is available since 2008 (Buyya et al., 2009, p. 602). Within limitations you can use the application free of charge. The paradigm of a PaaS to abstract the underlying runtime, applies here, too. You get load balancing and automatic scaling out of the box. When you need additional resources, automatic clones of your application are created (Walraven et al., 2014, p. 681). Your application runs within a secure sandbox without harming others (Ciurana, 2009, p. 1).

GAE is one of the most popular PaaS offerings. Google (2015c) lists important customers of GAE on their website like

- Coca Cola – used GAE for their Happiness Flag[1], the worlds largest mosaic flag.
- Udacity – offers massive open online courses. They build most of their services on GAE.
- Song Pop – a mobile app to guess a song's artist or title. They build their backed on GAE.

[1] http://www2.happinessflag.com

Starting with Python as the first supported programming language, GAE now also supports Java, PHP, and Go (Google, 2015e). To work with them, download the according Software Development Kit (SDK). It contains the program `appcfg` to upload and manage your application (Google, 2015n).

Besides the runtime environment GAE offers lots of cloud services like user and mail service (Walraven et al., 2014, p. 682) and several storage possibilities like

- Google Datastore for Java objects with 1 GB free quota (Google, 2015h),
- Google Cloud Storage for Binary Large Objects (BLOBs) with 5 GB free quota (Google, 2015m), and
- SQL database without free quota (Google, 2015a).

You control your application via a web interface, the Developer Console. It offers an overview of the ongoing activities with, logs, traces and different kind of dashboards. You have direct access to your source code. Logs and stack traces are linked to it, like IDEs offer it. You can look inside Google Cloud Storage or Google Datastore to add, modify and remove items. To log in the Developer Console use your Google Apps account (Ciurana, 2009, p. 2). There are a lot more features, but they are less relevant for Wahlzeit.

Applications on GAE are hosted by default under the `appspot.com` domain[2] (Ciurana, 2009, p. 2). Now the target platform and its restrictions are fixed. The following section estimates the migration effort to GAE.

4.3 Estimation of migration effort

As already mentioned in the previous section, using a GAE SQL database would cost money from the first query and dataset and is therefore not applicable for Wahlzeit during the lessons. Hence a new data store with a free quota has to be found. How this will be solved is explained in Subsection 4.3.1. Subsection 4.3.2 analyzes what other classes of Wahlzeit have to be adjusted, while Subsection 4.3.3 estimates the total migration effort and finally answers the question whether it is faster to migrate Wahlzeit to GAE or to redevelop it from scratch.

[2]You find an instance of the migrated Wahlzeit at wahlzeit2.appspot.com

4.3.1 Migrating storage components of Wahlzeit

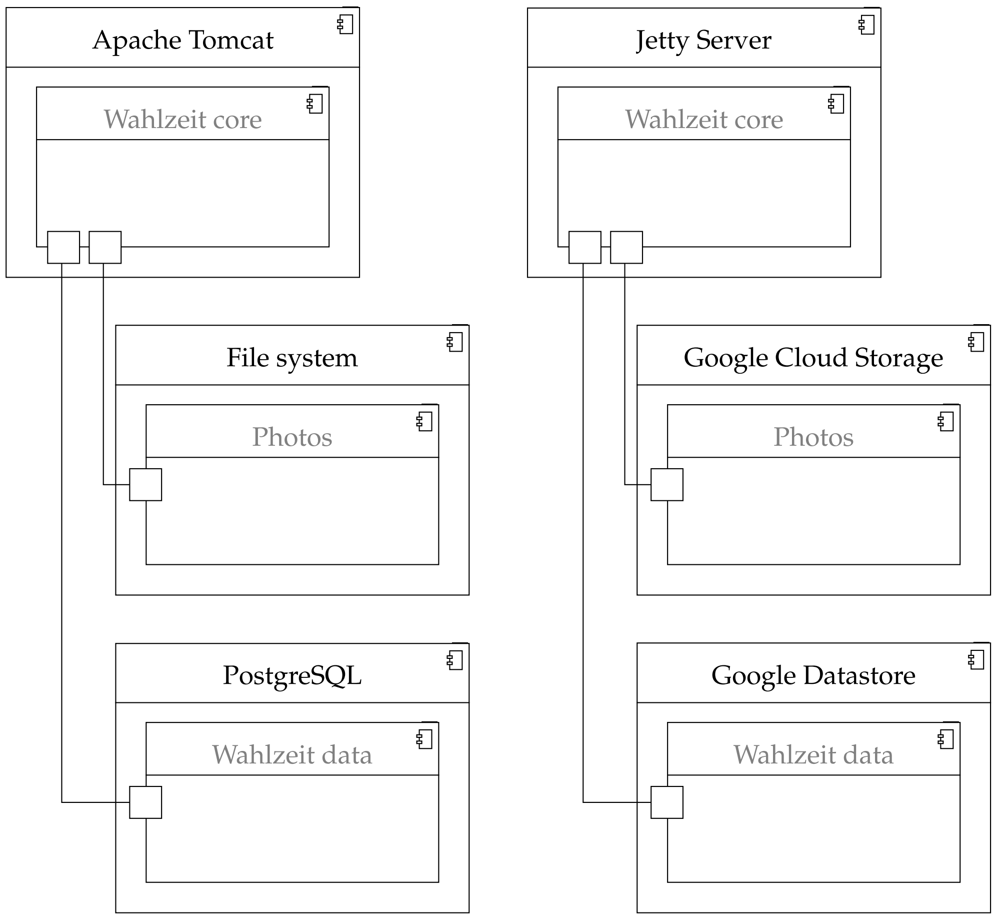
As mentioned in the previous section, GAE offers SQL databases, but not a free quota for them (Google, 2015a). Legacy Wahlzeit uses a PostgreSQL database to store data like user information, tags, and photo meta data. Hereafter this data except the photos is just called Wahlzeit data. Photos are stored directly on the file system. We will treat both type of data, the Wahlzeit data and the photo BLOBs different.

Google offers the schemaless NoSQL database called Google Datastore. We will use this for the Wahlzeit data (without photos). Its important features for Wahlzeit are (Google, 2015h):

- *No planned downtime* — a highly reliable database for the Wahlzeit data.
- *No fixed entity schema* — adding or removing properties of existing entities will become a lot easier.
- *Automatic scaling* — you do not have to care about setting up additional databases and synchronizing them.
- *1 GB free quota* — enough for hundreds of users, tags and other data.

The main drawback of this new storage system is the non trivial migration (Binz et al., 2011, p. 2). SQL statements have to be replaced by read/write operations for the Datastore. Operations like joins or aggregated queries are not possible (Google, 2015h). The concept of a database connection has to be removed and no table layout is predefined. But, despite this migration effort, it is feasible to migrate from SQL to Google Datastore. As already mentioned it will bring some essential advantages beside just recreating the old functionality. Hence what currently is stored in PostgreSQL will be migrated to Google Datastore (see Figure 4.2). This is the first big migration task.

Beside the Wahlzeit data there are the photos. Figure 4.2(a) shows the photos of Wahlzeit 1.2 are stored directly on the file system without any database. This has to be changed because you can not write files on the GAE file system after uploading your application. You can only read static data (Vu & Asal, 2012, p. 276). Hence the place for photos has to be changed to some GAE compatible technology. Uploading and therefore storing new photos is an essential feature of Wahlzeit that has to be kept in the cloud, too.



(a) Wahlzeit 1.2 runtime system (b) Wahlzeit 2.0 runtime system

Figure 4.2: Changes of the runtime systems between Wahlzeit 1.2 and Wahlzeit 2.0.

Although you might argue to save the photos in the Google Datastore, too; this is out of several reasons not the best idea. First, the quota of 1 GB has to be split up on both, the Wahlzeit data and the photos, where the latter might consume 2 MB or more for each. Second, Google offers a dedicated space to store BLOBs like photos, the Google Cloud Storage (Google, 2015g). This technology offers several benefits for Wahlzeit as a photo storage:

- *5 GB free quota* — perfectly fine for ~2500 photos (Google, 2015m).
- *Access control lists* — limit access only to the ones that are allowed.
- *Resume upload feature* — a robust way to upload photos to Wahlzeit.

Because of these features Google Cloud Storage will be the new place for Wahlzeit photos in GAE (see Figure 4.2(b)). This is the second big migration task.

Beside the adjustments of the storage systems, there are other migration tasks left, that are systematically analyzed in the following section.

4.3.2 Identify classes of Wahlzeit that have to be adjusted

Although GAE offers a Java Virtual Machine (JVM) it does not support all classes of the Java Runtime Environment (JRE) (Vu & Asal, 2012; Weisbecker, Falkner & Höß, 2014). Therefore Google provides a list of all supported classes, the JRE Class White List (Google, 2015j). This is a good basis to check where to adapt Wahlzeit in order to migrate it to GAE. But manually searching through all the 124 classes of Wahlzeit (without test classes) and comparing them with the 1422 entries of the JRE Class White List is very inefficient and takes a lot of time. Nevertheless we need the information which classes have to be adjusted and which part of them. This is necessary to answer the question, ”is it practicable to migrate Wahlzeit to GAE?”

To identify classes of Wahlzeit that have to be adjusted we developed the GAE Java Class Checker. It is a python program that searches in your Java source code for GAE incompatible Java classes (see Appendix A)[3]. The program uses the JRE Class White List as basic amount of supported classes for the cross check. You can extend the list by additional packages/classes that you have tested successfully on GAE like third party libraries. There exist several libraries that work on GAE (Frey, Hasselbring & Schnoor, 2013, p. 1113). Furthermore you can exclude supported classes of the JRE because of own requirements.

Although the GAE runtime environment would throw exceptions, if you use an unsupported class and even the IDE does some checks on the JRE Class White List[4] we decided to develop an own program because of several reasons:

- There is no support by the compiler to check the compatibility at compile time. Hence you have to deploy your program, and wait for the first exception. To do this comprehensively you have to make sure that you pass all classes of your program with your test. This is

[3]Github project: https://github.com/tfrdidi/GAE-Java-Class-Checker
[4]At least IntelliJ Ultimate

an sophisticated, manual and therefore an error-prone process. The GAE Java Class Checker provides the same information even before compile time and hence saves you a lot of testing time and errors when forgetting to test a certain class.

- Although some IDEs provide a feature to cross check the JRE Class White List, this list can not be adjusted. The GAE Java Class Checker provides more than just a compatibility check for GAE, you can customize it to your own needs, e.g., for Wahlzeit the SQL classes should be blacklisted.
- If you have successfully tested third party libraries you can add them to the GAE Java Class Checker which is not possible when relying on the according IDE feature.

GAE Java Class Checker crawls the all imports of your Java code, to check if GAE it contains unsupported classes. As a result you get a list of unsupported but yet used classes you have in your code, and the place where they are used. GAE Java Class Checker requires your source code to fulfill two properties, first, no unnecessary imports and second, no wildcard imports. Unnecessary imports would cause unnecessary hints for changes. Wildcard imports would make your life harder to find the concrete class that has to be replaced by pointing only to its package. Hence adjust your imports like follows:

`import java.package.*` → `import java.package.Class1`.

To do this exhaustive and efficiently make use of your IDE [5]. This has been done for Wahlzeit as one of the first adjustments. The intermediate stage where unnecessary imports of Wahlzeit have been removed and wildcards have been replaced is tagged in the git repository with `optimizedImports`[6]. The result of running GAE Java Class Checker 1.0 on the mentioned code base are 38 different, potentially unsupported, imported classes. They are used within 25 Java files. The files contain between 1 and 11 unsupported classes. The complete list of the 25 Wahlzeit classes that have to be adjusted is visualized in Figure 4.3.

We could extend the list of GAE supported classes by deploying Java test projects on GAE in order to test the possible implementations of single functions like uploading a picture or storing information in the Google

[5]IntelliJ IDEA: https://www.jetbrains.com/idea/help/optimizing-imports.html
[6]see https://github.com/tfrdidi/MigrateWahlzeitIntoTheCloud

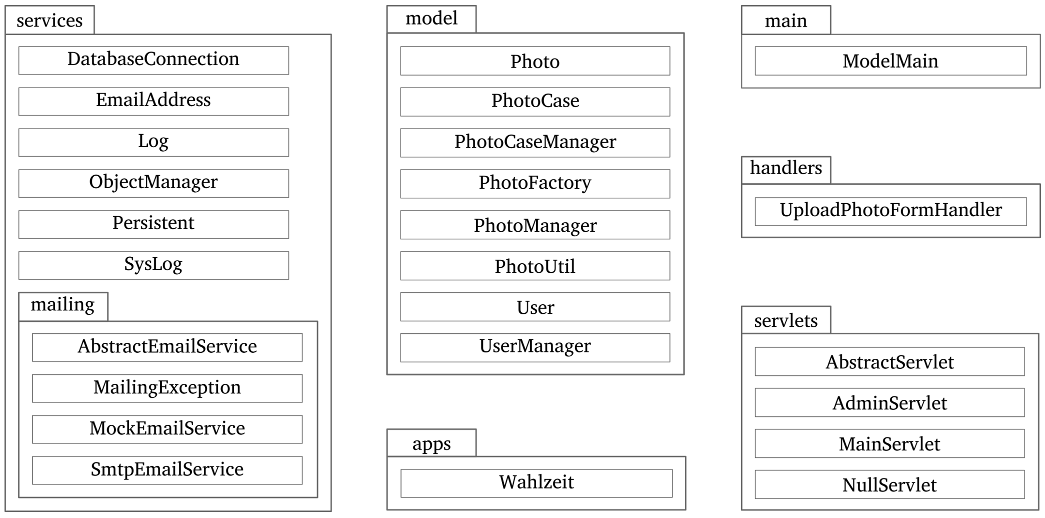


Figure 4.3: Wahlzeit classes importing GAE incompatible classes.

Datastore. The following can be added to the list of GAE supported packages:

```
com.google.common
com.google.api
com.google.appengine
com.googlecode.objectify
org.apache.commons.fileupload
org.apache.http
```

Furthermore, the `java.sql` package is blacklisted although it is supported by GAE. Using GAE SQL would not fulfill the no cost requirement (see Subsection 4.3.1). Therefore, SQL statements have to be replaced by corresponding Google Datastore queries.

Besides migrating the storage systems of Wahlzeit — 1. Datastore for Wahlzeit data and 2. Cloud Storage for photos (see Section 4.3.1) — the results of GAE Java Class Checker show further construction sites:

3. replace `java.awt` photo management by GAE compatible library,
4. use Java Servlet 2.5, higher is not supported (Google, 2015i), and
5. adjust the sending of emails.

Beside the already mentioned 5 migration tasks, some literature research on other pitfalls has been done.

Wahlzeit is based on sessions, i.e., information is saved on a `UserSession` object that is valid for the communication between one client within a certain time period. By contrast GAE is by default stateless, i.e., it does not use sessions (Jonge, 2011, p. 78). But it is able to support sessions at the cost of using Google Datastore and Memcache for session handling. This will consume both free quota and performance. It furthermore comes along with the requirement that all objects being stored in a session have to implement the `java.lang.Serializable` interface (Jonge, 2011, p. 36). Although using GAE sessions comes along with a major refactoring effort, this task will be the 6. migration task of Wahlzeit in order to run on GAE.

Further construction sides are sending emails to user in a regular interval and tidying up the Datastore. Using GAE in its intended way it is difficult to execute tasks asynchronously or in a regular interval, e.g. once a day. Usually a Hypertext Transfer Protocol (HTTP) request is processed and a respond is sent within few seconds or less. Google used this mechanism as a basis for their Task Queue service, that allows you to create a small set of tasks that run in the background (Malawski, Kuźniar, Wójcik & Bubak, 2013, p. 51 ff.). This service can be used for the long running processes of Wahlzeit like sending emails to all subscribed users in a regular interval and tidying up the Datastore by removing outdated sessions. This is the 7. migration task of Wahlzeit.

For long running tasks that arise when processing an HTTP request, the Task Queue can be used, too. One candidate is uploading a photo that has to be scaled and saved for each of the offered sizes. Adjusting this process is not an essential migration step, but a nice performance optimization for the user.

Whether the effort of migrating Wahlzeit is smaller than redeveloping it for GAE or not, is discussed in the following section.

4.3.3 Discussion of migration effort

As identified in the previous subsection, there are 7 essential migration tasks. To solve them the analysis of the GAE class checker showed, that 25 of the 124 Java classes (without unit tests) have to be adjusted. These numbers do not indicate that there would be challenges when migrating Wahlzeit that result in so much effort that creating a new GAE Wahlzeit from scratch would be a better solution. According to Mooney (1990, p. 59) Wahlzeit is portable. That means it will be migrated and not redeveloped for GAE.

Beside the effort estimation, we check the requirements for Mandatory Goal 1, the migration into the cloud (see Section 2.2). You see the result in Table 4.2.

Requirement	Solution	Fulfilled?
Reduce initial effort to deploy Wahlzeit on own PC	Gradle wrapper reduces this to checking out the repository and executing a single command	√
Reduce regular effort to deploy Wahlzeit online	Gradle wrapper reduces this to executing a single command	√
Improve debugging for Wahlzeit online	GAE offers cloud debugging, live logging linked to source code, and traces	√
Reduce maintaining effort for the department staff	No effort at all necessary because each student has its own infrastructure	√
Costless infrastructure during the lecture	Generous, temporal unlimited free quota, enough for medium web sites	√
Migration of all existing functionality	There is no feature of Wahlzeit that could not be migrated to GAE	√
Same performance in the cloud	No reason why Google infrastructure should be slower than own server	√

Table 4.2: All the requirements necessary for successfully migrating Wahlzeit to GAE are fulfilled.

The migration effort will be significant smaller than redeveloping all 127 class of Wahlzeit, the requirements for Wahlzeit in the cloud can be fulfilled, hence we migrate Wahlzeit to GAE.

As the cloud is a new environment for Wahlzeit we check in the following section how the development process changes in this new environment.

4.4 Development process in the cloud

Under the term *development process* we understand the following activities, programming, building, testing, and deploying. This covers for Wahlzeit not the complete Application Lifecycle Management (ALM) (Rossberg, 2009) that starts already with requirements engineering.

The activity of programming does not really change in the cloud. Hence we focus in this section on the latter three activities especially with regard to continuous integration. These activities are at least triggered by the IDE, which therefore plays an important role.

When you search for *the best Java IDE* you find very versatile answers and lots of ongoing debates as they are known for computer science. We therefore do not want to focus on one IDE. Our claim is to completely remove all IDE dependencies, to support the most developers possible, even those working without an IDE. Beside that we analyze in this section how building, testing, and deploying change in the new cloud environment, and how they collaborate.

4.4.1 Remove IDE dependencies

Google (2015p) introduces Apache Maven as software project management tool to speed up development of GAE projects. It is able to manage dependencies, build your application, start the local development server, and deploy your application to GAE. Anyway a software management tool is the right step towards IDE independence, because you do not need an IDE at all for the mentioned activities by using such a tool.

On top of Maven, there is Gradle (Muschko, 2014, p. 23):

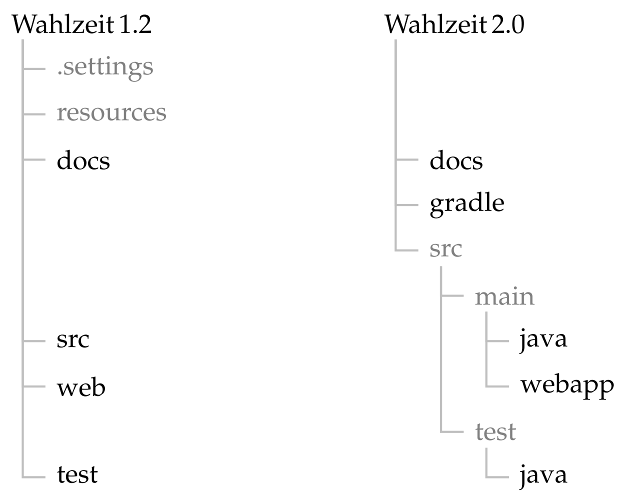
> ”Gradle is the next evolutionary step in JVM-based build tools. It draws on lessons learned from established tools like Ant and Maven and takes their best ideas to the next level.”

As Gradle is an advancement of Apache Maven, it is our first choice. There is a GAE plug-in for Gradle, the gradle-appengine-plugin[7]. It supports all GAE related tasks like starting local test environment or deploying your application to GAE. It is used for Wahlzeit 2.0 in version 1.9.22. Gradle offers lots of advantages compared to Maven. We want to list three of them (Muschko, 2014, p. 26–28):

- reduced build script size,
- better readability, and
- it can use Maven archives but also others.

[7]see `https://github.com/GoogleCloudPlatform/gradle-appengine-plugin`

Gradle projects have a typical layout. Hence, we adjusted the layout of Wahlzeit 2.0 accordingly. You can see the detailed changes in Figure 4.4.



(a) Old project layout (b) New project layout

Figure 4.4: Comparison of the old and the new project layout of Wahlzeit. The directories *resources* and *.settings* have been removed. The *gradle* directory has been added. The source code directories are now grouped below the *src* directory.

The directories *.settings* and *resources* are removed, because they contain Eclipse specific stuff and old logging configurations respectively. All the source code is moved into the *src* directory, divided in *main* and *test*.

Gradle offers a wrapper that allows you — no matter if you have installed Gradle or not — to run the Gradle build script. To enable the user to start Wahlzeit directly after checking it out of GitHub, we have to put the Gradle wrapper in the repository. It is located in the directory *gradle* (see Figure 4.4(b)). For Eclipse 4.5.0, you just have to install the Buildship plug-in[8] to run Gradle tasks directly out of the IDE.

Another advantage of the new project layout: IDEs like Eclipse 4.5.0, or IntelliJ IDEA Ultimate 14.1.4 recognize it properly. Importing the Wahlzeit 1.2 project the first time, leads to lots of errors, because the path to the *webparts* package is not recognized properly. Furthermore *main* instead of *org.wahlzeit* was interpreted as root folder, which has to be adjusted in the project structure of the IDE manually. Now the user experience when

[8]see `http://projects.eclipse.org/projects/tools.buildship`, tested version 1.0

importing the Wahlzeit 2.0 project the first time is much better and no errors are thrown.

After importing the project you usually first build the software to check if this process works before extending the software. This process is described in the following subsection.

4.4.2 Adjust building activity

When importing Wahlzeit 1.2 the first time in your IDE you have to add all the dependencies to the IDE project manually in order to build it properly. For Wahlzeit 2.0 we have the Gradle build file and a new local GAE test environment which does all the work out of the box. When you clone Wahlzeit 2.0 from GitHub, open it with your IDE or just open a terminal and navigate to the Wahlzeit directory. There execute `gradle war`, and Gradle builds the WAR file. For deploying it locally and uploading it to GAE we do not need this step, there are two Gradle tasks that do this for us. They include the building process (see Subsection 4.4.4).

The magic of Gradle to include the Wahlzeit dependencies out of the box and to provide the necessary tasks is hidden in the Gradle build file[9]. It is a Groovy file with configuration objects. It uses three plug-ins to build Wahlzeit 2.0: `java`, `war`, and `appengine`. They provide the necessary Gradle tasks to build a Java application, to export a WAR file, and to deploy it to GAE respectively.

Thereafter in the `buildscript` object the external dependencies[10] are defined. For Wahlzeit 2.0 this is the `gradle-appengine-plugin`. Furthermore the current `gaeVersion` is set in order to have only one place in this build script where it is defined. As you can see it is used to keep all the dependency versions consistent.

What follows is a check for the environment variable `JAVA_HOME`. It has to be set to a valid Java installation in order to execute the mentioned Gradle tasks. It is checked here explicitly because otherwise only a "could not find tools.jar" message is displayed which does not give a hint for a novice developer, that `JAVA_HOME` is the problem and not something with *tools.jar*.

Afterwards the repository for the `dependencies` is set to `mavenCentral`. They follow thereafter. As the dependencies are declared in this file

[9] *build.gradle* in the root of the project folder.
[10] https://docs.gradle.org/current/userguide/organizing_build_logic.html

explicitly you do not need to maintain them in your IDE project manually. For executing local unit tests special testing dependencies are necessary which are only used to compile the tests.

After the dependencies the `test` object follows. It offers you a filter to specify exactly what classes are tested when executing `gradle test`. This makes test suites redundant which is described in Subsection 4.4.3.

For the GAE configuration you can specify the port of the local test environment within the according `appengine` object. You can specify whether to download the GAE SDK or to use an already installed one. Furthermore you can specify the port for debugging the local test instance and the authorization method to deploy a new version.

The next step after building Wahlzeit 2.0 is executing the unit tests. How this process will change is described in the following subsection.

4.4.3 Adjust testing activity

Wahlzeit comes along with 13 JUnit test classes containing 54 test cases within 24 Java files. They are based on the outdated version JUnit 3. Hence at least the testing framework has to be updated to the current version of JUnit 4.12 (effective August 2015). The in-between updates come along with some nice new features.

The 13 JUnit test classes within 24 Java files suggest that there is some kind of overhead, necessary or not. Hence let us have a deeper look at the test package `org.wahlzeit.handlers`. It contains 6 files, but only one of them is a JUnit test class. There is an interface defining a method necessary for setting up the test environment, 2 test suites, one super class for the JUnit test class, and the JUnit test class `TellFriendTest` itself. The other packages have more JUnit test classes, but they still have at least one test suite.

What has been created after best practices for code reuse and single responsibilities has its limitations:

- only code reuse between classes with the same super class is possible,
- inherit from two sources is impossible, and
- the code for one unit test is spread above several files.

We have to keep in mind that the great part of Wahlzeit developers today are students that have about four months to understand and learn from Wahlzeit and afterwards extend it with own functionality. Hence the unit tests, which are an important part of good software development, have to be up to date and easy to understand.

The current version of JUnit offers **rules**. A rule is setup or tear-down code encapsulated in a Java class. It can be executed, before a test class, before each test case, after a test class, or after each test case, or an arbitrary combination of them. One test class may use several rules. For rules with dependencies between each other, you can specify the order of execution with **rule chains**. This maximizes the principle of single responsibility and strongly reduces duplicated code. Setup code is only written once, encapsulated in a rule, and used wherever it is necessary. Adjusting the existing unit tests to use rules and rule chains properly is the first task in order to update them to the current JUnit version. Rules can be used from each test class, hence they should be placed in one dedicated package `org.wahlzeit.testEnvironmentProvider`.

Beside the proper encapsulation of setup and tear down tasks with rules, we have to think about the **test suites**. Each package has its own test suite. Their only purpose is to bunch the execution of all unit tests in one package. All test suites are themselves bunched in another global test suite. This was intended to easily execute all unit tests or only a selection of them. But those manually maintained test suites are sources of errors. Each time a unit test is deleted completely, the test suite has to be adjusted, otherwise you get a compile error. Every time you add a new unit test, you have to make sure to add it to the corresponding test suite, otherwise it will not be executed. The same applies to new or removed packages in the test project.

If a cascade of test suites would be the only possibility to get the mentioned functionality you could easily argue to hazard the consequences of manually maintaining them. But IDEs like Eclipse or IntelliJ IDEA offer the same functionality with a single right click on the corresponding package without a test suite. Furthermore, you can create JUnit run configurations where you can select arbitrary combinations of test cases. Even Gradle offers the possibility to write tasks that execute a specified amount of test cases (see Subsection 4.4.2). Hence we simply remove all those test suites.

After the JUnit tests are executed successfully the next activity in the development process is the deployment to GAE which is described in the following subsection.

4.4.4 Adjust deployment activity

The deployment activity contains both, deploy Wahlzeit locally and deploy it to GAE. The GAE SDK offers a local test environment, including fully functional Datastore and Cloud Storage mocks. Starting this test environment with Wahlzeit 2.0 is one command, `gradle appengineRun`. This starts a local web service that you can reach under `localhost:8080`. To look inside the Datastore or the Cloud Storage go to `localhost:8080/_ah`, the admin interface. You can add, modify or remove elements of both Datastore and Cloud Storage. You can also debug the local instance of Wahlzeit 2.0. IDEs like Eclipse or IntelliJ IDEA offer remote debug run configurations that can be uses to connect to `localhost` via port `8000`. You can debug Wahlzeit like a normal Java application.

Deploying Wahlzeit online is also manged by the Gradle. Therefore execute `gradle appengineUpdate`, that uploads Wahlzeit to GAE. To authorize for uploading an application we use OAuth 2.0. The first time you deploy an application you have to allow this deployment by accepting the process in an new browser window that is triggered by the upload process. You can also use your SSH key to authorize, then the deployment needs no interaction at all. This one liner is much simpler and faster than logging in the web service of the department server to upload and deploy the WAR.

In the following subsection we want to analyze how those three activities work together.

4.4.5 Collaboration of building, testing, and deployment

After the concept of continuous integration it should be assured that building, testing, and deployment are always executed in this order (Fowler, 2000). This assures that each deployed version has successfully passed the testing phase. For Wahlzeit 2.0 this has to be analyzed in detail. The unit test do not cover every aspect of Wahlzeit especially not the HTML creation and the UI. Therefore a manual test is necessary for both, Wahlzeit deployed in the local test environment and deployed to GAE. The complete workflow is visualized in Figure 4.5.

After building and during each test phase, errors may occur that trigger adjustments and hence start the workflow from the beginning. As local computer and GAE are different environments, a manual test should be done for both. GAE for example keeps the content of the Google Datastore

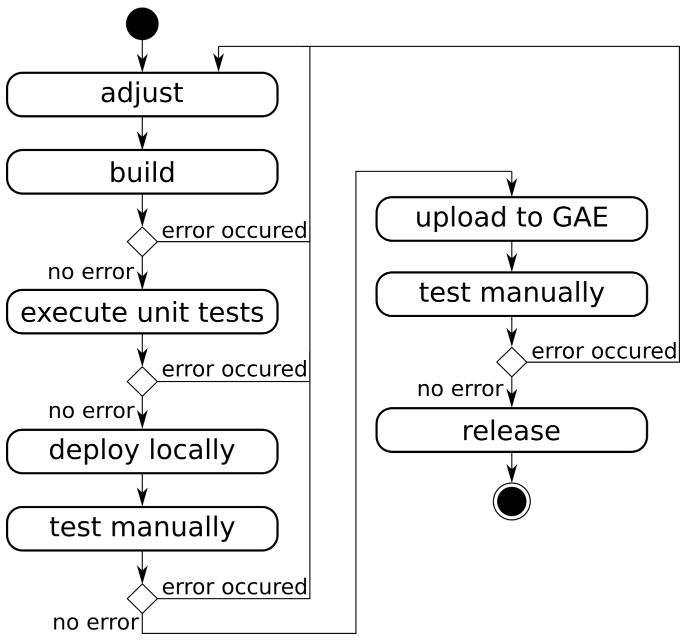


Figure 4.5: Wahlzeit 2.0: collaboration of building, testing, and deployment.

and the Google Cloud Storage when deploying a new version, in contrast to the local testing environment. Uploading and deploying an application to GAE takes some time, therefore upload only a locally tested version.

The advantage of a build server to have a convenient build environment for all developers is nearly disappeared due to the new Gradle setup. It could support the workflow only in the latter part. A build server makes sure that every time a new version has been uploaded the unit tests are successfully passed, otherwise the new version will not be deployed. When you have executed the unit tests before running Wahlzeit locally as it is intended, the build server only causes additional effort. Furthermore setting up a build server requires a virtual machine. This infrastructure is provided by Google, but without a free quota. Hence every time a build server is used, you have to pay for it (Google, 2015l).

All in all a build server would infringe Requirement 5 (no costs) and create additional effort for the staff or the students. As already mentioned the manageable additional value of a build server is not worth infringing two requirements. As a Wahlzeit developer you are free to set up your own build server, but no central build server will be provided for the lecture.

In the following section we want to analyze the effort which is necessary to develop a new Wahlzeit UI.

4.5 Effort for UI renovation

The UI of Wahlzeit 1.2 is inflexible for greater changes. It is created out of HTML templates and smaller modules, which themselves are dependent on the language. Currently German and English are supported. Adjusting the UI doubles the effort compared to one language independent UI with references to a list of translations. Fundamental changes in the UI affect several HTML files and the according Java handler classes.

Beside the inflexibility for the developer, the UI itself is not ready for the growing variety of devices. Google's mobile-friendly test[11] indicates that Wahlzeit is not mobile ready. The text is too small to read, the links are too close together, and there is no mobile viewport set. Visiting Wahlzeit with a bigger desktop screen ($\geq 1920 \times 1080$) only a quarter of the possible space is used with the default settings.

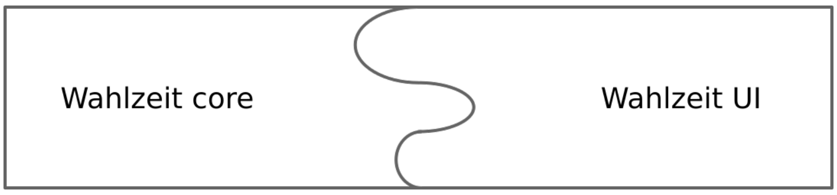
To support the mobile users we have plans for developing a Wahlzeit app. We have to keep this in mind when reworking the UI because then different clients have to be supported. In this case data and logic layer have to serve at least two different UIs. In the following we name data and logic layer the Wahlzeit core.

There are about 20 handler classes (package *org.wahlzeit.handlers*). These classes are the connection between the Wahlzeit core and its UI. A handler itself typically has two methods `doMakeWebPart` and `doHandlePost`. The first one is used to put page specific information into the result of a get request. The second handles the changes of an incoming post requests. These methods have no information about how the UI is generated. They only provide `Strings` and handle post requests. This separation is the result of a nice design that concentrates all methods for actually generating the UI in their super classes `AbstractWebFormHandler` and `AbstractWebPartHandler`. But this inheritance is the tight connection between the current Wahlzeit core and its UI (see Figure 4.6(a)).

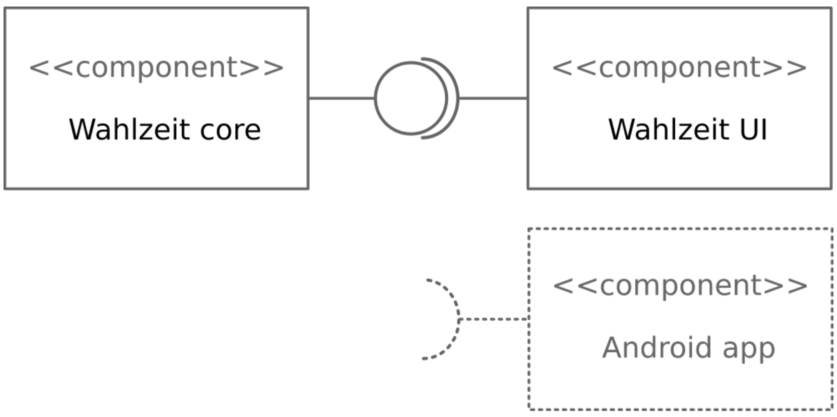
The directories *webapp.config.static* and *webapp.config.templates* contain all the HTML templates and modules that are used to create a complete webpage. They contain only UI specific stuff.

Now we want to define the target state. The core is the part of Wahlzeit that contains model and logic. It provides `Strings`[12] and photos as the result of

[11] see `https://www.google.com/webmasters/tools/mobile-friendly`
[12] Their format is discussed later.



(a) Current UI and core are tightly coupled.



(b) Distinct components with a defined interface is the target.

Figure 4.6: The current UI is tightly coupled with the Wahlzeit core. Both have to be separated in order to extend Wahlzeit by additional clients like an Android app.

HTTP requests. The Wahlzeit UI should only serve UI specific stuff, like Cascading Style Sheets (CSS), HTML files, language dependent strings like descriptions of values, and UI specific pictures like the header picture. Java Script or another technology is served (in HTML files) by the UI component to the client. Once loaded, it loads the actual data from the Wahlzeit core. With this concept we could clearly separate core and UI and make massive use of caching UI components, which is not done in Wahlzeit 1.2. In this version e.g. each request also downloads the Wahlzeit CSS file.

The first step for the UI renovation is to uncouple user interface and logic layer. That affects especially the handler classes. The result is a Wahlzeit core that only contains the data and the logic layer, and a dedicated package that contains all stuff necessary for the UI creation. In a further step the UI itself can be reworked and additional clients like a mobile app can be created.

As all of the three tasks (uncoupling, UI renovation, and app creation) are quite comprehensive. We focus in this thesis on the first one. To support different UIs for Wahlzeit, like mobile app and desktop website we plan to transform the core of Wahlzeit into a RESTful service. The necessary tasks are described in the following section.

4.6 Design of a RESTful API for Wahlzeit

As described in the previous section, the Wahlzeit core and its user interface have to be separated and the core should offer a flexible interface for more than the current website UI. Fielding (2000) first introduced the architectural software design for called Representational State Transfer (REST). It is intended for distributed software systems, that provide static content and dynamic information for a global information system in a standardized way. Exactly what we want for Wahlzeit.

There are 5 core principles for an HTTP based REST application (Tilkov, Eigenbrodt, Schreier & Wolf, 2015, p. 11–19):

1. *Resources with unique identification*: each resource or set of resources that is relevant for the user gets an unique identifier. Each resource can be accessed independent 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. The server implementation or the workflow may change but the client still stays stable. Complete new functionalities indeed are not yet available without an update of the client.
3. *Standard methods*: implement the HTTP protocol with its standard methods correctly and use only these methods. This has the advantage that whatever resource you work with, you know what operations you can use.
4. *Different representations*: provide different representations of your resources for each requirement. This enables each client that can process a resource in a specific format to use your API. This client does not necessarily have to be offered by you.
5. *Communication without status*: the server manages only resource states, no client specific sessions. This helps enormous to scale your application, shut down one of the server or to distribute the load between servers.

The necessary tasks to achieve those core principles for Wahlzeit are described in the rest of this section. For the discussion about REST we take Wahlzeit 2.0 as basis and describe the required changes from this version and not from Wahlzeit 1.2 as it was the case in the previous sections.

4.6.1 Resources with unique identification

Resources are the central concept of REST. They decide what is visible via the API and how you can change things. To clearly identify the resources, each of them should have an unique Uniform Resource Identifier (URI).

A resource itself is an abstract idea of one of your business objects. What you can request and see are representations of a resource (Tilkov et al., 2015, p. 35), for example an HTML representation of a user or a JavaScript Object Notation (JSON) representation. Internally a resource does not have to be an element in the database, it can also be generated dynamically, or calculated out of others.

Identifying all resources and structure them properly is a hard task. We want to describe a first resource design for Wahlzeit. To support this process Tilkov et al. (2015, p. 37–40) list several resource types which help us to identify all Wahlzeit resources. Indeed all of them are resources itself. Hereafter the resource types are named and the corresponding Wahlzeit resources are listed with their representation formats.

Primary resources are the core components of the software. For Wahlzeit the following primary resources with their representations for the web UI have been identified:

- REST resource 1: photo (HTML, image/jpeg),
- REST resource 2: user (HTML),
- REST resource 3: photo case (HTML),
- REST resource 4: settings (HTML), and
- REST resource 5: about, contact/imprint, and terms (HTML).

If only a certain part of a resource is used, it is called a **sub resource**. For Wahlzeit this is REST resource 6: photo meta data including the rating (HTML).

Often not only single resources are necessary but, **lists** of resources. This is the third resource type. For Wahlzeit there are:

- REST resource 7: the list of all visible photos (HTML),
- REST resource 8: the list of deleted photos (HTML), and
- REST resource 9: the list of all open photo cases (HTML).

For the case you do not want the whole list, but only a filtered subset, **filter** can be seen as a special type of list. Wahlzeit offers the function to filter photos by user name, and/or tag (HTML). This is REST resource 10.

Longer lists are divided into result sets of about 10–20 per website. This mechanism of **pagination** is another subtype of the resource type list. The current version of Wahlzeit does not offer a list using pagination, but the list of all photos uploaded by a user can be a potential candidate or the future usage of pagination.

For the case you want to show only a subset of the information a primary resource offers, a **projection** can be created. This is useful for lists, where only some important information for each list element are shown. To distinguish this type from sub resources, here mostly aggregation of information is done, whereas sub resources are just a subset of information of the actual resource. Projections could be used for the Wahlzeit main page, where the rating (REST resource 11) of the previous photo is shown.

All those mentioned resources are quite static. When Wahlzeit should have a proper REST API all sites and functions have to be provided by the API and hence have to be represented by resources. What is missing until now are **activity** resources, the next resource type Tilkov et al. (2015, p. 39) is listing. For Wahlzeit this are the following activity resources:

- REST resource 12: upload a photo (HTML),
- REST resource 13: flag a photo (HTML),
- REST resource 14: process a photo case (HTML),
- REST resource 15: tell a friend (HTML), and
- REST resource 16: administer (HTML).

For a complete REST API these 16 REST resources all need an unique identification with the corresponding URI. But without a connection they have no real value. How they are connected is described in the following subsection.

4.6.2 Links and hypermedia

A REST application has to fulfill the following properties:

- The amount links of an application represents the total amount of navigation possibilities for the user. You as a user do not have to enter an URI manually.
- You do not need an external description to use the application. All necessary information is contained, even for machines (hypermedia).
- The server uses a format that the client understands.
- The application status is based on the resource representations of the client and the resource state of the server, not on session information.

If those properties are fulfilled you get the following advantages:

- you decouple client and server,
- changes in the resource model are transparent for the client,
- the server defines the workflow, and
- external meta data is reduced.

HTTP websites are a good example for the implementation of those properties. All the information necessary to use it, is (mostly) contained. It changes, when the server decides to change and it can be interpreted by the client. In this case the client is the browser which understands the used formats and therefore can use the website, even if the website creator is not the browser creator. But websites are created for human users. They mostly do not implement the property of hypermedia — providing a self describing website that can be interpreted by machines.

Beside the website we have the plan to develop a Wahlzeit app. One exclusive motivation for developing a mobile app, is to cache and preload content for times the mobile device has no Internet connection. The argument to have a mobile app for a better user experience does not hold. There are lots of good examples for mobile websites that offer the same functionality than the desktop website, but with a much better user experience of the mobile device[13].

Hypermedia means that all the navigation possibilities are listed as links and described in a way the client can interpret them without external

[13] `http://qz.com`, `http://maps.google.com`, or `http://huffpost.com`

knowledge. How this is achieved is not specified. Hypermedia will become important for developing an app. Instead of having a fixed workflow hard coded in the app, it should be able to change the workflow from the server side. Beside that the app should give the user (in contrast to a website) the possibility to use it offline, at least until all downloaded photos are viewed and rated.

Links between resources suggest that you can follow them without problems. This means executing a get request on the link target. What other methods are offered by HTTP and how they should be used is described in the following subsection.

4.6.3 Standard methods

Fielding (2000) only specifies that there exists a limited amount of standard methods for REST, but he does not mention concrete methods. As Wahlzeit is based on the HTTP protocol, a selection of the methods defined by HTTP 1.1 will be used for Wahlzeit. They are described below.

- **get**: provides a representation of a resource. It is safe (does not change a resources state) and idempotent (can be executed several times without without changing the result). Each resource should have a get operation. This supports finding bugs, and makes your API more transparent (Tilkov et al., 2015, p. 53–55).
- **head**: provides only the header information that would be part of the corresponding get methods result. Hence it is safe and idempotent, too. This method is useful to check changes of a resource or its existence before actually downloading it (Tilkov et al., 2015, p. 55–56).
- **put**: creates a new resource specified by the URI, or changes an existing resource. This method is not safe, as it is intended to change a resources, but it is idempotent. This makes it unproblematic to send the request again if a previous one did not lead to an answer (Tilkov et al., 2015, p. 56).
- **patch**: modifies the resource specified by the URI. If the resource exists the server may create a new resource. In contrast to put, patch describes only the changes and does not contain the complete new resource (Dusseault & Snell, 2010).
- **post**: creates a new resource, too, but specifies only the resource that creates the new resource, not the URI of the new resource. As it can

neither be cached, nor it is save or idempotent it is often abused for something that can not be done with one of the other methods (Tilkov et al., 2015, p. 57).

- **delete**: removes a resource specified by the URI. It is idempotent. Often deleting is only a logical delete, not a delete in the persistence layer (Tilkov et al., 2015, p. 57–58).
- **options**: returns meta data about a resource. Among others the supported operations of the resource are part of the result. It is idempotent and save. If not explicitly defined you may not cache an answer of an options request. A server should support this operation (Tilkov et al., 2015, p. 58).

	get	head	put	patch	post	delete	options
save	√	√	–	–	–	–	√
idempotent	√	√	√	–	–	√	√
cacheable	√	√	√	√*	–	√	√*

Table 4.3: Overview of the seven HTTP standard methods and their properties that are relevant for Wahlzeit 2.0. * The result is cacheable only if it is explicitly defined.

HTTP 1.1 furthermore defines the methods **trace** and **connect** but they are not relevant for the development for the Wahlzeit REST API. You can see an overview of the seven relevant HTTP standard methods and their properties in Table 4.3.

The current handling of standard methods in Wahlzeit 2.0 is as follows: the class `AbstractServlet` is the primary servlet of Wahlzeit. Nearly all requests are routed to this class. It extends the `HttpServlet` defined by the JRE. It implements 2 of the mentioned HTTP standard methods, `doGet` and `doPost`. Both methods only do some basic operation like setting the session and the character encoding. If the Wahlzeit system is not shutting down, both methods redirect the request to `myGet`/`myPost` of the class `MainServlet`. Both methods search the proper handler for the request, calculate the processing time and generate some log messages. Then the corresponding handler for the URI handles the actual request.

Now we analyze the 6 standard HTTP methods in order to specify the target state for Wahlzeit.

The **get** method, is save, hence no writing access to any REST resource visible to the user is allowed. Logging messages or other server side analyses are allowed (Tilkov et al., 2015, p. 55) because they are not visible for the user. Beside that we have to check if all get methods are idempotent. The third task to make the get methods REST ready, is to ensure every REST resource listed in Section 4.6.1 has its own get method. This means that the structure of handlers where currently one serves a site has to be reconsidered. But this only comes along with adjustments in the UI.

The **head** method has the same properties as get. Its result is even a subset of the corresponding get method. For Wahlzeit with its current functionality and its own UI there is no suitable use case where a head method would have a real benefit. Hence an implementation of the head method is not essential necessary for making Wahlzeit 2.0 REST ready. But the head method gets several use cases when a mobile app for Wahlzeit is developed. For the mobile app, the generated traffic has a great impact. Hence head can be used to check the size of a certain picture, without actually downloading it. Depending on the current Internet connection (Wireless Local Area Network (WLAN) or mobile network) a larger version can be downloaded or a smaller one. Furthermore the status of already downloaded (cached) photos can be checked, because the may have been deleted or flagged.

Currently the **put** method is not implemented at all. Changes to resources are solely done by post. This makes sense for uploading a photo, where the URI of the new photo is not known to the client because the server generates a unique ID which is part of the URI. But for all the other changes, like adjusting user settings, flagging or rating a photo, the put method should be used. This has the advantage that the client gets the promise of an idempotent method. This is especially useful for the case the answer to the rating of a photo does not reach the client. Now the client does not know if the rating reached the server. If this is implemented with a put method, sending the request again is no problem. Like for the head method, this is not essential necessary for the web UI but it has to be implemented for the mobile app.

In contrast to put **patch** is not idempotent and only cacheable if explicitly defined. If so, a cached response may only be used for a following get or head request, not for other methods (Dusseault & Snell, 2010). This makes it to a less reliable method for updating resources. Patch has its advantages when greater resources have to be updated and only few changes to this resources have to be done. In this case only the changes are transferred instead of the whole resource. Wahlzeit has only small resources that may be changed by the user[14], hence this advantage can not be used. It may be

[14]Images may not be changed once uploaded.

interesting for future use cases where bigger resources will be introduced and modified.

Like for post, the **delete** method has only one use case: deleting an own photo. This method should be used even for the current UI in order to make proper use of the REST principles.

The last method that is relevant for Wahlzeit is called **options**. It delivers only meta data about a resource. Hence it will be useful for the mobile app in order get information about a resource, without actually downloading it. Also the list of allowed methods per resource are interesting informations for the mobile app, in order to make it as generic as possible.

This makes three essential necessary adjustments for a RESTful Wahlzeit API: rework get methods to match the requirements, use more put methods for adjusting resources, and use delete for deleting photos.

Beside the standard methods of the resources the according representation of the resources is important to decide. This is described in the following subsection.

4.6.4 Different representations

The question for the right representation formats can not have a right or wrong answer (Tilkov et al., 2015, p. 87–110). Hence we first want to describe the different resources of Wahlzeit and hereafter think about their proper representation formats.

The main resource of Wahlzeit are photos. In terms of formats we speak about JPEG, PNG, WEBP, GIF (including animated GIF), BMP, TIFF and ICO (Google, 2015f). All those formats can be uploaded and processed since Wahlzeit 2.0. When serving photos, only JPEG is used for convenience reasons. Hence, for photos we have at least two representations, first, *image/jpeg*, and second HTML. The latter is used to get the according website to show it in the browser. Currently an URI for a photo looks like `https://wahlzeit2.appspot.com/x1ac4.html`. The target should be, only to have `https://wahlzeit2.appspot.com/x1ac4` whereas the accepted media type, either HTML or JPEG decides if the whole website is served or only the photo[15].

[15]Indeed the website contains the photo, so the photo is requested for both.

All other resources work only with HTML representations. Although there is currently no handler for all resources the current UI works quite well with the existing ones. For creating a mobile app or another UI, this decision will not hold anymore. For both variants, it should be possible to get the actual content of a site (e.g. the imprint) without the header and the footer. The same applies for a more complex site like the main page, where left and right of the actual content additional informations and functions are shown.

To make Wahlzeit 2.0 REST ready, the `.html` ending has to be removed from the URIs, and the actual format should be defined in the according HTTP header. Furthermore the two representations for photos have to be implemented.

A bigger construction site will be the removal of the status which is described in the following subsection.

4.6.5 Stateless communication

Communication without status means, that the server should not store a clients status or a session[16]. This has the advantage that the client is able to access Wahlzeit from an arbitrary number of devices without a break in its view of Wahlzeit. Even for one device an expired session will be no problem anymore. Furthermore the server does not need to persist a client session which spares the free quotas of Google Datastore.

Wahlzeit 2.0 makes intense use of the class `UserSession` which wraps an `HttpSession`. The latter is managed and persisted automatically by GAE. These classes save the clients status on GAE which objects the REST principles.

The class `UserSession` of Wahlzeit 2.0 has about 360 Source Lines of Code (SLOC). Its 8 members including a `HashSet` for further parameters and the according methods have to be moved to the identified REST resources (see Subsection 4.6.1) to be persisted. The target of this task is to completely remove the class `UserSession` and the according GAE session handling, but keeping all the functionality. Beside bringing us a step closer to a RESTful Wahlzeit core, this would solve several existing inconsistencies of Wahlzeit:

[16]Except log messages or other server internal stuff that is not visible to the client.

- The praised photos and the current photo are part of the `UserSession`. That means if you come back to Wahlzeit after some days and the session has been expired, you get photos to rate that you have already rated. This is not only annoying it offers the possibility to abuse it, in order to distort the rating.
- Accessing Wahlzeit from two devices, even if you are logged in, means you get the same photos twice to rate. Each device has its own `UserSession` with the according photos to rate. This has the same disadvantages than already mentioned in the previous point.
- If you login and set some preferences like the language or your preferred photo size, this information gets lost when the session is expired or you login from another device.

This shows that removing the client sessions does not only help to get a RESTful Wahlzeit core, but also makes it more stable and better to use.

Although we want to remove the class `UserSession` the functionality of Wahlzeit has to be kept. Hence it has to be moved to other classes. For lots of members like prior photo, the photo size, the photo filter, the praises photos, and the language configuration the new target class will be the `Client`. This is identified as REST resource and hence will be persisted. For the uploaded image the new class will be the `User` because `Guests` can not upload photos. The heading will be moved to the according page handler. It should always be the same for one resource. In case it is not, the REST resource should be separated (e.g. into two activity resources) where each of them has an own static heading. The same applies for the message.

The last task to completely remove the `UserSession` is a way to identify the actual user. For a logged in user this is done by `com.google.appengine.api.users.UserServiceFactory` automatically. But for guests which are not logged in, we have to think about an own solution. Tilkov et al. (2015) suggest to create an own resource. As `Guests` are already resources, we should use this and create URIs that support this concept.

5 Implementation

In this chapter we describe the implementation details of the designs mentioned in the previous chapter. Therefore we start with the coding guidelines for Wahlzeit in Section 5.1. It is followed by a detailed analysis of the essential Wahlzeit migration steps from a Tomcat application on the own server to a GAE application with the same functionality. The chapter is completed by other adjustments of Wahlzeit in order to make it more secure and apply good software engineering principles in Section 5.3.

5.1 Coding guidelines

As the source code of Wahlzeit is intended to be evaluated and extended by lots of students every semester, a compact and speaking documentation is necessary. Riehle (2000b) lists 3 general **method types** for Java programs, **query methods**, **mutation methods** and **helper methods**. Query methods give you informations about the object, but do not change its state. This can be seen as the opposite of mutation methods. They change the state of the object but do not return anything. Helper methods neither change the object state nor give you informations about it. They encapsulate repeating operations in order to reduce code redundancy. A popular example are factory methods (Riehle, 2000b, p. 22–25).

All methods should only have one identifiable task (Beck, 1996). Hence, take it as a hint to rethink your method design, when you have problems to classify it with one of the 3 general method types. These types can be subdivided like follows:

Query method types:

- getter methods (get. . .) — return a value,

- boolean query methods (is. . . , has. . .) — return a boolean value,
- comparison methods (equals) — compare two objects, and
- conversion methods (as. . . , to. . .) — return representation of the object.

Mutation method types:

- setter methods (set. . .) — change one field,
- command methods (handle. . . , execute. . .) — potentially change several fields, or encapsulate more complicated commands, and
- initialization methods (init. . . , initialize. . .) — called for initialization or for re-usage.

Helper method types:

- factory methods (new. . . , create. . .) — create and return an object, and
- assertion methods (assert. . . , check. . . , test. . .) — make sure a certain condition is hold, returns silently if so, otherwise throws an exception.

In Wahlzeit 2.0 we use these method types in the documentation of each new or adjusted method. They are classified with the corresponding tag, for a getter method this is `@methodtype getter`.

Method types are especially relevant when you want to document a class that can be used by others, for example as part of an API. Wahlzeit is intended to be extended by programmers on potentially every part of the software. Therefore it is also important to document methods of abstract super classes and interfaces, to make sure they are properly implemented. Therefore we use **method properties**. In contrast to a method type (one type per method) one method can have several method properties (Riehle, 2000a, p. 62).

For implementing a superclass or an interface method properties are important. There exist 2 Java specific and 4 general method properties:

Java specific method properties (Riehle, 2000a):

- **accessibility** with the possible values `public`, `protected`, etc. and ,
- **evolution** with the possible values, `deprecated` and `final`.

General method properties (Riehle, 2000a):

- **Class implementation** (primitive, composed, regular) describes the structure of its implementation. Primitive methods have only one task and do not call others, composed methods have a more complex task and call other methods.
- **Inheritance interface** (hook, template, regular) describes the inheritance behavior. Hook methods have a distinct functionality that may/should be overridden by subclasses. Template methods define how to compose hook methods. They should not be overridden.
- **Class/instance level distinction** (class, class-instance, instance) a method is either allocated to a class, a class-instance, or an instance. Java class methods are all static methods. Class-instance methods only exist in `java.lang.Class`, they are allocated to a class object. All other methods are instance methods.
- **Convenience methods** (convenience or not) encapsulate another method calls as a wrapper or to make the call easier with default parameters.

Like the type of a method, the general method properties are documented in the comments of each methods with the according tag, e.g., `@methodproperty primitive, hook`.

Beside the mentioned guidelines for documentation, we want to **avoid code redundancy**, keep Wahlzeit 2.0 **extensible and understandable**, and we want that **each component** (class or method) **should have only one distinct task**.

With these coding guidelines in mind we focus in the next section on the essential changes in order to migrate Wahlzeit to GAE.

5.2 Essential migration steps

In this section we summarize the migration steps that are essentially necessary to migrate the whole functionality of Wahlzeit 1.2 to GAE. This migrations already provides a lot of benefits like a scalable infrastructure, cloud debugging possibilities, and an easier deployment.

We start this section with the changes of the persistence layer, followed by the migration of the session management. Hereafter the changes of the asynchronous tasks are described and the adjustments of the Java

Servlets. The section is finished by the migration of the logging and the email functionality.

5.2.1 Change SQL database to Google Datastore

Although Google offers MySQL databases in the cloud, we not use it for Wahlzeit because there is no free quota (Google, 2015a). Instead we use the Google Datastore. The ”Datastore is no relational database, which requires complex changes to the Java Web service. A human developer will do the required code changes” (Binz et al., 2011, p. 2).

To make the required code changes easier and to offer other developers a simple way to communicate with the Datastore, we decided not to choose the low level Datastore API. Google (2015h) lists Objectify (Schnitzer, 2015) as a very simple an convenient interface for Google Datastore. It hides most of the complexities of the low level Datastore API. Therefore we use Objectify for Wahlzeit.

Each class whereof an object is written to the Google Datastore has to be registered to `ObjectifyService`, more specific its factory. This is done in Wahlzeit 2.0 by the new class `OfyService`. Objectify works a lot with Java annotations. Besides the mentioned registration, a class that should be persisted has to be annotated as `@Entity` or as `@Subclass` of an `@Entitiy`. Wahlzeit entities are `Photo`, `Globals`, `Tag`, `Client`, and `PhotoCase`, subclasses are `User`, `Administrator`, `Moderator`, and `Guest`. In the following we name all those classes that are stored in the Datastore entities.

For entities there is no need to have a specific structure (that matches to the SQL table). You can add or remove attributes from an entity or create one just from scratch in the code. In contrast to SQL where you find your object by the according table and the primary key, Google Datastore is like a huge persistent `HashMap`. To find your entity you need a unique key that is either a `Long` or a `String` and an entity type. Hence all entity classes have to be adjusted to have such a unique key. You only have to annotate an according `Long` or `String` member with `@Id`. If you do not specify a value for this member, Objectify creates a new unique one when saving it.

For each member you can define to persist it or not. Helper variables, that serve only as cache during runtime do not have to be stored, e.g., the `uploadedImage` of the `User` that is only used from the time an `Image` is uploaded until it is stored in the Google Cloud Storage.

All classes that have been stored in Wahlzeit 1.2 at the SQL database implement the interface `org.wahlzeit.services.Persistent`. It defines the methods to load and store the objects in the SQL database and to manage their state (changed since the last write operation or not). For Wahlzeit 2.0 this interface can been reduced by all SQL related methods, all its implementations accordingly. We use `Persistent` now only to indicate whether an object has been changed, since it has been saved the last time to the Datastore to save unnecessary write operations.

Besides that changes all constructors of entity classes that get a SQL `ResultSet` as parameter can be removed without substitution. An entity loaded from the Datastore is created automatically, there is only an explicit constructor with no arguments necessary. This constructor does not need to have any logic.

One big advantage of Google Datastore is, that all its commands are plain Java and no String encoded statements like

```
getReadingStatement("SELECT * FROM photos");
```

It has the advantage, that Datastore operations can be checked already at compile time in contrast to SQL statements which are created and checked at runtime. Changing the statements has impact on all manager classes like the `PhotoManager` where most of the adjustments for Wahlzeit 2.0 have been done.

The class `DatabaseConnection`, which implemented the pool pattern (Kircher & Jain, 2002) to cache common SQL statements can be deleted without substitution. Google Datastore has not concept of database connection.

In the context of reworking all entity classes, we have to find a solution for the so called "globals". These are the last photo ID, the last user ID, the last session ID, and the last case ID. All of them are used, to make sure an ID is not used twice. To store them in the SQL database, there was of course an according table, but no Java class to encapsulate those globals. We decided to create an according Java class for several reasons:

1. There is one place in the code where all globals are saved. Once one of them is added or removed (which will be the case for the session ID) this only affects this class. Other code that references this class is recognized by the compiler or the IDE in case of a change.
2. Loading and storing each ID on its own in the Google Datastore, would increase the read and write count and reduce our free quota.

One object containing all, only costs as much as storing one ID on its own.

3. For a new developer it is now clear out of the design of the class `Globals` that there are these global variables. Nowhere was documented that those are managed by the class `ModelMain` which loads them from the SQL database and puts them into the according manager class and the other way round.

These were the main implementation tasks in order to store the Wahlzeit entities in Google Datastore. In the following subsection the migration of the photos is described.

5.2.2 Migrate the photo storage

In Wahlzeit 1.2 the photo metadata (owner, tags, praise, etc.) is stored in the SQL database. For each photo size an own image file is saved on the file system without a database. One such file is represented in Java as `java.awt.Image`. We already identified in Subsection 4.3.2 that this class and most other `awt` classes are not included in the JRE Class White List (Google, 2015j).

In GAE Google (2015f) offers an advancement of `java.awt.Image`: `com.google.appengine.api.images.Image`. It is able to support much more formats than its `awt` equivalent. Therefore we use it for the representation of a photo in a certain size. In the following we distinguish **photo**, that contains meta data, name, and references to `Images` for each available size, and **image** which represents one photo in a certain size.

In contrast to Wahlzeit 2.0 we save the `Images` in the Google Datastore or the Google Cloud Storage respectively. Binz et al. (2011, p. 2) suggest to use Adapter for such migrations. We follow this suggestion and use the Adapter design pattern (Gamma, Helm, Johnson & Vlissides, 1995, p. 133–143). It is visualized in Figure 5.1. This design helps us to reduce dependencies on storage specific code and offers us the flexibility of two Google storage options and to store `Images` elsewhere, e.g., at another cloud provider like Dropbox[1].

Due to the adapter we can easily switch the storage, either the Datastore or the Cloud Storage. The images are managed by the class `ImageStorage`. It is a technology and provider independent super class for the adapters. It

[1]A new adapter would be necessary for another cloud provider but no other changes.

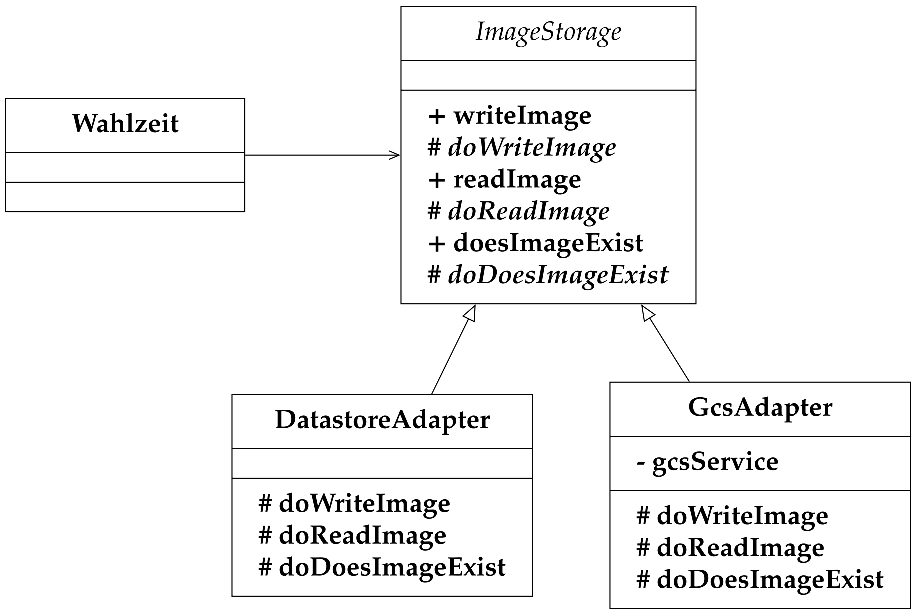


Figure 5.1: Adapter design pattern for decoupling Wahlzeit functionality and storage specific code.

contains methods that already check parameters and log the corresponding operations. The `ImageStorage` provides abstract hook methods (Riehle, 2000a) that do the actual work, implemented in the adapters. Those adapters implement the vendor and technology specific part of managing `Images`. In our case we have two adapters, one for the Google Cloud Storage, the `GcsAdapter`, and one for the Datastore, the `DatastoreAdapter`. During the lecture we use the Datastore, because this does not require to enable billing with a credit card. To change the storage, just create the according adapter in `ModelMain.startup()`.

To stay as much vendor independent as possible, the `ImageStorage` only works with `Serializables`, not with `Images`. They are casted only when they are need. This abstraction makes it very easy to use for example a third party image library or to migrate Wahlzeit to another cloud provider that does not support the Google image classes.

Requests for `Images` are handled by the `StaticDataServlet`. It is intended to serve static data like `Images` and provide them without overhead. Hence this functionality is not implemented as a handler.

Beside Images, the session handling has to be changed for Wahlzeit 2.0. This is describe in the following subsection.

5.2.3 Adjust Wahlzeit to GAE session management

GAE offers automatic session management if you enable it. Therefore you just have to put the tag <sessions-enabled>true</sessions-enabled> onto the app engine configuration file *appengine-web.xml* (Google, 2015b). Only if this is done you can get the HttpSession object directly out of the HttpServletRequest like it was the case for Wahlzeit 1.2 (see ensureUserSession() in *AbstractServlet.java*).

So far the things that behave equivalent, now we look at the changes. GAE automatically persists HttpSessions in the Google Datastore when the first request is done. This is done to support scalability. With a persisted session entity each GAE instance can manage a request no matter what instance managed the previous request. Sessions are stored as _ah_SESSION entities with the prefix _ahs (Google, 2015b). Furthermore it is persisted whenever you call the method setAttribute().

In Wahlzeit 1.2 the UserSession is an attribute of HttpSession. As the latter is not persisted and the getAttribute() method returns a writable reference to the UserSession member, this enables you to modify the UserSession without using setAttribute().

For Wahlzeit 2.0 we have to call .setAttribute() every time UserSession is changed, otherwise the changes get lost. Therefore the UserSession is changed to become a wrapper class for HttpSession. It stores its internal state completely in the HttpSession. This ensures that every change of the UserSession is persisted immediately. The interface of UserSession nearly stays the same. This minimizes the necessary changes of other classes using the UserSession.

Although the handling of the UserSession stays nearly the same, the classes that are persisted have to be adjusted. To persist a Java object in the HttpSession the corresponding class and all its members have to implement the java.io.Serializable interface. Therefore classes like PhotoId, Client, and ModelConfig implement the Serializable interface in Wahlzeit 2.0.

Although you can define when HttpSessions expire, deleting them from the Google Datastore requires a custom task. The SessionCleanupServlet[2] offered by Google only removes the sessions, but not other entities which expire with the session. For Wahlzeit this are the Guest entities. If the HttpSession is expired and deleted, the Guest entities have to be deleted, too. Therefore we decided to take the SessionCleanupServlet as a model to create our own tidy up servlet for both Guests and HttpSessions, the Wahlzeit class SessionCleanupServlet.

Its execution is triggered by a Cron job, currently every 3 hours (see *webapp/WEB-INF/cron.xml*). How asynchronous tasks or recurring tasks like deleting expired sessions are handled in GAE is explained in the following section.

5.2.4 Replace Java tasks with GAE tasks

Both environments, the Tomcat Server and GAE handle each user request in an own task. Therefore no adaption is necessary. But in Wahlzeit 1.2 there is one functionality that actively creates Java tasks: sending emails in an regular interval to all users that have subscribed themselves. To migrate this functionality we have to get active, because creating standard Java tasks, is not allowed in GAE. But there is an alternative.

As already mentioned in Subsection 4.3.2, GAE has a **Task Queue API** that offers developers to execute a set of tasks asynchronously in the background. In contrast to a web request, those tasks may need up to 10 minutes (Malawski et al., 2013, p. 51); enough for all our tasks. Each of them is triggered by an HTTP request. You can pass parameters for the task in the HTTP get method of the request (Malawski et al., 2013, p. 52).

We use the Task Queue API to **send email notifications** in Wahlzeit 2.0. The according functionality is implemented in NotifyUsersAboutPraiseAgent. The mechanism how to identify praised photos has been adapted, in order to decouple the PhotoManager and the NotifyUsersAboutPraiseAgent. Now each photo remembers the praise count of the last notification. Once a day, all photos are checked if they got praised since the last notification. If so, an email is generated and send. The according Cron job triggers the AgentServlet. This servlet processes the request and starts the

[2]see https://github.com/GoogleCloudPlatform/appengine-java-vm-runtime/blob/master/appengine-managed-runtime/src/main/java/com/google/apphosting/utils/servlet/SessionCleanupServlet.java

`NotifyUsersAboutPraiseAgent` via the `AgentManager`. In contrast to Java Threads, a running Cron job can not be stopped. Besides sending mails, two other functionalities work with the Task Queue API.

Scaling an uploaded photo to each offered size and storing it in the Google Cloud Storage takes longer than scaling and saving it directly on the file system as it was the case in Wahlzeit 1.2. To avoid unnecessary waiting time for the user, this process is done asynchronously in the background after the upload process is completed. The according functionality is implemented in the `PersistPhotoAgent` and the `PhotoManager`. This is the second functionality that works with the Task Queue API.

The third function that uses the Task Queue API is **tidying up the Datastore**. It is implemented in the `SessionCleanupServlet`. Although you can define a time span after that an `HttpSession` is expired, the according entity in the Datastore is not removed automatically. Furthermore we have to care about expired guest entities in the Datastore, too. A `Guest` is found by the according session ID. Hence, if the session is expired, the `Guest` is expired, too.

The mentioned jobs are triggered by HTTP requests handled by servlets. In the next subsection we have a detailed look on how servlets have to be adjusted for GAE.

5.2.5 Adjust Java Servlets

GAE uses the **Java Servlet 2.5 standard** for web applications (Google, 2015i). Wahlzeit 1.2 utilizes the Java Servlet 3.0 with annotations, e.g., in the class `MainServlet`. The class is annotated with `@MultipartConfig`, the Servlet 3.0 support for file upload. This feature and annotations at all are not supported in the 2.5 standard and hence in GAE.

Nevertheless file upload is possible in GAE. Google (2015d) suggests to implement it with the **Apache Commons FileUpload** package. This is what we do in Wahlzeit 2.0, too.

The main change in the photo upload process is creating an `Image` object instead of a file on the file system (see Wahlzeit 2.0 `MainServlet`). This object is associated to the current user for scaling and storing it asynchronously in the Datastore or the Cloud Storage.

Besides that, the *web.xml* has to be adjusted for GAE. This file manages the distribution from ingoing user requests to the servlets. In Wahlzeit 1.2

web.xml has 4 tasks. It defines `org.wahlzeit.apps.Wahlzeit` as the main listener. Second, it redirects requests for static files like HTML and CSS to the `org.apache.catalina.servlets.DefaultServlet` which just serves the according files as result. Third, other requests are redirected to the `MainServlet` of Wahlzeit. Forth, a list of welcome files is defined.

As already mentioned in Subsection 5.2.4, persisting photos is triggered, after the upload process is completed. As asynchronous tasks are handled by GAE by HTTP requests and an according task queue, this is processed in an own servlet, the `PersistPhotoAsynchronousServlet`. The mapping to the servlet is also managed by the *web.xml*.

Photos are served in Wahlzeit 2.0 by the `StaticDataServlet`, hence those request are directed to this servlet. This is an own class, because it requires some logic to load a photo either from the `PhotoManager`, the Cloud Storage, or the Datastore.

Also the mapping to the `SessionCleanupServlet` is introduced. Every 3 hours a Cron job triggers this servlet to search in the Google Datastore for expired session and guest entities. This servlet has a special security constraint to avoid the access of unauthorized users. Furthermore once a day the sending of user notifications is triggered. Those requests are redirected to the `AgentManager`.

Finally, Objectify requires a filter that cleans up thread local transactions and asynchronous operations at the end of a request. Hence the `ObjectifyFilter` is added at the end of *web.xml*.

To analyze the correct behavior of the servlets, we make intense use of logging. How logging changes in Wahlzeit 2.0 is described in the following section.

5.2.6 Use custom LogBuilder and remove Log4j

The self made management tool of Wahlzeit 1.2 that you use to upload and deploy Wahlzeit on the department server, offers the possibility to download the recent log files. Once downloaded you have only a text file. In case of an exception with a stack trace there is no link to the according source. Furthermore you have to manually download a new one, every time you create new log entries. Debugging like this is a cumbersome task.

GAE offers a much better logging management. In the developer console (Monitoring – Logs) you have a live stream of your application logs. You can filter them, either by the application version, the log level, the time span, or custom buzz words. Once connected to a repository, you have links to your source code, too. This makes the debugging process more efficient.

Wahlzeit 1.2 creates log messages with several additional information, like the user/service level, the session id, and the client name. Java offers a `Formatter` that you can extend to add those information to each log message. When using a `Formatter` you only have to call `log.info("test message")` and you get

`level=sl, session=dkf149KhdFklsw3, client=anon, test message`.

But it is not intended and allowed by GAE to adjust this live logging by an `Formatter`[3]. To achieve those log messages without placing those information into each log message manually, we decide to create a Builder (Gamma et al., 1995, p. 153–161) for this purpose. It is the **`LogBuilder`** with two factory methods, one for system and one user messages. User messages (abbreviated with `ul`) are the direct result of user interactions, like pressing a button or following a link. System messages (abbreviated with `sl`) are all other messages that give you informations about the internal processing of the Wahlzeit system. Beside the factory methods, there are several methods that offer extensions to the log message, like adding parameters with their name, actions, or custom messages.

GAE offers 5 logging level, whereas Java has 7. When creating a log message in the code we have to define the Java logging level. In the developer console we se the according GAE logging level. How they are mapped, and how they are used in Wahlzeit 2.0 is visualized in Figure 5.2.

The last great essential adjustment of Wahlzeit is the email service. It is described in the following subsection.

5.2.7 Adjust email service to work with GAE

There are two use cases in Wahlzeit when an email is send, first, when you register, a confirmation mail is send to you, and second, when one of your photos gets praised and you want to get notifications, once a day an according email is send. Therefore Wahlzeit 1.2 uses the `javax.mail` classes. GAE supports them, too. Hence the overall structure of the Wahlzeit package `services.mailing` does not have to be adjusted.

[3]see https://stackoverflow.com/questions/30345665

Java logging level	GAE logging level	Wahlzeit logging level
	!!! Critical	GAE critical error
Severe	!! Error	uncompensable error
Warning	! Warning	compensble error
Info	i Info	important events
Config	λ Debug	debug stuff
Fine	Debug	—
Finer	Debug	—
Finest	Debug	—

Figure 5.2: Mapping of Java to GAE logging levels and their usages in Wahlzeit 2.0.

The main changes are necessary in `SmtpEmailService`. For initializing this service in Wahlzeit 1.2 host address, port, user name, and password are necessary. In Wahlzeit 2.0 you only need host and protocol. Usually you do not have to change them which means you can use the default ones.

GAE offers 3 possibilities for the sender email address (Google, 2015k):

1. Gmail/Google Apps Account of current user,
2. Any address of the own name space: `*@appname.appspotmail.com` or `*@appalias.appspotmail.com`, or
3. Any email address in the Email API Authorized Senders list.

We send emails from a Cron job that has no signed in Google user, hence Possibility 1 is dropped. We can get `appname` or `appalias` dynamically from the according GAE `SystemProperty` and therefore do not need adjustments for a new Wahlzeit instance in order to send emails with Possibility 2. Possibility 3 requires at least adjustments in the list of authorized senders for each new Wahlzeit instance. To minimize the effort for setting up a new Wahlzeit instance, we select possibility 2 for sending Wahlzeit 2.0 emails.

We furthermore do not need a special `Authenticator` for sending emails, hence the custom `SmtpAuthenticator` class is dropped.

Beside the essential necessary adjustments of Wahlzeit to run in GAE, we did some convenience and security adjustments of Wahlzeit, which are described in the following section.

5.3 Convenience and security adjustments

In addition to the complete migration of Wahlzeit 1.2 to GAE, we made Wahlzeit 2.0 more secure (see Subsection 5.3.1), updated and extended its unit tests (see Subsection 5.3.2), and corrected the Wahlzeit 1.2 bugs that we identified during the migration (see Subsection 5.3.3). Furthermore we found inconsistencies in Wahlzeit 2.0, some of them were already present in Wahlzeit 1.2. We describe and suggest a solution for them in Appendix B.

5.3.1 Google user management instead of custom one

In Wahlzeit 1.2 the complete user management is self made. For a new Wahlzeit instance an administrator is created automatically with a default password[4]. Each time a new user registers, an according object is created and persisted. This includes both, the user name and the plain text password. This self made user management has several critical disadvantages:

- A new Wahlzeit instance has an administrator with known password.
- Each person with access to the Datastore sees all passwords.
- Each person with access to the Datastore can modify all passwords.
- The custom login mechanism has potentially errors that allow attacks.

GAE offers the Users Java API that works with Google accounts (Google, 2015o). The API offers a login screen with the according login mechanism. In the local test environment, arbitrary user name password combinations can be used. Once deployed to GAE, you need a valid Google account to sign in. Using this Google user management eliminates all of the previously mentioned disadvantages. Furthermore we do not have to care about the registration process. The only disadvantage of the Google user management

[4]The default password is documented in the readme file.

is that you need a Google account. There exist persons that do not want to create accounts in services of Internet giants like Google. In order to protect the ones better that work with Wahlzeit 2.0, we use the Users Java API from Google.

The class `UserService` creates the login screen and the Users Java API cares about the registration process. Hence we can remove the Wahlzeit login screen *LoginForm.html* and the `SignupFormHandler` with its *SignupForm.html*. Furthermore we can remove the `confirmationCode` in the `User` and the according functionality. Another member of `User` that can be removed is the `password`. It stays completely at Google. Wahlzeit 2.0 only gets the email address, the nickname, and the login status of a user.

The `UserService` also creates a logout functionality, which does not fully replaces the Wahlzeit `LogoutHandler` but in the `AbstractModelConfig` the according logout link has to be adjusted to trigger the logout functionality of the `UserService`. Another problem that is now solved is the creation of the default administrator. This creation is removed completely. When a new Wahlzeit 2.0 instance is created, it starts with no users. To get an administrator account you either have to login with the account of the one that deployed the Wahlzeit instance, or another administrator that is listed in the Developer Console of the Wahlzeit instance.

To automatically test the new and the old functionality of Wahlzeit we updated the JUnit tests to the current version of the test framework and extended them by tests of the new functionality. This is described in the following subsection.

5.3.2 Migrate unit tests to JUnit 4.12

As already mentioned in Subsection 4.4.3 all `TestSuites` have been removed in Wahlzeit 2.0 because their functionality is contained in most of the IDEs and within the Gradle build file. Furthermore redundant code of several test cases is no longer encapsulated in super classes where it is limited to the single inheritance of Java. It is now within own classes that can be used in arbitrary test classes.

We start with the package `org.wahlzeit.handlers`. In Wahlzeit 1.2 it contains five Java classes and one interface. One of the classes is the JUnit test case of this package, `TellFriendTest`. The class `HandlerTestSetup` for example ensures that the `SessionManager` provides a mock session. This functionality is used in other test cases, too. Hence we have here code duplication, because other test cases have own super classes. This setup task is implemented in Wahlzeit 2.0 by the class UserSessionProvier. It can be used via @Rules in the test cases of other packages, UserManagerTest and LogBuilderTest, too.

The TellFriendTest defines its dependencies by @Rules and @ClassRules. Two @ClassRules are setting up SysConfing and WebFormHandler before the execution of the unit tests. They have no dependencies between each other, hence no RuleChain is necessary. Before each test case the @Rules are executed to provide an empty local Datastore, a registered Objectify environment, and a UserSession. As these @Rules are dependent on each other, their execution order is defined by a RuleChain. Hereafter the setUp() method is executed. After the execution of a test case, first the tearDown() method, then the @Rules, and at the end of the test class the @ClassRules are executed.

In the package org.wahlzeit.model we made use of the expected property to explicitly define what has been implemented in Wahlzeit 1.2 like follows:

```
try {
  AccessRights.getFromInt(5);
  fail("getFromString() method did not throw
   IllegalArgumentException");
} catch (Throwable ex) {
  assertTrue(ex instanceof IllegalArgumentException);
}
```

We now have

`@Test(expected = IllegalArgumentEsception.class)`

at the beginning of the according test case. The expected property is used in AccessRightTest and GenderTest, too. Now the test cases of the regular and the ones of the error behavior are clearly separated. We applied the same for VersionTest in the package org.wahlzeit.utils.

In the models.persistence package we introduce DatastoreAdapterTest and GcsAdapterTest. They test the functionality of the ImageStorage adapters to load and persist images. As both adapters implement the same interface a super test class is quite useful.

In the package org.wahlzeit.serivces we created a new test class, the LogBuilderTest. It tests the functionality of the class LogBuilder introduced in Wahlzeit 2.0. To set up the proper test environment five

`@ClassRules` are necessary. They are interconnected by a `RuleChain` because they are dependent on each other.

The new package `org.wahlzeit.testEnvironmentProvider` contains all classes that are used as `Rules` or `ClassRules` in other test cases. They are put into an own package, because they do not belong directly to one package, they can be used by all unit tests.

Besides the test cases we corrected some bugs and inconvenient behavior that is already present in Wahlzeit 1.2. They are described in the following subsection.

5.3.3 Corrected bugs

In the following we describe bugs and inconvenient behavior of Wahlzeit 1.2 that is corrected in Wahlzeit 2.0.

Deploying a new Wahlzeit version deletes all session information. Sessions are not persisted in Wahlzeit 1.2, hence deploying a new version removes the information about the login status of all active users; as a result they get logged out and their changed settings get lost. Due to the persistence of sessions in Wahlzeit 2.0 this behavior is corrected. You only recognize that your request might take several seconds if the deployed Wahlzeit instance is currently setting up. Even this can be avoided. Therefore deploy the adjusted Wahlzeit with a new version, wait until it is ready, and then switch the default version, that is provided to the customer, to the new version.

The selected language shown at the bottom of the page and in the settings are not synchronized. In Wahlzeit 1.2 the language is saved in both, `UserSession` and `User`. Their synchronization is not very consistent, hence you can easily generate the status that the language shown in the settings and the one shown at the bottom of the page are not the same. In Wahlzeit 2.0 some properties are already moved from the `UserSession` to other classes in order to remove sessions completely (see Section 4.6). The language is one of them. It is now solely saved in the `User` class. This means no inconsistencies anymore.

Refreshing the current page loads a new random photo. In Wahlzeit 1.2 pressing F5 loads a new photo. But refreshing a page means loading the same page with the same content again. Hence we corrected this behavior. In Wahlzeit 2.0 pressing F5 loads the current page with the current photo again, instead of a new photo. This is important, if you have no stable Internet connection and the page did not arrived completely at your device. When reloading the page you do not want to see another photo. Therefore use the skip functionality.

The previous photo gets lost when another Wahlzeit site is loaded. When you rate Photo A, Photo B is loaded as a new photo. The thumbnail of Photo A is shown at the left side of the page as the previous photo. If you go now to the settings page to change some preferences and come back to Photo B, the information about the previous Photo A is lost in Wahlzeit 1.2. It is saved in the `UserSession` and removed, when a new page is loaded. In Wahlzeit 2.0 the order of photos is managed by the `PhotoManager`, so you always get the same previous photo, no matter what page you visit in between. This is also a step towards a RESTful API. The result of a request should always be the same, independent of the client session.

The first gender is shown in the profile, not the current. Wahlzeit offers you to select between two genders in your settings. The first in the list is male, the second female. When you select female and return to the settings page, male is shown. Indeed the gender is set correctly in Wahlzeit, but the HTML page that you get, always comes with the first gender selected. This behavior is corrected in Wahlzeit 2.0 with `https://github.com/tfrdidi/MigrateWahlzeitIntoTheCloud/commit/5512a577c0c45f6c208bc`.

6 Conclusion and outlook

In Section 2.2 we list the goals of this thesis. Now we analyze if they are fulfilled.

Mandatory Goal 1 is to migrate Wahlzeit into the cloud. We selected Google App Engine as target cloud service and migrated Wahlzeit with all its functionalities to this new platform. In Subsection 4.3.3, especially in Table 4.2 we verified the necessary requirements as fulfilled. The implementation is described in Chapter 5. Thereby the most important goal is reached.

Although the next goal is optional (analyze the application evolution) we fulfilled it, as described in Section 4.4. We removed IDE dependencies, adjusted the building activity with Gradle, we furthermore updated the tests. Because of cost reasons we do not change the deployment activity. Furthermore we analyzed the collaboration of all these activities. This optional goal is reached, too.

For the next optional goal, the UI renovation, we laid the foundations in this thesis. As described in Section 4.5 the first step is a clear separation of UI and Wahlzeit core. To make the core as flexible as possible for different clients, we suggest to transform it into a RESTful service (see Section 4.6).

The second mandatory goal is to analyze the multi-tenant ability for Wahlzeit. As described in Section 3.3 multi-tenant software offers some advantages for SaaS providers. But such a software must also have some core features that are not yet present in Wahlzeit and need a great effort to implement. Examples are security isolation, performance isolation, and on the fly customization. Besides that great effort, a multi-tenant Wahlzeit running on one GAE instance does not profit from the free quotas of several GAE instances. One multi-tenant instance increases effort and cost and therefore infringes our requirements for the migration. Unless it is foreseeable to sell Wahlzeit in a great extend, we recommend to use the multi-tenancy of GAE instead of implementing an own in Wahlzeit. Also this goal is fulfilled.

The last optional goal is analyzing the extension of Wahlzeit to a framework that can be adjusted by plug-ins. Similar to the multi-tenant ability, this would mean one Wahlzeit core provides different theme specific extensions. One main advantage of a plug-in framework would be a faster deploy process. As the effort of deploying a new Wahlzeit version to GAE is not that big (< 1 minute with standard Internet connection), this goal has first of all scientific character. Its application would be, running several tenants that can only adjust a limited part of Wahlzeit. It saves resources, because only one Wahlzeit core runs on the system. But great effort is necessary to secure the plug-ins from each other, and to enable the users to adjust Wahlzeit to their needs. Hence we suggest to not extend Wahlzeit to such a framework unless the general conditions for Wahlzeit change drastically.

In addition to the fulfillment of all goals, we corrected bugs and inconvenient behavior (see Section 5.3.3) and made suggestions for the remaining open ones (see Appendix B).

For the future development we recommend to make Wahlzeit a RESTful service as described in Section 4.6. Afterwards the UI should be renovated. During this process a special focus should be a responsive web site, that can be used on both, a mobile device and a desktop Personal Computer (PC) without limitations. If this is implemented correctly, the need for an additional mobile app is gone from the current point of view.

Appendix A GAE Java Class Checker

```python
import fnmatch, os, re, urllib2, collections
from collections import defaultdict

sourceDirectory = "/home/didi/Projekte/legacyWahlzeit/
    MigrateWahlzeitIntoTheCloud/src/main/java"

packageName = "org.wahlzeit"

# empirical list, based on the migration experience of wahlzeit,
# there are more for other projects
gaeSupportedPackages = ["com.google.common", "com.google.api",
  "com.google.appengine", "com.googlecode.objectify",
  "org.apache.commons.fileupload", "org.apache.http"]

blacklistPackages = ["java.sql"]

GAEWhitelistPath = "https://cloud.google.com/appengine/docs/java
    /jrewhitelist"

pathOffsetLen = len(sourceDirectory) + len(packageName) + 2

def downloadGAEJavaWhitelist():
  packageLinePattern = re.compile("<li>[a-z|A-Z|\.]+</li>")
  pageSource = urllib2.urlopen(GAEWhitelistPath).read()
  packageLines = re.findall(packageLinePattern, pageSource)
  GAEJavaWhitelist = []
  for packageLine in packageLines:
    newClass = packageLine[4:-5]
    appendNewClass = True
    for blacklistPackage in blacklistPackages:
      if blacklistPackage in newClass:
        appendNewClass = False
    if appendNewClass:
      GAEJavaWhitelist.append(newClass)
  return GAEJavaWhitelist

def isImportSupported(importLine):
  result = False
  for supportedPackage in gaeSupportedPackages:
    if supportedPackage in importLine:
      result = True
  if result == False and packageName in importLine:
    result = True
  return result

```

```
importPattern = re.compile("import .+;")
GAEJavaWhitelist = downloadGAEJavaWhitelist()
print "whitelist length: " + `len(GAEJavaWhitelist)`
filesToAdjust = defaultdict(list)
unsupportedPackages = []

for root, dirnames, filenames in os.walk(sourceDirectory):
  for filename in fnmatch.filter(filenames, '*.java'):
    path = os.path.join(root, filename)
    for line in open(path).readlines():
      for match in re.finditer(importPattern, line):
        if not isImportSupported(line):
          lineSplit = line.split( )
          package = lineSplit[len(lineSplit)-1][:-1]
          if package not in GAEJavaWhitelist:
            shortPath = path[pathOffsetLen:len(path)-5]
            filesToAdjust[shortPath].append(package)
            if package not in unsupportedPackages:
              unsupportedPackages.append(package)


print `len(unsupportedPackages)` + " diffent potentially
    unsupported imported classes:"
unsupportedPackages.sort()
for element in unsupportedPackages:
  print element

print "\n" + `len(filesToAdjust)` + " files to adjust:"
orderedFilesToAdjust = collections.OrderedDict(sorted(
    filesToAdjust.items(), key=lambda t: t[0]))
for k, v in orderedFilesToAdjust.items():
  print(k + ' includes ')
  for w in v:
    print("\t" + w)
```

Appendix B Suggested adjustments

Beside the already corrected inconsistencies and bugs listed in Subsection 5.3.3 one task raised at the end of the Wahlzeit 2.0 development, and two old inconsistencies have not been corrected, because correcting bugs is not the focus of this thesis. We list them in the following for future development.

Remove or correct dead links to a Wahlzeit blog. The page footer and the page shown when all photos have been praised, contain a link to a Wahlzeit blog. Both lead to a non existing site. We suggest either to correct those links to lead to an existing Wahlzeit blog, or to remove them completely.

Adjust the authorization method. In Wahlzeit 2.0 use the OpenID authorization method for users. But we recognized it right at the end of the development, that it is declared as deprecated. Google suggest to use OAuth 2.0 instead (Google, 2015o, 2015q).

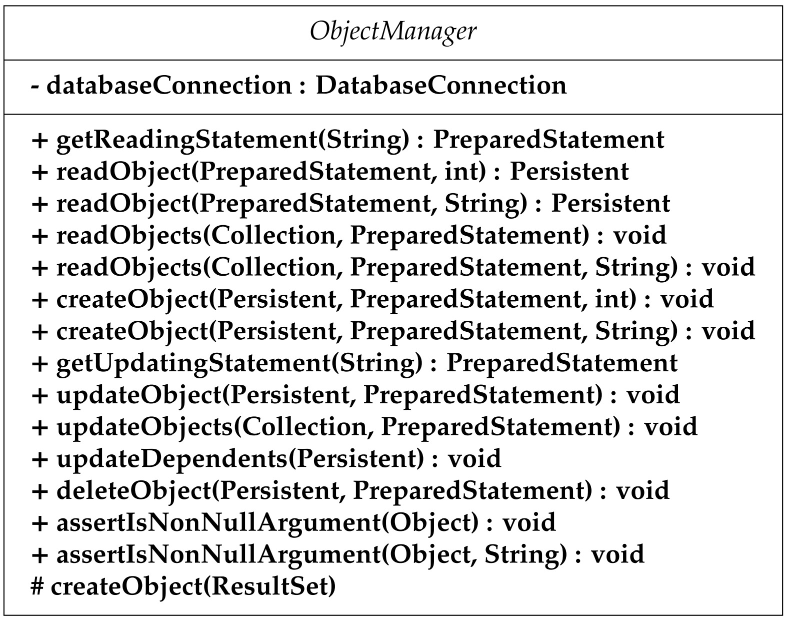
Implement a user page showing the profile of another user. For each photo in Wahlzeit the name of the owner is shown as a link, but against the intuitive expectation the link does not lead to the user page of the owner. Instead the link silently adjusts the photo filter. Afterwards only photos of the owner are shown and a new photo from the owner is loaded, if there is one left. The user does not get any visible feedback about what is happening. Neither the URI changes, nor a message is shown that only photos from this owner are shown. Only when you click on the link to toggle the filter, you see that the owner is used as filter.

There are three options to solve that inconsistency:

- remove the link completely, only show the owner name,
- when clicking on the link show a message that only photos of the owner are shown, and unfold the filter without an additional click, or
- create a user page for other users that includes a list of all uploaded photos of this user.

We suggest to implement the last option.

Appendix C Changes of ObjectManager

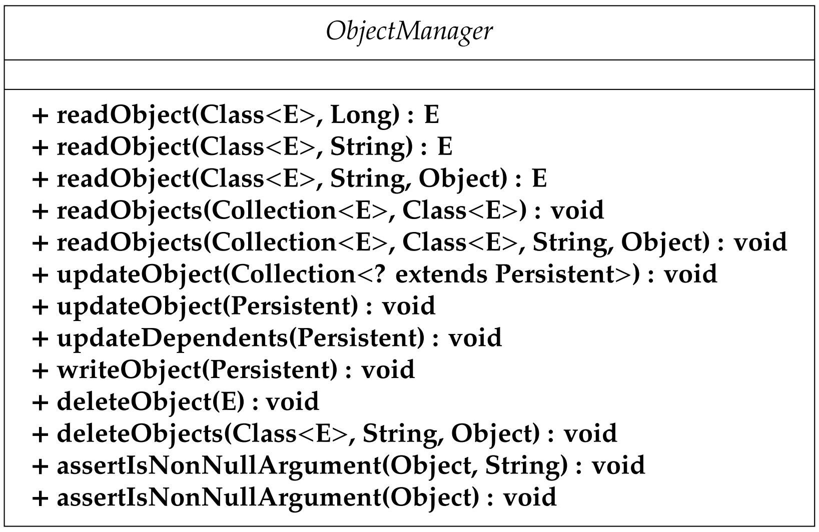


ObjectManager

- databaseConnection : DatabaseConnection

```
+ getReadingStatement(String) : PreparedStatement
+ readObject(PreparedStatement, int) : Persistent
+ readObject(PreparedStatement, String) : Persistent
+ readObjects(Collection, PreparedStatement) : void
+ readObjects(Collection, PreparedStatement, String) : void
+ createObject(Persistent, PreparedStatement, int) : void
+ createObject(Persistent, PreparedStatement, String) : void
+ getUpdatingStatement(String) : PreparedStatement
+ updateObject(Persistent, PreparedStatement) : void
+ updateObjects(Collection, PreparedStatement) : void
+ updateDependents(Persistent) : void
+ deleteObject(Persistent, PreparedStatement) : void
+ assertIsNonNullArgument(Object) : void
+ assertIsNonNullArgument(Object, String) : void
# createObject(ResultSet)
```

Figure 6.1: Old ObjectManager



```
ObjectManager

+ readObject(Class<E>, Long) : E
+ readObject(Class<E>, String) : E
+ readObject(Class<E>, String, Object) : E
+ readObjects(Collection<E>, Class<E>) : void
+ readObjects(Collection<E>, Class<E>, String, Object) : void
+ updateObject(Collection<? extends Persistent>) : void
+ updateObject(Persistent) : void
+ updateDependents(Persistent) : void
+ writeObject(Persistent) : void
+ deleteObject(E) : void
+ deleteObjects(Class<E>, String, Object) : void
+ assertIsNonNullArgument(Object, String) : void
+ assertIsNonNullArgument(Object) : void
```

Figure 6.2: New ObjectManager

References

Beck. (1996). *Smalltalk best practice patterns*. Prentice-Hall.

Beimborn, Miletzki & Wenzel. (2011). Platform as a service. *Wirtschaftsinformatik*, *53*(6), 371–375. doi: 10.1007/s11576-011-0294-y

Binz, Leymann & Schumm. (2011). Cmotion: A framework for migration of applications into and between clouds. In *Service-oriented computing and applications* (pp. 1–4). doi: 10.1109/SOCA.2011.6166250

Buyya, Yeo, Venugopal, Broberg & Brandic. (2009). Cloud computing and emerging it platforms. *Future Generation Computer Systems*, *25*(6), 599–616. doi: 10.1016/j.future.2008.12.001

Ciurana. (2009). Google app engine. In *Developing with google app engine* (pp. 1–10). Apress. doi: 10.1007/978-1-4302-1832-6_1

Dusseault & Snell. (2010). *Patch method for http*. Online: `https://tools.ietf.org/html/rfc5789`. (visited 24.08.15)

Fielding. (2000). *Architectural styles and the design of network-based software architectures* (dissertation, University of California, Irvine). Retrieved from `https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf`

Fowler. (2000). *Continuous integration*. Online: `http://www.martinfowler.com/articles/originalContinuousIntegration.html`. (visited 06.24.15)

Frey, Hasselbring & Schnoor. (2013). Automatic conformance checking for migrating software systems to cloud infrastructures and platforms. *Journal of Software*, *25*(10), 1089–1115. doi: 10.1002/smr.582

Gamma, Helm, Johnson & Vlissides. (1995). *Design patterns – elements of reusable object-oriented software*. Addison-Wesley.

Google. (2015a). *Cloud sql*. Online: `https://cloud.google.com/sql`. (visited 06.30.15)

Google. (2015b). *Configuring appengine-web.xml*. Online: `https://cloud.google.com/appengine/docs/java/config/appconfig`. (visited 08.08.15)

Google. (2015c). *Customers.* Online: https://cloud.google.com/customers. (visited 06.10.15)

Google. (2015d). *Google app engine for java questions.* Online: https://cloud.google.com/appengine/kb/java. (visited 08.11.15)

Google. (2015e). *Google app engine: Platform as a service.* Online: https://cloud.google.com/appengine/docs. (visited 07.14.15)

Google. (2015f). *Images java api overview.* Online: https://cloud.google.com/appengine/docs/java/images. (visited 08.07.15)

Google. (2015g). *Java client for google cloud storage.* Online: https://cloud.google.com/appengine/docs/java/googlecloudstorageclient. (visited 07.24.15)

Google. (2015h). *Java datastore api.* Online: https://cloud.google.com/appengine/docs/java/datastore. (visited 07.09.15)

Google. (2015i). *Java runtime environment.* Online: https://cloud.google.com/appengine/docs/java. (visited 08.11.15)

Google. (2015j). *The jre class white list.* Online: https://cloud.google.com/appengine/docs/java/jrewhitelist. (visited 06.19.15)

Google. (2015k). *Mail java api overview.* Online: https://cloud.google.com/appengine/docs/java/mail. (visited 08.15.15)

Google. (2015l). *Push-to-deploy with jenkins.* Online: https://cloud.google.com/tools/repo/push-to-deploy. (visited 07.02.15)

Google. (2015m). *Quotas.* Online: https://cloud.google.com/appengine/docs/quotas. (visited 07.14.15)

Google. (2015n). *Uploading and managing a java app.* Online: https://cloud.google.com/appengine/docs/java/tools/uploadinganapp. (visited 06.29.15)

Google. (2015o). *Users java api overview.* Online: https://cloud.google.com/appengine/docs/java/users. (visited 08.17.15)

Google. (2015p). *Using apache maven.* Online: https://cloud.google.com/appengine/docs/java/tools/maven. (visited 06.29.15)

Google. (2015q). *Using oauth 2.0 with the google api client library for java.* Online: https://developers.google.com/api-client-library/java/google-api-java-client/oauth2. (visited 08.24.15)

Guo, Sun, Huang, Wang & Gao. (2007). A framework for native multi-tenancy application development and management. In *E-commerce technology* (pp. 551–558). doi: 10.1109/CEC-EEE.2007.4

Hayes. (2008). Cloud computing. *Communications of the ACM*, *51*(7), 9–11. doi: 10.1145/1364782.1364786

Höfer & Karagiannis. (2011). Cloud computing services: taxonomy and comparison. *Journal of Internet Services and Applications*, *2*(2), 81–94. doi: 10.1007/s13174-011-0027-x

IBM. (2015). *Ibm bluemix.* Online: https://console.ng.bluemix.net. (visited 06.06.15)

Jamshidi, Ahmad & Pahl. (2013). Cloud migration research: A systematic review. *Cloud Computing, IEEE Transactions on*, *1*(2), 142–157. doi: 10.1109/TCC.2013.10

Jelastic. (2015). *Jelastic unlimited PaaS and container-based IaaS.* Online: https://jelastic.com. (visited 06.06.15)

Jonge. (2011). *Essential app engine*. Pearson Education. Retrieved from https://books.google.de/books?id=59ltM_nz048C

Karttunen, Kröger, Oja, Poutanen & Donner. (2007). *Fundamental astronomy*. Springer. doi: 10.1007/978-3-540-34144-4

Kircher & Jain. (2002). Pooling. In *Europlop* (pp. 497–510).

Majewski. (2014). *Coordinate systems.* Online: http://www.astro.virginia.edu/class/majewski/astr551/lectures/COORDS/coords.html. (visited 06.24.15)

Malawski, Kuźniar, Wójcik & Bubak. (2013). How to use google app engine for free computing. *Internet Computing, IEEE*, *17*(1), 50–59. doi: 10.1109MIC.2011.143

Mell & Grance. (2011). *The nist definition of cloud computing.* Online: http://www.nist.gov/manuscript-publication-search.cfm?pub_id=909616. NIST. (visited 06.23.15)

Microsoft. (2015). *Microsoft azure.* Online: https://azure.microsoft.com. (visited 06.07.15)

Mooney. (1990). Strategies for supporting application portability. *Computer*, *23*(11), 59–70. doi: 10.1109/2.60881

Muschko. (2014). *Gradle in action*. Manning Publications Co.

Oracle. (2015). *Oracle java cloud service.* Online: https://cloud.oracle.com/java. (visited 06.06.15)

OutSystems. (2015). *Rapid application delivery platform for the enterprise.* Online: http://www.outsystems.com. (visited 07.07.15)

Pivotal. (2015). *Home — pivotal.* Online: https://pivotal.io. (visited 07.07.15)

redhat. (2015). *Openshift by red hat.* Online: https://openshift.com. (visited 07.06.15)

Riehle. (2000a). Method properties in java. *Java Report*, *5*(5), 62–77. (Online: http://dirkriehle.com/computer-science/industry/2000/jr-2000-method-properties.html)

Riehle. (2000b). Method types in java. *Java Report*, *5*(2), 22–26. (Online: http://dirkriehle.com/computer-science/industry/2000/jr-2000-method-types.html)

Rossberg. (2009). Application lifecycle management. In *Pro visual studio team system application lifecycle management* (pp. 23–41). Apress. doi: 10.1007/978-1-4302-1079-5_2

Sadiku, Musa & Momoh. (2014, Jan). Cloud computing: Opportunities and challenges. *Potentials, IEEE*, *33*(1), 34–36. doi: 10.1109/MPOT.2013.2279684

Salesforce.com. (2015). *Data integration tools: Platform enterprise solutions.* Online: `https://salesforce.com/en/salesforce1`. (visited 07.07.15)

Schnitzer, J. (2015). *Objectify.* Online: `https://github.com/objectify/objectify`. (visited 06.26.15)

Tilkov, Eigenbrodt, Schreier & Wolf. (2015). *Rest und http*. Dpunkt Verlag.

Tran, Keung, Liu & Fekete. (2011). Application migration to cloud. In *Proceedings of the 2nd international workshop on software engineering for cloud computing* (pp. 22–28). ACM. doi: 10.1145/1985500.1985505

Vu & Asal. (2012). Legacy application migration to the cloud: Practicability and methodology. In *Services, 2012 ieee eighth world congress on* (pp. 270–277). doi: 10.1109/SERVICES.2012.47

Walraven, Truyen & Joosen. (2014). Comparing paas offerings in light of saas development. *Computing*, *96*(8), 669–724. doi: 10.1007/s00607-013-0346-9

Weisbecker, Falkner & Höß. (2014). Integrationsszenarios und -plattformen für die migration von anwendungssystemen in die cloud. *Praxis der Wirtschaftsinformatik*, *51*(2), 119–130. doi: 10.1365/s40702-014-0018-z