TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

JUKKA TUPAMÄKI

IMPROVING FRAMEWORK-BASED WEB APPLICATION ARCHITECTURE AND DEVELOPMENT IN CLOUD ENVIRONMENT

Master of Science Thesis

Examiner:
Professor Tommi Mikkonen

Instructor:
D.Sc. Risto Sarvas (Futurice Oy)

Examiner and topic approved by
the Council of the Faculty of
Computing and Electrical Engineering
on February 6th, 2013.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
TUPAMÄKI, JUKKA: Improving framework-based web application architecture and development in cloud environment
Master of Science Thesis, 52 pages
May 2013
Major: Software Engineering
Examiner: Professor Tommi Mikkonen
Keywords: cloud application, cloud computing, cloud platform, web application, web development, web framework

Frameworks are widely used in web development. As web frameworks provide the developer with basic functionality, such as object-relational mapping and database abstraction, the developer has more time to concentrate on the actual problem. A common practice has been that framework-based web applications are deployed in a single-server environment.

As the advent of cloud computing has made cloud platforms popular, web developers are facing a change in their working environment. The change does not only affect the environment but also the techniques and practices used within the current web frameworks. The main problem is that the web frameworks have their roots in the single-server era when cloud platforms did not exist in such large scale as they do currently.

In this thesis we focus on how web developers can use their previous competence in the cloud environment. The research is done by developing an example application using the Django web framework. The example application is then deployed to the Heroku cloud platform. The example application and its implementation are used throughout the thesis to identify the most common pitfalls a developer might encounter while deploying a framework-based web application to a cloud platform. The pitfalls are analysed in order to find the root causes for why the current web frameworks do not fully fit to the cloud environment.

The findings show that most of the pitfalls are related to using web framework practices or techniques that e.g. store the application state inside the server's memory or the local file system. As the cloud platform environment is a distributed system, the application state should be stored in a persistent storage and made accessible for each web server. In addition, the findings tell that developers must pay extra attention to application design and architecture in the cloud environment.

The thesis gives an analysis method for choosing third-party plug-ins and suggests ways to improve framework-based development. By specifying essential cloud platform features, the needs of an elastic cloud application are defined. The conclusions provide insight into the current status of web development and discuss how the web development can be improved by using the current cloud platforms and web frameworks.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
TUPAMÄKI, JUKKA: Sovelluskehyslähtöisen web-sovelluksen arkkitehtuurin ja kehittämisen parantaminen pilviympäristössä
Diplomityö, 52 sivua
Toukokuu 2013
Pääaine: Ohjelmistotuotanto
Tarkastaja: Professori Tommi Mikkonen
Avainsanat: pilvilaskenta, pilvialusta, sovelluskehys, web-sovellus, web-kehitys

Web-sovellusten kehittämisessä hyödynnetään usein web-sovelluskehyksiä. Sovelluskehykset tarjoavat kehittäjille toimintoja, jotka helpottavat ja nopeuttavat kehittämistä. Esimerkkejä tällaisista toiminnoista ovat olio-relaatio -mallinnus ja tietokannan rakenteen luominen sovelluksen tietomallin pohjalta. Sovelluskehyslähtöiset web-sovellukset viedään usein yhdestä palvelimesta koostuvaan tuotantoympäristöön.

Markkinointi ja hyvät kokemukset ovat tuoneet pilvilaskennan ja pilvipalvelut monien tietoisuuteen. Pilvialustat ovat suosittuja web-sovellusten kehitys- ja tuotantoympäristöiksi niiden elastisuuden vuoksi. Koska tietoisuus pilvipalveluista on kasvanut, yhden palvelimen ympäristöjä käytetään yhä harvemmin. Ympäristön muutoksesta johtuen web-sovellusten kehittäjät joutuvat opettelemaan, miten uudessa hajautetussa pilviympäristössä toimitaan. Muutos ei ainoastaan vaikuta kehittäjiin ja heidän toimintatapoihin, mutta myös sovelluskehysten käytäntöihin ja tekniikoihin. Sovelluskehysten suurin ongelma on se, että niiden kehittäminen on aloitettu silloin, kun pilvialustoja ei vielä ollut yleisesti käytettävissä.

Tässä työssä keskitytään siihen, kuinka web-kehittäjät voivat hyödyntää aikaisempaa osaamistaan pilviympäristössä. Tutkimuksen pohjana käytetään itse rakennettua esimerkkisovellusta, joka on yksinkertainen kuvanjakopalvelu. Sovellus on toteutettu Django-sovelluskehyksen avulla ja sen tuotantoympäristöksi valittiin Heroku-pilvialusta. Sovellusta käytetään tunnistamaan yleisimmät ongelmakohdat, jotka toistuvat, kun sovelluskehyslähtöisiä web-sovelluksia viedään pilviympäristöön. Löydettyjä ongelmakohtia käytetään apuna perimmäisen ongelman määrittämisessä, joka estää sovelluskehyslähtöisten web-sovellusten toimimisen sellaisenaan pilviympäristössä.

Työn keskeisimmät löydökset osoittavat, että ongelmat pilviympäristössä johtuvat usein siitä, että sovelluskehysten toiminnot ja käytännöt on suunniteltu huomioimatta hajautetun ympäristön asettamia vaatimuksia. Tällaiset toiminnot ja käytännöt liittyvät esimerkiksi web-sovelluksen tilan tallentamiseen palvelimen muistiin tai paikalliseen tiedostojärjestelmään. Hajautetussa ympäristössä nämä paikat eivät ole tilatiedolle soveltuvia, johtuen esimerkiksi pilvialustan ominaisuuksista. Lisäksi havaittiin, että monet ongelmat voidaan välttää huolellisella suunnittelulla.

Työn tulosten pohjalta ehdotetaan analyysitapaa laajenteiden valintaan. Havaittuja ongelmakohtia analysoimalla tarjotaan uusia käytäntöjä sovelluskehyksiin. Työn yhteenvedossa pohditaan, kuinka nykyisillä sovelluskehyksillä ja pilvialustoilla voidaan luoda elastisia web-sovelluksia havaituista ongelmista huolimatta.

PREFACE

I started to think about a thesis subject while working on a small cloud application project at Futurice Oy. From the ideas and inspiration that arouse from the project, I managed to find a proper subject. I thought I could write the thesis in the beginning of 2013 in two months, as I had become very familiar with the subject. The schedule looked realistic at first, but in January happened something unexpected and sad that paused the thesis work for some weeks and kept me thinking of other things.

I would like to thank the examiner of the thesis Professor Tommi Mikkonen from Tampere University of Technology and my instructor D.Sc. Risto Sarvas from Futurice Oy. Professor Mikkonen gave me excellent feedback on the subject throughout the writing process, which strengthened my motives to finish the thesis. Dr. Tech. Sarvas helped and guided in the process of writing and encouraged to think from different angles and viewpoints.

I would also like to thank my friends and family who have supported me during these times to keep moving on with the thesis project. I guess without their support I would not have managed to write the thesis as quickly as I did. And finally, huge thanks go as well to my current employer, Futurice Oy, for making this possible.

Tampere, April 14, 2013

Jukka Tupamäki

TABLE OF CONTENTS

1 INTRODUCTION

Cloud computing got popularized around the year 2005 by large companies such as Amazon, which began to offer their extra computing resources as a service for third parties. This meant, that developers could leverage the massive infrastructure of such companies. Why the companies had such computing resources was that they needed to keep their Internet business running smoothly at all times. However, during the quiet times those extra resources where not in use.

Cloud platform services are an example of services that are built on top of infrastructure services just like what Amazon offers. Cloud platform services abstract the complicated infrastructure from the developer and allow them to easily deploy web applications to the cloud environment. Mostly because of the easiness, cloud platform services have become an efficient choice for hosting web applications instead of traditional hosting services or self-owned, expensive hardware. The cloud platform service providers promise to take care of maintenance work related to the underlying infrastructure. They provide tools for e.g. scaling of application resources and load balancing between multiple web servers. Deploying a web application to a cloud platform is currently so easy that any developer can do it just by following tutorials. Although the deployment is easy, being able to run a web application on a cloud platform does not guarantee that the web application's design fits to the cloud environment. Especially framework-based web applications are likely to make use of techniques and practices that do not work in the cloud environment.

In this thesis we focus on how web developers can use their previous competence in the cloud environment. The main research questions are to find how web frameworks can be used safely in the cloud environment and how the development should be improved in order to make the development experience better. The research is based on an example web application that was built from scratch as a part of the thesis work. By the empirical approach to the subject, it is possible to reveal common pitfalls that exist in developing web applications for the cloud environment. The found pitfalls are used as a basis for improving the development process and practices, and framework features. The findings are based on the selected combination of the Django web framework and the Heroku cloud platform, but can also be applied to other web frameworks and cloud platforms because of their similarity.

Chapter 2 sums up the background and motivation behind the subject. The subject is reviewed from a web developer's point-of-view, and web frameworks' readiness for the cloud is questioned. In the chapter, literature is reviewed to show that the application layer is currently left without much attention. In addition, the chapter introduces the

server environments that are referenced throughout the thesis. The chapter reviews cloud computing in general and relevant concepts. *Chapter 3* gives an overview of distributed systems and cloud application design. The Django web framework and the Heroku cloud platform are introduced. The rest of the chapter describes the implementation of the example application and the details related to the cloud deployment. *Chapter 4* reviews the encountered pitfalls and their workarounds when applicable. The chapter also discusses the pitfalls and their root causes, and defines an analysis method for choosing third-party plug-ins. *Chapter 5* is about improving web frameworks and the development process. The chapter discusses web application's portability and also defines a set of cloud platform features that an elastic cloud application requires from the platform. The set of features is based on the needs that were encountered in the development of the example application. *Chapter 6* concludes the work and discusses what should be taken into account in web development when using the current web frameworks and cloud platforms without modifying them in any way.

2 ENVIRONMENTS AND CONCEPTS

This chapter gives an introduction to the terms and background that are relevant in understanding the thesis. In Section 2.1, a quick overview of the current status of web frameworks and cloud platforms is given. Section 2.2 introduces to the basics of cloud computing and the most commonly used acronyms are defined. In Section 2.3 the single-server environment is explained and a reference stack is given and in Section 2.4 basic concepts of a clustered environment are reviewed. In Section 2.5 a set of relevant cloud concepts is described.

2.1 Background

During the past several years cloud computing platforms have gained attention and are nowadays recognized as a serious choice for deploying web applications mostly because of their elasticity. The increasing interest towards cloud platforms affects developers in a way that they are facing a change in their working environment. As Sitaram & Manjunath (2011, p. 205) state, that for developers to utilize a cloud computing platform to its full extent, they have to learn new methods to design web applications. Developers are used to work in a traditional single-server environment, which might be a reason for that web application design is lagging behind the current requirements placed by the cloud computing platforms and therefore new methods and practices have to be learned.

Toffetti (2012) points out that cloud computing platforms have been advertised naively which has resulted in that, for example, some cloud providers have oversimplified the deployment to the cloud platforms. Because of the oversimplification, advertising has become misleading. Toffetti gives an example of an advertisement which tells porting a traditional web application to the cloud requires just deploying the same application package to the cloud platform (Toffetti 2012, p. 1–2). Although Toffetti (2012) acknowledges there are certain problems on the application level, they do not discuss the issue any further. These advertisements do not address that application level issues exist, which is likely to cause elasticity related problems with deployments such as in the Toffetti's example.

As Toffetti saw that cloud platform advertising could be misleading, at the same time software companies advertise themselves as 'rapid development partners'. What makes these companies 'rapid' is usually explained by telling they are using agile methods and some specific web frameworks, which, again, are advertised as productive, rapid and easy to use. For example, by taking a look at how web frameworks are advertised, we can clearly see that each framework promises to be the tool for building web applications:

> *"Grails Framework Full stack – Build modern, sophisticated and robust Groovy web applications in record time! [...] Rapid, dynamic, robust."*
> *– Grails homepage (29.1.2013)*

> *"Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. […] It lets you build high-performing, elegant Web applications quickly."*
> *– Django homepage (29.1.2013)*

> *"Rails – Web development that doesn't hurt. Ruby on Rails® is an open-source web framework that's optimized for programmer happiness and sustainable production. It lets you write beautiful code by favoring convention over configuration."*
> *– Ruby on Rails homepage (29.1.2013)*

> *"ASP.NET MVC gives you a powerful, patterns-based way to build dynamic […] ASP.NET MVC includes many features that enable fast, TDD-friendly development for creating sophisticated applications that use the latest web standards."*
> *– ASP.NET MVC homepage (16.3.2013)*

The benefits of using frameworks are well known, and in that sense, web development does not differ from any other software development. A framework gives architecture to the developed web application and liberates the developer to solve the actual problem. Despite this, a major downside is that the practices and conventions found and used in the current popular web frameworks originate from the single-server era, which obviously was before cloud computing platforms even existed in public. When running a framework-based web application in the cloud environment, problems will arise at some point if the framework and its plug-ins do not address the change of the environment. Therefore, the frameworks need to change, and before they have changed the advertisement hype does not have that much meaning.

Current popular web frameworks such as Grails (Grails 2013a), Django (Django 2013a), ASP.NET MVC (Microsoft 2013) and Ruby on Rails (Ruby on Rails 2013) implement the Model-View-Controller (MVC) pattern. The MVC pattern is a popular architecture pattern used in developing GUI (Graphical User Interface) applications for all kinds of devices and operating systems. The pattern separates presentation logic from business logic and data (Koskimies & Mikkonen 2005, p. 142). In an elastic, distributed environment, such as a cloud computing platform, the usage of MVC-based frameworks has to be reconsidered: When the application needs to scale, it might be impossible because of its architecture and conventions which have been inherited from the used web framework.

2.2 Cloud computing

Mell & Grance (2011) from National Institute of Standards and Technology (NIST) define cloud computing as "a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources." By the shared pool of configurable computing resources, they mean networks, servers, storage, applications and services "that can be rapidly provisioned and released with minimal management effort or service provider interaction." Mell & Grance's (2011) definition underlines rapid provisioning of resources and minimal management effort.

Armbrust et al. (2010, p. 50) specify that cloud computing refers to both the application delivery method and the underlying infrastructure. To meet Armbrust et al. (2010) definition, the application must be delivered over the Internet and the application must be hosted at a data center environment capable of offering such benefits as scaling of computing resources. By combining these definitions, we have a set of requirements that are required of all cloud computing service types:

- Rapid provisioning of configurable, on-demand computing resources.
- Minimal management effort or service provider interaction.
- The application is hosted in a data center that meets the above requirements.

Figure 2.1 positions the cloud computing service types, the end-users and the hardware. The service types are *Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS)* and *Software-as-a-Service (SaaS)*. The service types are used to describe the underlying service model. These cloud service types can be understood as service layers. In the figure, the abstraction level meter illustrates that the abstraction level grows when moving from IaaS to SaaS. At the same time, the amount of configurability shrinks.

In the Infrastructure-as-a-Service (IaaS) model, the service provider shares their hardware infrastructure resources to customers. IaaS services are considered as the lowest layer of cloud computing. Mell & Grance (2011) define the IaaS service model as the capability to provision low-level computing resources to consumers. According to them, these resources can be networks, processing units, and storage. For example, Amazon offers *Elastic Compute Cloud (EC2)* for provisioning virtual machines, *Simple Storage Services (S3)* for file storage, and *Virtual Private Cloud (VPC)* for creating and managing private networks. IaaS services abstract the provisioning of resources, which can be considered as a key enabler for elasticity. The infrastructure offered by an IaaS service can be used for building other types of cloud services on top of it. A core technology in building IaaS services is *virtualization.*

The Platform-as-a-Service (PaaS) model's purpose is to offer IaaS services in an easy manner by automating tasks such as virtual machine provisioning, configuring, and application deployment. PaaS services enable developers to rapidly deploy web applications to the cloud environment. As illustrated in **Figure 2.1**, configurability on the PaaS

layer is limited, which results in that some properties are platform specific. This fact is supported by Mell & Grance (2011) who point out that the PaaS service's consumer does not have access to the underlying IaaS service, which means that the consumer does not have control over the operating system or other IaaS level facilities. Instead, the consumer can configure certain properties of the web application server.

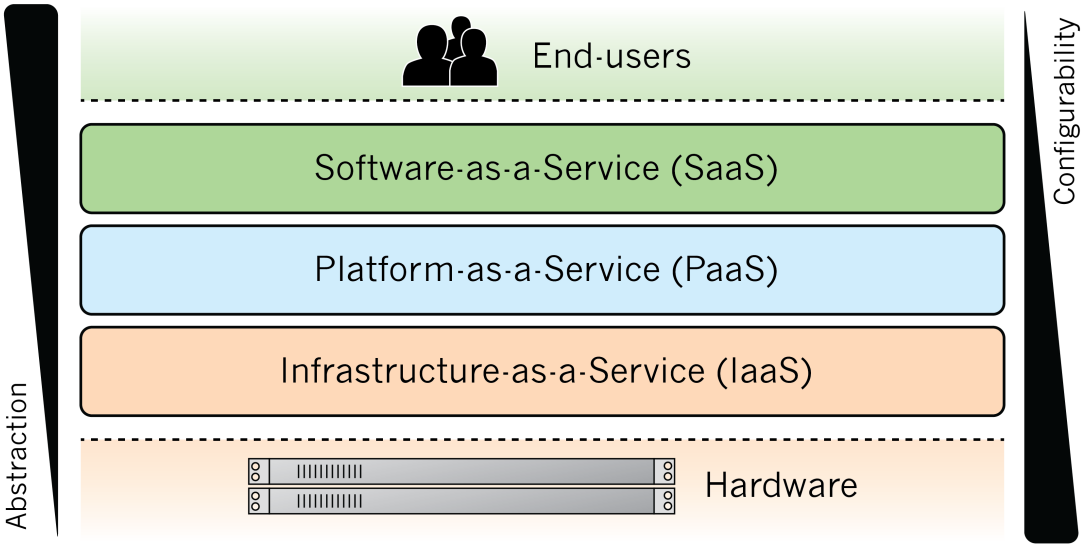

Figure 2.1. *Cloud computing service layers, end-users and hardware positioned. When the abstraction level grows, the amount of configurability shrinks and vice versa.*

The Software-as-a-Service (SaaS) model represents the highest abstraction level in cloud computing as shown in **Figure 2.1**. Zhang et al. (2010, pp. 9–10) describe that a SaaS service is an on-demand application on the layer the end-user acts on. This description is valid as a web application deployed to a PaaS service can act as a SaaS service. What makes a web application a SaaS service is the way the service is offered to the end-user. If the application does not allow 'self-service' or if the application is not served from cloud environment, it most likely is not a SaaS service. Mell & Grance (2011) provide a description for the SaaS service model, where the SaaS service itself should be possible for accessing via a thin-client, e.g. a web browser. Most importantly, Mell & Grance (2011) specify that the SaaS model's core idea is the capability to use the provider's applications. Some examples of well-known SaaS services are *Google Mail (http://www.gmail.com)* and *Dropbox (http://www.dropbox.com)*.

Zhang et al. (2010) give similar definitions for IaaS, PaaS, and SaaS as Mell & Grance (2011). According to Zhang et al. (2010) the common feature for the three is the on-demand provisioning of resources, which is also included in the Mell & Grance's (2011) definition for cloud computing. As Zhang et al. (2010, p. 9) point out, the cloud service stack is similar to the OSI (Open Systems Interconnection) model's design. When taking into account that the OSI model layers provide interfaces to the upper lay-

er and the layer below (Tanenbaum & Van Steen 2007, p. 117), we note that the cloud service layers act in the same way: A PaaS service has access to the underlying IaaS service's interface, but not directly to the hardware. Similarly, a SaaS service may use the PaaS service via the PaaS service's interfaces, but may not gain direct access to the IaaS layer. The IaaS layer may be completely hidden from the PaaS layer's users. Therefore the end-user of the SaaS service cannot easily know where the service is hosted.

According to Reese's (2009, p. 2) definition, the cloud service types are usually based on a pay-as-you-go payment model. In the pay-as-you-go model, costs are based on the amount of used resources. The payment model and the possible benefits have attracted businesses to externalize their IT infrastructure to the cloud completely or partially. By utilizing cloud services, businesses can concentrate on their core competence and save money that would otherwise be used for e.g. licensing fees, hardware upgrades, and maintenance costs.

2.3 Single-server environment

A single-server environment represents the simplest environment where a web application can be hosted in. A single-server environment consists of an operating system, a web server, and a database server. The benefits of a single-server environment are obvious: the environment is easy to setup and relatively inexpensive. For small-scale production use, prototyping and development purposes, a single-server environment is a cost-effective choice. A single-server environment is usually built-in to the operating system and enabling it is just a few clicks away. We specify a reference stack to give an idea of what kind of components and software a single-server environment consists of:

- Apache web server (http://httpd.apache.org/),
- PostgreSQL database server (http://www.postgresql.org/) and
- Ubuntu Server operating system (http://www.ubuntu.com/business/server).

In **Figure 2.2**, the client acts directly with the web server by sending an HTTP request to the web server. The web server processes the request and forms an HTTP response that is sent back to the client's web browser. The web server interacts with the web application to form the response. The web application may access the local file system and the database server to complete the request. All communication between the parties happens inside the operating system never leaving the server machine.

Web hosting services offer disk space from a single-server environment by creating new user accounts for each user inside the same operating system. It is typical of single-server environments that users have a pre-defined quota of resources. According to Nebula's (2013) service listing, a basic setup has two gigabytes of storage, 50 gigabytes of monthly bandwidth, and support for the PHP scripting language with a limited set of pre-installed libraries. If the customer needs a database, it has to be bought separately.

Billing in a hosted environment is in most cases based on monthly fees and not on actual usage. The environment is completely maintained by the service provider. Because of that, the users have a very limited access to the underlying operating system and are bind to the software choices made by the service provider. These choices force the customer to use whatever web server, database server or other software the service provider offers.

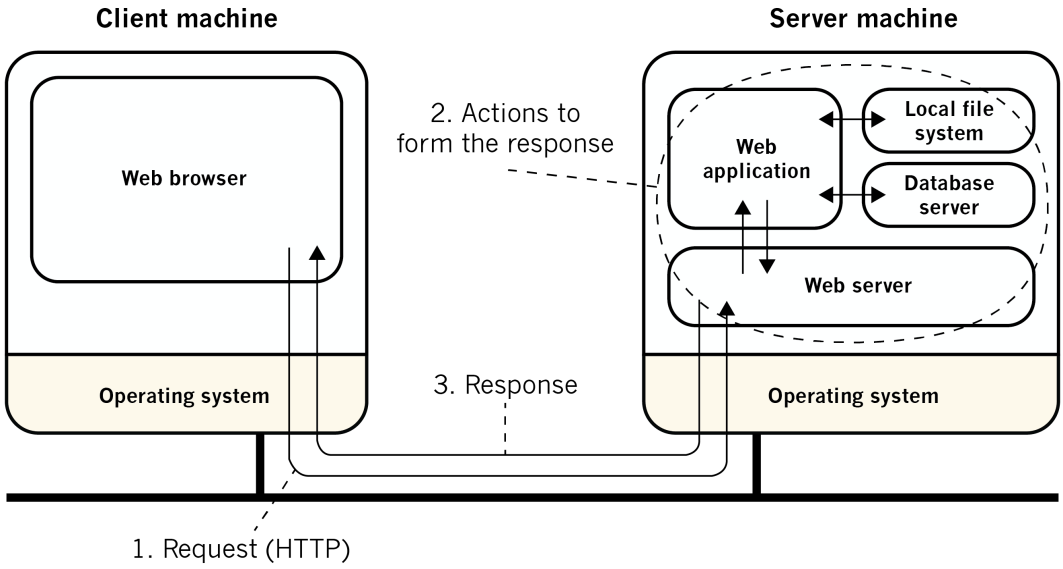

Figure 2.2. *Illustration of the client-server interaction. The client's web browser sends an HTTP request to the web server (1). The web server handles the request (2) and returns an HTTP response back to the client (3). The web server interacts with the web application. The web application may use the local file system and the database server in order to form the response. The illustration is based on Tanenbaum & Van Steen (2007, p. 547).*

Although deploying an application to a single-server environment is inexpensive and relatively easy, its limitations begin to realize when the application's usage grows. For example, increased traffic can exceed the bandwidth limits or the stored data could grow too large for the environment to handle. In a single-server environment, these kinds of changes in application usage can overload the server and cause downtime (Tanenbaum & Van Steen 2007, p. 558).

2.4 Clustered environment

A clustered environment consists of multiple servers, which together form a server cluster, or a multi-server environment. In a cluster, the servers can be, e.g., web servers, file servers or database servers. For example, a simple web server cluster consists of more than one web server and a frontend as seen in **Figure 2.3**. An obvious benefit of using multiple web servers is that multiple servers can handle a larger load than one server. Hosting a web application in a web server cluster gives higher availability and better

fault tolerance. As there are more available resources, the cluster is able to serve a greater amount of concurrent users, for example.

Tanenbaum & Van Steen (2007, p. 93) state that a clustered environment follows generally a three-tiered architecture as seen in **Figure 2.3**. The first tier forwards the incoming request to the second tier, which processes the request. The second tier can interact with the third tier. The third tier could provide the second tier e.g. database or file system services. Those services can reside in separate clusters. In the case of a web server cluster, the first tier forwards incoming HTTP requests to the second tier. The web servers in the second tier are replicas and host the same web application. It is important to notice that web servers in a cluster do not usually communicate with each other and do not know of each other's existence.

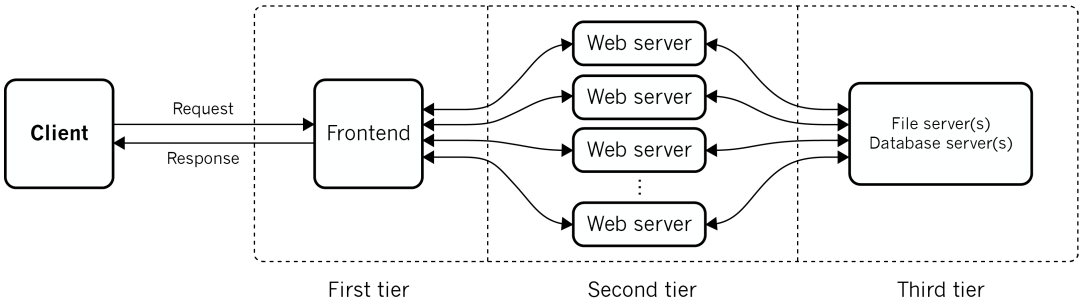

Figure 2.3. *Illustration of a three-tiered web server cluster. The frontend distributes the workload by forwarding incoming HTTP requests to the replicated web servers. Each request can be forwarded to any of the replicas. A web server may interact with other services to complete the HTTP request. The illustration is based on Tanenbaum & Van Steen (2007, p. 93; 2007, p. 558).*

2.5 Concepts

This chapter introduces to basic concepts every web developer should be aware of. First we take a look at virtualization and load balancing in Sections 2.5.1 and 2.5.2, the key concepts behind current cloud service offerings. After those elasticity and scalability are introduced in Section 2.5.3. Sections 2.5.4 and 2.5.5 advance to discussing background processing and content delivery methods.

2.5.1 Virtualization

In cloud computing, virtualization is the enabling technology. Virtualization allows rapid provisioning of virtual hardware resources such as *virtual machines*. A virtual machine is called *a guest,* and the virtualization environment under it is called *a host*. A virtual machine can run any operating system just like a physical computer. An operating system that runs inside a virtual machine is called *a guest operating system*.

According to VMWare (2006), virtualized environments can have *a hosted architecture* or *a bare-metal (hypervisor) architecture*. In a hosted architecture that is shown in

Figure 2.4, the host operating system (e.g. Linux) runs the virtualization layer (e.g. VMWare Player) as an application just like any other application to host a guest operating system. The virtualization layer uses the resources and devices provided by the host operating system (VMWare 2006, p. 4). In a bare-metal architecture shown in **Figure 2.5**, a low-level virtualization layer replaces the host operating system. In this case, the virtualization layer is often called *a hypervisor*. In VMWare's hypervisor appliances, the virtualization layer is operated from a service console to e.g. add or remove virtual machines.

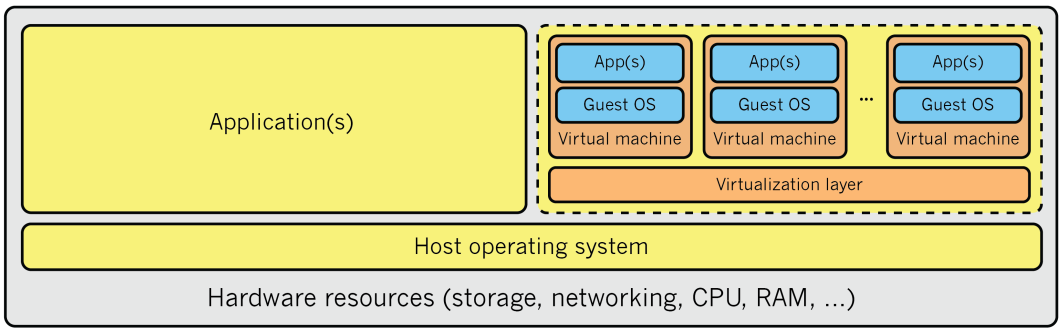

Figure 2.4. *Hosted architecture. The virtualization layer is installed and runs as an application. The virtualization layer relies on the host operating system for physical resource management and device support. (VMWare 2006, p. 4)*

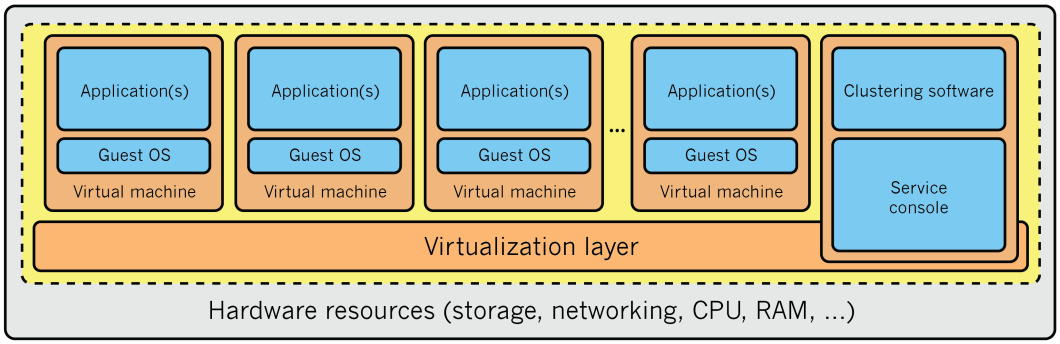

Figure 2.5. *A bare-metal (hypervisor) architecture. The virtualization layer is a minimalistic kernel that takes care of physical resource management. The service console is for managing virtual machines and helper applications. (VMWare 2006, p. 4)*

In this thesis, the concept of virtualization is linked to the provisioning of virtual hardware resources in cloud platform service context. Despite that, virtualization has other appliances, too. For example, the Java Virtual Machine (JVM) allows running Java byte code in many different platforms and operating systems (Java.com 2013). The byte code is universal, but the JVM implementation varies. The Java Virtual Machine is available for the most common platforms and operating systems including, e.g., Mac OS X, Windows, Linux, and Solaris (Java.com 2013).

2.5.2 Load balancing

A clustered web server environment uses a frontend for spreading incoming HTTP requests across the web servers. It is important that the web server cluster appears as a single host to users to make sure that the load balancing is transparent (Cardellini et al. 1999, p. 28). In the case of web applications, the frontend is usually accessed via a single URL. In this context, transparency means keeping the same URL despite that any of the web servers can respond to the user's request. There are several mechanisms for implementing load balancing including DNS-based and dispatcher-based approaches (Cardellini et al. 1999). In this sub-section we focus on the dispatcher-based mechanisms. The concept is described at an abstract level to give an idea what kind of aspects there are in load balancing.

Dispatcher-based mechanisms specify a frontend for dispatching requests. The purpose of the frontend is to balance the load by dispatching requests to those web server replicas that are able to process requests at that time. Tanenbaum & Van Steen (2007, pp. 558–559) suggest several methods for load balancing. According to Tanenbaum & Van Steen (2008, p. 558), the frontend may use web server's load information in making decisions regarding to which web server it should forward the request. In a clustered environment, the frontend can forward serial requests to different web servers. An example of load balancing is shown in **Figure 2.6**. In the example, the frontend dispatches the client's request based on current web server CPU load. The frontend selects the server with the lowest CPU load at that time to handle the request. To make this kind of behaviour possible, the web servers and the frontend need to exchange load information to support the frontends decicions.

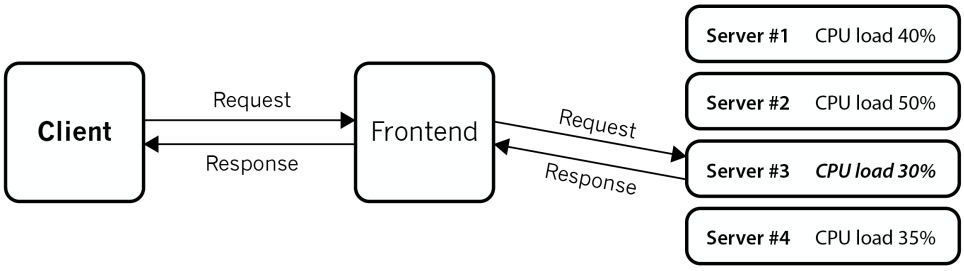

Figure 2.6. *A frontend, or a load balancer, decides where and how to forward an incoming HTTP request. In this example, the decision is made based on the CPU load percentage. The request is forwarded to the server with the lowest CPU load. The illustration is based on Tanenbaum & Van Steen (2007, p. 558).*

In environments with exceptional requirements regarding, e.g., availability and service quality, effective load balancing is a must. By load balancing, the environment can be made more robust and fault-tolerant. A load-balanced environment is different from what developers are used to. Load balancing affects the use of practices and techniques that have been used earlier in a single-server environment.

2.5.3 Elasticity and scalability

Despite the need to scale up when needed, elasticity is not just about the capability of 'massive scalability' that is often understood as just adding more horsepower under the hood (Fardone 2012 & Gartner 2009). Gartner (2009) has changed their earlier definition of cloud computing from "massively scalable IT-related capabilities" to "scalable and elastic IT-related capabilities". The changed definition signals that the scalability of a cloud application, and thus cloud computing in general, has previously caused confusion (Gartner 2008 & Gartner 2009).

Gartner (2009) defines that elasticity is related to the economic model of the application, which allows it to scale automatically up and down. In addition to the Gartner's definition, Mell & Grance (2011) describe the needs a bit further: They underline the rapidness of the scaling procedure and the unlimited amount of resources that appear to the application. Summing the two definitions, elasticity is the capability to react to changes affecting an application in a way that produces beneficial results. For example, those results could be related to preserving a certain service level or just making sure that money is not spent on over-optimistically allocated computing resources.

Elasticity can be achieved by providing scalability. Scalability is required of an elastic application but scalability itself does not guarantee that the application is elastic. For example, if the underlying infrastructure allows scaling to a strict maximum amount of web servers, the application can be considered inelastic in situations where the maximum amount is not enough. A web application can be made scalable most importantly by design and by being deployed to a cloud platform that supports scaling. Briefly put, scaling is the act of increasing or decreasing the amount of computing resources.

Reese (2009, p. 145) describes two scaling strategies: *dynamic scaling* and *manual scaling*. According to Reese, dynamic (automatic) scaling can be divided into *proactive scaling* and *reactive scaling*. Proactive scaling uses a planned schedule to scale up for busy hours and to scale down for quiet hours. Reactive scaling is based on the actual usage of the web application. While proactive scaling is based on a pre-defined scaling plan, reactive scaling is reacting in real-time to the increases and decreases of usage.

To determine whether a web application is elastic or not, one should consider what is required from the application. In an environment where application usage can be predicted, there might be very few requirements regarding elasticity. In the other hand, an application could have a need for almost infinite elasticity. A good example of the former is an intranet application with an amount of users that is well known. For the latter part, an example can be found from the current social media services like Facebook, which seem to have an ever-growing need to scale up. When analysing application's elasticity the application's context and requirements must be kept in mind.

2.5.4 Background processing

Web applications have often needs to implement features that process data or do maintenance without any user interaction. Depending on the application, the processing

may take from seconds to hours or even days. The processing may be triggered on-demand by a user or by a scheduler after specific intervals. At times, such processing can get resource intensive. In the worst case, the processing can render the web application unresponsive or completely inaccessible. To minimize side effects, resource-intensive features can be taken into account in design by leveraging the idea of *threads*.

Tanenbaum & Van Steen (2007, pp. 71–72) demonstrate threads by giving an example of a spread sheet application that can be used to make calculations. The user can execute the calculations from the application's user interface. In a single-threaded application the user interface and the calculations are executed in the same thread. Execution of the application seems to be paused when the calculations are being processed. After the calculations are done, the user gets the results and can continue their work. By separating the user interface and the calculations into separate threads, the user can continue working on other tasks in the application while the calculations are being processed. As Tanenbaum & Van Steen (2007, p. 73) state, threads are useful in structuring large applications into logically separate parts that can be executed at the same time. This helps to understand why a web application can get inaccessible or unresponsive without the distribution of processing.

As we know, the interaction between the client and the web server is based on the HTTP protocol. Traditionally, the web server must respond to any client's request by sending back a complete response at once containing the results. Because of this synchronous behaviour, the user must wait for the web application to complete the request before being able to continue. If there are multiple users requesting the same, the overall performance of the web application is likely to decrease.

Similarly to the spreadsheet application example where the features were separated into threads, features in a web application can be separated similarly by distributing resource intensive work to *background services.* In a clustered environment, the background services are located in the third tier. Work can be distributed to background services by implementing *message queueing*. In a message-queueing architecture, the background services listen to a message queue. The web application places messages to the message queue. The messages contain all information that is needed for the processing of the job. When a background service receives a message, the service processes it and returns the results to e.g. a database. The web application can then check from the database if the job is completed or still running, returning the user the state of the process. This kind of behaviour is called *background processing.*

2.5.5 Content delivery methods

Web applications deliver content to the client. The content consists of, e.g., HTML documents, style sheets and JavaScript files *(static content)*, but also of files that are uploaded by the users *(dynamic content)*. Content delivery methods play an important role when developing an elastic web application. The chosen delivery method may imply that the web application cannot be scaled up without breaking it. Therefore, choosing a

correct content delivery method for each use case is relevant to maintain elasticity and the possibility to scale up.

A web application can store content in a web server's local file system or in an external service. Storing content in the local file system is valid in single-server environments and is usually the preferred method for static assets. In the case of dynamic content, the web server's file system may easily run out of space. In addition to the storage problems, the web application can break if it is scaled up because of load balancing. In a load-balanced environment, the response to the user's request can come from a different web server than that which was used for uploading the content as is shown in **Figure 2.7**. In the figure, the web application tries to locate the uploaded file from the web server #2, which does not have it. If the request gets dispatched to the web server #1, the web application would seem to function correctly. Storing dynamic content in a web server's local file system can be considered as a design flaw in a clustered environment.

The problems caused by load balancing can be solved by using an external service for storing dynamic content. This practice works in all environments and also solves problems related to e.g. storage capacity. An external storage service does not require much effort from the developer as usually a storage service can be accessed via an HTTP API. The benefits offered by external services, such as the Amazon S3 (Amazon Web Services 2013), are obvious. For example, the service's infrastructure allows storing large amounts of content and it takes care of backups and redundancy. It should be noted that the internal implementations of different storage services vary depending on the use-case. Despite the benefits, there are downsides: Using a storage service increases the overall complexity of the web application. To be able to retrieve files from the storage service, files' keys must be saved to a persistent place e.g. to a database. It is important that every web server in the cluster has access to the database.

Figure 2.8 gives an example of using an external storage service for uploaded files and a database for storing keys to the files. Now that uploaded files are stored in the storage service, it allows the web application to scale up. Scaling up was not possible in the example shown in **Figure 2.7** because the uploaded content became inaccessible to other servers in the cluster.

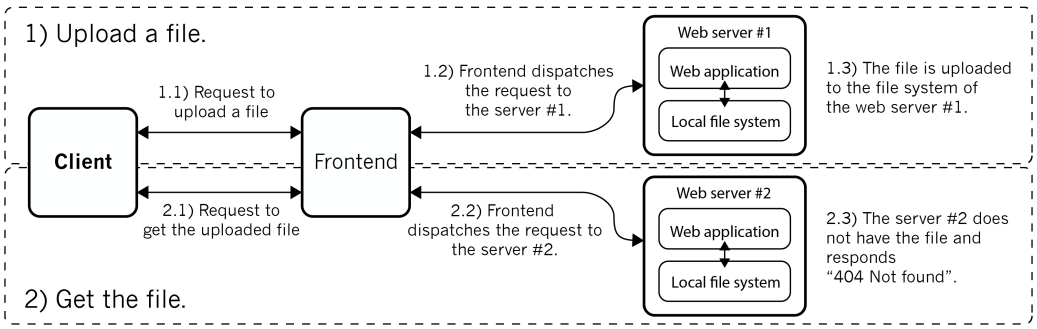

Figure 2.7. *User uploaded content cannot be found if a request to get the content is dispatched to the server that does not have the content.*

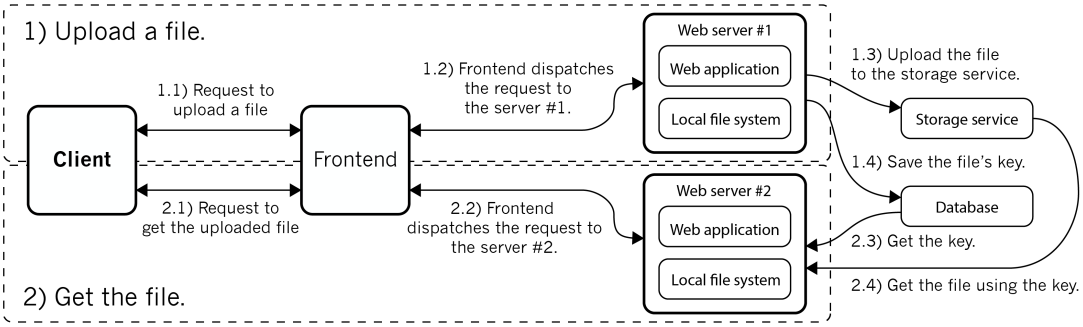

Figure 2.8. *Using a storage service and a database, the load balancing problem can be solved.*

In **Figure 2.8**, the request to upload the file is dispatched to the web server #1. The web server uploads the file to the storage service and after that stores the file's key to the database. When the request to get the file arrives, it is dispatched to the web server #2. The web server retrieves the key from the database. The key is used to locate the file in the storage service. As shown in the example, the file may be proxied through the web server. An optional method is to provide a direct URL to the file in the storage service. In that case the phase 2.4 can be left out. Given the endpoint address to the client, the address to the file can be formed in client's side. This behaviour requires that a read access to the files is granted to public.

3 DEVELOPING A CLOUD APPLICATION

In this chapter we discuss cloud applications and distributed systems, introduce cloud application design principles, the example application and its architecture. In Section 3.1 distributed system's definitions are reviewed and we discuss how a web application must be seen when it is deployed to a cloud platform. In Section 3.2 cloud application design principles are reviewed briefly. Sections 3.3 and 3.4 introduce the Django web framework and the Heroku cloud platform. In Section 3.5 we give a description of the example application and reveal its architecture. Section 3.6 describes how the implementation was modified for deploying to the cloud. In Section 3.7 the example application's cloud implementation is inspected using several attributes: *elasticity and scalability, background processing* and *content delivery*.

3.1 Cloud applications and distributed systems

A distributed system does not seem to have an ultimate, widely accepted definition. Coulouris et al. (1994, p. 34) define that a distributed system or application consists of separate software components, which must interact in order to perform tasks. In cloud application context, the separate software components mentioned by Coulouris et al. (1994) can be understood as other cloud applications, services, or such components that allow, for example, interaction via an API (Application Programming Interface).

Tanenbaum & Van Steen (2007, p. 2) introduce a rather similar definition but they underline that the distributed system must appear as a single coherent system to the end user: actions like scaling the amount processing units (e.g. web servers) must not affect the user experience in any way. The mentioned definitions have a strict technological perspective, which can be hard to understand for a non-technology oriented person. Luckily, Leslie Lamport, a well-known computer scientist, has defined a distributed system in a way that is very understandable from an end user's point-of-view:

> *"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."*
> *– Lamport (1987)*

Lamport's down-to-earth definition applies to today's cloud applications very well. When a web application is deployed to a cloud platform it becomes a part of a distributed system, which consists of multiple service layers. The cloud platform can be built on top of a cloud infrastructure service layer, which in cloud computing is can be considered as the lowest service layer a developer can act on. Due to the fact that the cloud

platform is a distributed system, it is very likely that the web applications deployed to such platforms suffer from problems aroused by the underlying infrastructure. This fact supports Lamport's (1987) definition, as it is nearly impossibly for an end-user to tell where the application binaries are located and why the application failed.

3.2 Cloud application design

As Coulouris et al. (1994, pp. 29–30) state, trying to reach for high-level goals is a challenge in designing distributed systems and applications. According to them, such goals are *performance, reliability, scalability, consistency* and *security*. Sodhi & Prabhakar (2011) give a similar list of goals, or attributes, as Coulouris et al. (1994). In addition to the listed goals, Sodhi & Prabhakar (2011) mention *portability and interoperability*, *maintainability* and *operability and deployment* as non-functional quality attributes of a cloud application.

Sodhi & Prabhakar (2011) introduce two design approaches for cloud application design: *cloud-agnostic* and *cloud-aware.* They compare the design approaches from the listed attributes' point-of-view and try to find which attributes are easier and which are harder to achieve depending on the design approach. In short, they have noticed that a cloud application's design needs to address these attributes in order to leverage any cloud platform's benefits. According to Sodhi & Prabhakar (2011, pp. 1–2), they describe the design approaches as follows:

Cloud-agnostic design approach makes the following assumptions:
- The application does not care about which platform it is deployed on.
- The application does not know whether the resources are virtual or real.
- Design guidelines and techniques are the same as in single-server environment.

Cloud-aware design approach makes the following assumptions:
- The application acknowledges that its resources are shared and virtual.
- The application takes into account that its resources can be scaled and manipulated.
- Design guidelines and techniques differ from those used in single-server environment.

Although many papers (Sodhi & Prabhakar 2011; Zhang et al. 2010; Rimal et al. 2011) focus on large-scale distributed processing applications that are developed with such frameworks as Apache Hadoop, the conclusions are mostly applicable to framework-based web development: the cloud platforms do not guarantee any of the attributes or goals to be achieved automatically after deploying to the cloud platform. Therefore those have to be addressed in the design.

Sodhi & Prabhakar do not discuss directly that the cloud-agnostic approach is actually how an architect acts in a traditional single-server environment. By cloud-

agnosticity, one could also mean an application design that is as portable as possible in a way that it could be easily moved between platforms. When making everything agnostic, it instantly means keeping the application as autonomic and with as few bindings to other systems as possible.

As Sodhi & Prabhakar (2011, p. 4) noticed that reaching for an attribute is likely to make the application design very specific to the chosen cloud platform. Cloud application design could be described as a game of several attributes: when an attribute is doing well, another is left in the shadows. As with the case of cloud platforms, if any platform specific methods are used, it is very likely that portability is about to suffer.

3.3 Django web framework

Django is a web framework written in Python that was born in 2003. According to the Django homepage (2013a), the framework was developed to meet the needs of a fast-paced online newsroom environment. Django implements the Model-View-Controller architecture pattern by providing models, views and templates. Models define the data model of the web application, views handle HTTP requests by generating a response for each request and templates are used for allowing dynamic content in HTTP responses. Django provides useful features for web applications: URL routing, unit testing, database abstraction, user authentication, caching, logging, internalization and localization just to name a few.

A Django project represents a web site that runs under a specific domain. It is organized to one or more Django applications as seen in **Figure 3.1**. In Django language, an application is a reusable piece of code that is as autonomous as possible and implements a well-defined set of features. A Django project should be understood as a container for these building blocks, which together form a web application.

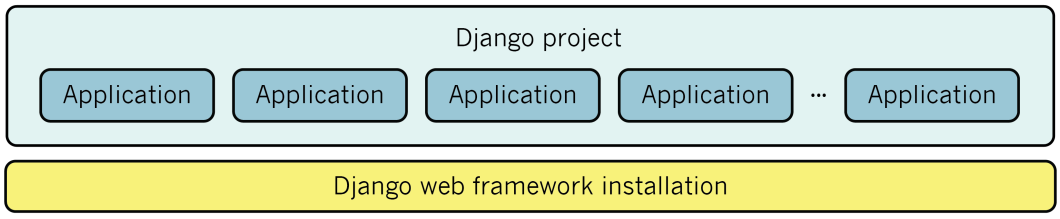

Figure 3.1. *A Django project consists of multiple Django applications. Under a Django project there is an installation of the Django web framework.*

An important feature of the framework is how URLs are routed. In the project's settings, URL routings can be configured separately for applications (see Django 2013d). Specific to Django, a powerful feature is that each application can define its own URL routings. It gives a huge advantage, that there is a possibility to configure URL routings inside an application. In a project that consists of multiple applications, the project's URL routings configuration gets easily messy. As the project's URL routings need only

to specify an endpoint for each application, the URL routings stay clean. **Program 3.1** gives an example of how to include application's own URL routings to a project. The example is taken from the Django documentation (2013d); a snippet from the actual Django homepage configuration.

Program 3.1. *An example of project's URL routings, that defines an endpoint URL for each application and includes application's URL routings. (Django 2013e)*

```
urlpatterns = patterns('',
    # ... snip ...
    (r'^comments/', include('django.contrib.comments.urls')),
    (r'^community/', include('django_website.aggregator.urls')),
    (r'^contact/', include('django_website.contact.urls')),
    (r'^r/', include('django.conf.urls.shortcut')),
    # ... snip ...
)
```

Django favours loose coupling in its design and architecture. In Django, loose coupling means that the different framework layers, such as views and models, do not have to know about each other and are not dependent on each other (Django 2013b). As the current frameworks tend to do, Django also encourages to the Don't Repeat Yourself principle (DRY). By the means of DRY, the amount of code is supposed to stay as low as possible.

As Django is built with Python, a Python shell is available. The Django shell is launched in Django's context for e.g. testing code. In addition to the shell, there are many helpful command-line tools for automating tasks such as creation of Django projects and applications. The tools offer scripts for database maintenance such as scheme syncing and loading of data. The development server can be launched at command-line as well. For all possible commands see the reference Django (2013f).

3.4 Heroku cloud platform

Heroku is a platform-as-a-service that is owned by Salesforce.com, a well-known enterprise cloud service provider. Heroku offers a distributed platform for web applications. The platform takes care of everything related to the environment including operating system updates, virtualization, virtual machine creation, load balancing, and application deployment (Heroku 2013k). A developer using Heroku does not have to take care of anything else than the web application's development. Heroku's servers might be located in Amazon's data centers in the region of Northern America (GigaOm 2012) but there are no mentions about the exact location in Heroku's documentation.

Heroku offers a simple set of features for running web applications. The most advanced configurations a developer can make are how to launch processes and how they are scaled. A Heroku application can have multiple web and worker processes deployed from the same source code. Each defined process can be scaled separately up and down via Heroku's command-line interface. The web and worker processes are called *dynos.*

Each dyno represents a virtual machine that runs Ubuntu version 10.04 when using the Celedon Cedar stack (Heroku 2013e).

Git, a popular version control system, is used for deploying web applications to Heroku. Each Heroku application has its own Git repository. The deployment process activates after every successful push to the repository. Depending on the web application, the process may vary. Deployed Heroku applications follow the three-tiered architecture that was introduced in Section 2.4.

Heroku supports a variety of programming languages and frameworks. The supported languages include Java, Python, and Ruby. Despite the wide support, the languages are supported differently. For example, Heroku was at first a platform for just Ruby applications (Irving 2012). Despite the fact that Heroku tries to be a multi-language platform, it still favours Ruby developers by providing a much more complete documentation for Ruby.

Billing is based on usage and is measured in seconds. Every Heroku application gets 750 hours for free per month. The usage is calculated on actual wall-clock seconds, not CPU time. At the moment, running a dyno costs $0.05 per hour. Usage begins to cost after consuming the free 750 hours. (Heroku 2013f)

3.5 Example application

The example application that is used throughout the thesis is a simple image sharing service. Registered users can upload images to the service and share them via the service. The service is expected to have a rapid increase in demand during its early stages.

Tanenbaum & Van Steen (2007, pp. 3–15) define goals that are required of a distributed system: making resources accessible, distribution transparency, openness and scalability. They state that even if building a distributed system is possible, it is not always wise (Tanenbaum & Van Steen 2007, p. 3), which sets a reason for analysing our example application's requirements in detail.

The service consists of roughly three parts on the application level responsible for *image uploading, image sharing,* and *image manipulation*. On a lower level, the service stores uploaded images into *a file storage service*. In order to keep track of system data, like user accounts and configurations, *a relational database* is utilized.

Image uploading should be possible via a web user interface through a web browser. When uploading an image, the user can give the image a title. After the upload is completed, the image is added to the user's gallery. In the gallery, the user can delete uploaded images.

Image sharing is achieved by allowing public access to the uploaded images via a link to a page containing the image. On the same page, the user's gallery can be browsed.

Image manipulation means resizing uploaded images into several sizes, which is useful when e.g. bandwidth usage needs to be optimized. The image manipulation part

can be configured to produce smaller size images (in kilobytes) if it looks like the bandwidth costs are getting too high.

3.5.1 Architecture

From a software architecture viewpoint, the highest priority requirement is to achieve an available service by allowing the application to scale up and down when needed. The problem is that just deploying a web application to cloud environment does not solve issues related to scalability or does not guarantee any elasticity. This means that the software architecture must address such scalability requirements. Rellermeyer et al. (2009) try to find a solution to the problems of cloud computing and distributed applications by splitting a web application into independent modules. By splitting an application into autonomous modules, one could take advantage of a cloud platforms elasticity related features such as resource scaling. Rellermeyer's (2009) module idea supports also *fine-grained scaling:* if each logical function of an application is a scalable module, the application can be scaled with greater control.

The application is split into three separate layers: *user interface*, *application logic* and *data services*. These layers follow the Model-View-Controller architecture. On the application logic layer as shown in **Figure 3.2**, there are three modules: *image uploading, image sharing,* and *image manipulation.* These three modules form the core of the web application. The modules appear to be logically separate features that are not dependent on each other. For fine-grained scaling, these modules could be separated into different web applications like Rellermeyer et al. (2009) suggest. For the sake of simplicity, only the image manipulation logic is separated from the main web application. The image manipulation logic should be implemented as a background process.

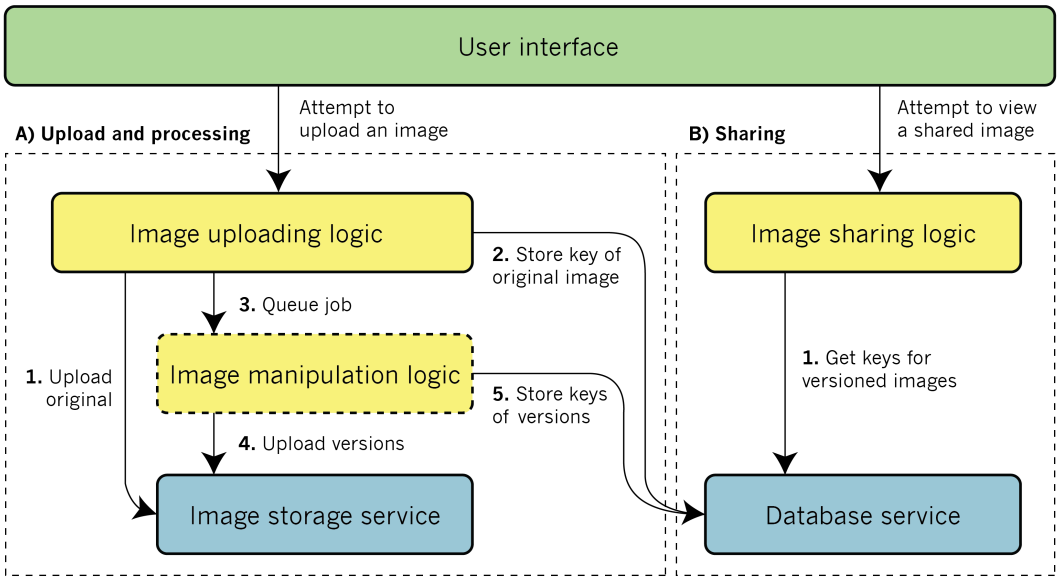

Figure 3.2. *Application flow diagram for A) Upload and processing and B) Sharing that illustrates the interaction between the modules. The dashed line around the Image manipulation logic module means it is a background process. The blue modules are external services.*

Figure 3.2 describes how the application flow works and how the modules interact. The figure is split into two parts: *A) Upload and processing* and *B) Sharing*. The user triggers the part A from the user interface by an attempt to upload an image. The image is uploaded to the service and then stored in the Image storing service. The key of the uploaded image is then stored in the database. Now the Image uploading logic sends a job to the image manipulation background process. The original image is manipulated and the manipulated versions are uploaded to the image storage service and the keys of the versions are stored in the database. When the user attempts to view a shared image, they trigger the part B. The part B is as simple as getting the saved keys from the database and then pushing them back to the user interface. The keys can be used to form the actual URLs of the images.

A cloud platform instantiates virtual machines for running the actual processes. **Figure 3.3** illustrates the infrastructure of the web application. The actual web application is running in *Application instances* and the background processes in *Background processing instances*. Communication between the application and background instances is handled via a message queue service.

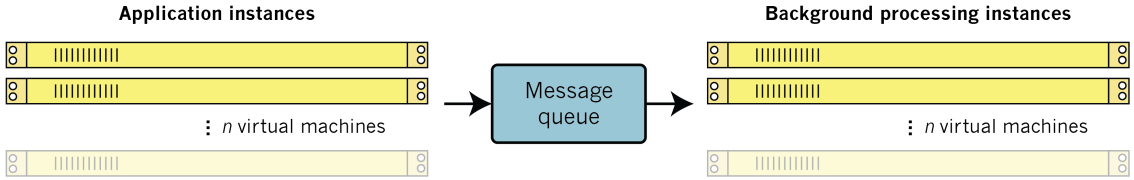

Figure 3.3. *Application instances use a message queue for queuing jobs. The message queue is listened by the background processing instances.*

3.5.2 Development environment

The development environment consists of Django, Virtualenv, and PostgreSQL. Git is used for managing the source code, which is written in Python. Django acts as the core the implementation is built upon. Virtualenv is used for isolating the development environment. PostgreSQL is used for managing the database. These are described in more detail in **Table 3.1**.

Django was chosen for the implementation because of its lightweight nature. Development is quick, as launching the development server or making modifications to configurations do not require massive compiling of the whole application like Java-based applications do. In web frameworks, managing settings should be made easy: Settings in Django are very simple and in one place and format instead of in multiple files and in various formats. For example, the Grails web framework has configurations both in Groovy and XML (Grails 2013b; Grails 2013c).

Developing inside Virtualenv helps to keep track of project's dependencies by isolating the project into its own environment. Using tools like Pip, it is possible to manage modules very easily just by issuing commands such as *pip install*. The rapidness of Py-

thon makes developing a fast experience and installing modules is more or less just copying files from one place to another.

Table 3.1. *Software used in the implementation.*

Name	Version	Description	Homepage
Django	1.4.5	Web framework.	https://www.djangoproject.com/
Python	2.7.3	Python runtime.	http://www.python.org/
Virtualenv	1.8.4	Tool for creating isolated Python environments in development environment.	http://www.virtualenv.org/
PostgreSQL	9.1.4	Database management system.	http://www.postgresql.org/
Git	1.7.10.2	Version control system.	http://git-scm.com/

One of Django's design goals is to achieve loose coupling (Django 2013b). For example, code should be split into separate Django applications (which reside inside a Django project) that can be considered as independent modules. It is then possible to separate the Django applications into another project, for example. Decoupling has been brought also to the URL routing. Every Django application has its own routing configurations in a file called *urls.py*. In the project's URL configuration, one could just simply plug-in a new module and have all its URL definitions working instantly (requires still that all dependencies are installed).

3.6 Deploying to Heroku

This section describes how the Django implementation can be deployed to Heroku. The purpose of this section is to point out how the development version was modified to achieve a successful deployment to the cloud platform.

3.6.1 Modifying the implementation for Heroku

Achieving a successful deployment to Heroku does not require huge modifications to the Django project, but some practices have to be adopted. For example, environment variables are used in Heroku for configuring the application instead of putting passwords and other sensitive information into the source code. Only the project's settings were modified to detect the Heroku environment and to extract database connection details from a database connection string located in the *DATABASE_URL* environment variable in Heroku. In Django projects, the database connection string can be easily placed into Django's database settings by installing *the dj_database_url* module (Heroku 2013d). When the module is installed, credentials to the local development database have to be stored otherwise. By following the environment management practices that

are introduced in Chapter 12 of the Django Book (Holovaty & Kaplan-Moss 2009), a custom environment detection script was created.

Table 3.2 lists the three files that are used in configuring the Heroku application. The *requirements.txt* file is vital to the deployment as it defines which modules the Django project depends on. The *Procfile* is optional. Procfile is used for defining the processes the Heroku application has and how they can be launched. The *runtime.txt* is optional, too. Its only job is to define which Python runtime version to use.

Table 3.2. *Heroku-specific files in a Django project.*

Filename	Location	Description
requirements.txt	Git repository's root	Defines the dependencies required by the Django project.
Procfile	Git repository's root	Defines process names and their sources. (optional)
runtime.txt	Git repository's root	Defines the version of the Python runtime. (optional)

The implementation was built using the four cloud services: *Heroku platform, Heroku Postgres, Redis To Go* and *Amazon S3* (**Table 3.3**). Heroku platform is the core of the implementation on which the cloud application operates. Heroku Postgres is a database service provided as a Heroku add-on. Redis To Go is used in implementing message queuing and is also provided as a Heroku add-on. Amazon S3 is used for storing the uploaded content. Only the Amazon S3 service was bought separately from Amazon Web Services.

Table 3.3. *The cloud services used in the implementation.*

Name	Description	Homepage
Heroku	Platform for deploying the web application.	http://www.heroku.com
Heroku Postgres	PostgreSQL database provided as a service. Provided as a Heroku add-on.	https://addons.heroku.com/heroku-postgresql
Redis To Go	Redis key-value store as a service for implementing message queuing. Provided as a Heroku add-on.	https://addons.heroku.com/redistogo
Amazon S3	Amazon Simple Storage Services is a data storage service. Bought directly from Amazon.	http://aws.amazon.com/s3/

In addition to using the mentioned cloud services, the implementation depends on several third party Python libraries. The libraries are *Boto* (Boto 2013) and *Redis Queue* (Redis Queue 2013). The Boto library can be used to interact with any of the Amazon

Web Services. In this case only the functionality regarding Amazon S3 was used. Redis Queue is a simple library for queueing jobs for processing using background workers (Redis Queue 2013).

3.6.2 Git and the Heroku build process

Heroku deployments are made with Git. Git's *push* command pushes a given branch from the local development repository to the master branch of the remote Git repository in Heroku. When the push happens, Heroku begins the build process. An example of this behaviour is given in **Program 3.2** where a new version of the Django application is pushed to Heroku. Git deployment to Heroku works using Git remotes. The correct Git remote can be configured by *heroku git:remote*. The command sets the remotes always for the current repository.

Program 3.2. *Git output after issuing a push command. In this example, the current branch is used instead of the local master branch.*

```
(venv)$ git push heroku +HEAD:master
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 371 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
...
```

The build process detects which programming language and which web framework is used. Based on those the build process automatically chooses *a buildpack* as shown in **Program 3.3**. A buildpack is a set of scripts that knows how to build the detected environment. Buildpacks can be manually written when something very specific is wanted to achieve, for example, when there is a need to download modules from a private source. Heroku offers buildpacks for several web frameworks.

The Python buildpack expects the file *requirements.txt* (see **Table 3.2**). The file can be created using the Pip tool in the root of the project's Git repository by running the command *pip freeze > requirements.txt.* When the dependency detection is successful, the dependencies are downloaded automatically and installed via Pip as shown in **Program 3.3**. In addition, it is possible to define which version of Python runtime to use by defining it in *runtime.txt* (see **Table 3.2** and Heroku 2013c).

Program 3.3. *The buildpack process has detected a Python application and installs the dependencies listed in requirements.txt. The project's Procfile located in the root of the repository is used for discovering the process types that the application implements. (Continued from* ***Program 3.2****.)*

```
-----> Python app detected
-----> Using Python runtime (python-2.7.3)
-----> Installing dependencies using Pip (1.2.1)
       Cleaning up...
```

```
-----> Discovering process types
       Procfile declares types -> imageprocessor, web

-----> Compiled slug size: 31.4MB
-----> Launching... done, v26
       http://immense-ridge-3848.herokuapp.com deployed to Heroku

To git@heroku.work:immense-ridge-3848.git
   f85b674..14e9111  HEAD -> master
(venv)$
```

3.6.3 Scaling up and down

After the dependencies are solved, the buildpack looks for a *Procfile* (see **Table 3.2**). The *Procfile* defines how the application is deployed meaning how the web workers are deployed and how to launch background workers. **Program 3.4** shows the *Procfile* used in the example implementation in which the web server process is named 'web' and the background worker process is called 'imageprocessor'. The 'web' process is launched automatically when a new version of the application is deployed to Heroku, but the background workers need to be manually started. In short, a *Procfile* defines the processes the Heroku application can have.

The defined processes can be scaled separately. For example, scaling the amount of web processes to three instances and worker processes to two instances, is achieved by running the command *heroku ps:scale web=3 imageprocessor=2*. In **Program 3.5** we show the process listing of the Heroku application right after the command. From the listing we can see the status of each process. The three web processes have 'starting' as their status, which means the processes are not yet able to handle incoming HTTP requests. If an application error happens, the status shows 'crashed'.

Program 3.4. *A Procfile that configures how to launch the Gunicorn web server and the worker process. These roles can be scaled separately.*

```
web: gunicorn imageshare.wsgi
imageprocessor: python manage.py rqworker default
```

Program 3.5. *Heroku application's process listing showing details for three web processes and two background processes right after scaling up. In this example, the web processes are still starting but the background processes are up.*

```
(venv)$ heroku ps
=== imageprocessor: `python imageprocessor/worker.py`
imageprocessor.1: up 2013/03/05 22:19:20 (~ 1s ago)
imageprocessor.2: up 2013/03/05 22:19:21 (~ 0s ago)

=== web: `gunicorn imageshare.wsgi`
web.1: starting 2013/03/05 22:19:17 (~ 4s ago)
web.2: starting 2013/03/05 22:19:18 (~ 3s ago)
web.3: starting 2013/03/05 22:19:18 (~ 3s ago)
```

To see the load-balancer in action, we can see the logs with *heroku logs -t*. The command opens the Heroku application's log for viewing in real-time as a stream. The log shows on a per request basis how the requests get served and which HTTP status the responses are given. The output after scaling the web process to three dynos is shown in **Program 3.6** where we can see that the web server actually varies between the two requests: The request #1 gets served from the dyno "web.1" and the request #2 from the dyno "web.2". Both requests are completed successfully.

Program 3.6. *Static assets come from different dynos after scaling up.*

```
-- Request #1
2013-04-09T21:26:00.313467+00:00 heroku[router]:
      at=info method=GET path=/static/browse_styles.css
      host=immense-ridge-3848.herokuapp.com fwd="85.194.225.199"
      dyno=web.2 connect=1ms service=4ms status=200 bytes=255

-- Request #2
2013-04-09T21:25:44.691867+00:00 heroku[router]:
      at=info method=GET path=/static/browse_styles.css
      host=immense-ridge-3848.herokuapp.com fwd="85.194.225.199"
      dyno=web.1 connect=2ms service=5ms status=200 bytes=255
```

3.6.4 Integrating with Amazon S3

To implement the image uploading, an external storage service was used. In this case the Amazon S3 service was selected. Amazon S3 provides an API for accessing the service from applications written in several programming languages including Python. For using the service, a third party Python module called Boto was installed to the Django project. In Amazon S3, there are *buckets* and *objects*. A bucket is a container for objects, and it is specific to the Amazon Web Services user account. An object represents a file that is uploaded to the service. Objects have several options for restricting access to them. In the example application, each uploaded object gets public read access.

Whenever a user uploads an image from the example application, the main web application takes care of uploading the original image to the S3 bucket. After the upload is completed, a job is placed to the queue. The job is then picked up by a worker process, which uploads the manipulated versions of the image to the S3 bucket.

3.6.5 Message queuing and background workers

The architecture of the example application was designed to utilize a message queue in distributing workload to background workers. For implementing a message queue, a backend is needed to store the queue. As a platform, Heroku offers multiple third party add-on solutions for message queuing. From the available options, Redis To Go was

selected to provide a Redis instance. A quick research revealed that a Redis instance could be used as a message queue using the Python module called Redis Queue. Redis Queue has methods for connecting to the Redis instance, placing jobs to the queue and listening to the queue. For implementing this kind of message queuing, there is a tutorial available in Heroku's documentation (Heroku 2012b).

Implementing a message queue and a background worker with Redis Queue was simple by following the tutorial. When the actual image manipulation logic was implemented in the background workers, the workers needed to get access to Django's settings and context. To allow the worker script to access the database using Django's object relational mapper, the *django-rq* module was installed. As the implementation was first done without including the context in the worker script, the worker script needed refactoring. Django-rq is a third party module that acts as a wrapper for the Redis Queue module.

The worker script implements the image manipulation logic. The implementation uses the Python Imaging Library (PIL). Unfortunately, there are not many imaging libraries available for Python. Thus PIL was selected despite that the latest version of PIL was released in November 2009. In addition to the inactive development, PIL has bugs related to JPEG support. To enable the JPEG support in Heroku, PIL had to be compiled from a modified source code available from Bitbucket.org (Stackoverflow.com 2012).

3.6.6 Implemented architecture

By using the cloud services Amazon S3, Heroku, Heroku Postgres and Redis To Go, the architecture formed into what is shown in **Figure 3.4**. In the figure, Heroku web dynos and worker dynos run the actual Django project. Amazon S3, Redis To Go and Heroku Postgres were provided as a service requiring zero or just a few steps to fully utilize them as parts of the implementation. The Django project is split to run in the Heroku web and worker dynos as described in earlier sections. Redis To Go implements the message queue that is listened by the Heroku worker dynos. The Heroku web dynos are used for sending jobs to the queue. It is worth noticing that the client may access any public resources in the Amazon S3 via a URL provided by the web application. The URL to an image can be formed by fetching the image's key from the database. The full URL can then be delivered to the client's web server in an HTML document.

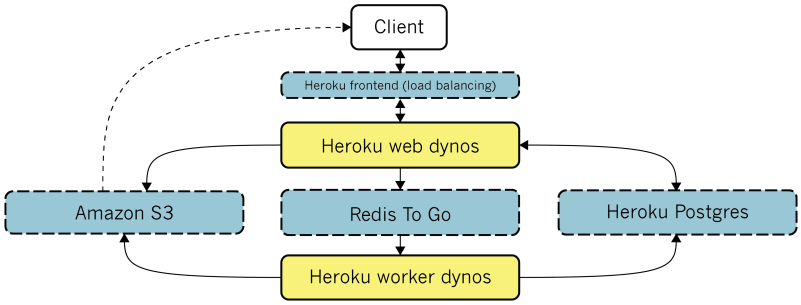

Figure 3.4. *The implemented architecture. The dashed line between the client and Amazon S3 illustrates that the client can retrieve public files from S3 directly without proxying them via the web dynos.*

3.7 Attributes under inspection

In this section we analyse the Heroku deployment by taking a look how *elasticity and scalability, background processing* and *content delivery* were implemented. By the analysis we can find how the attributes were achieved to be able to tell what should be improved.

3.7.1 Elasticity and scalability

Considering the requirements of the example application, automatic scaling is not currently available in Heroku. To maintain elasticity the lack of automatic scaling means, that the Heroku application must be monitored manually via logging and monitoring services. Reacting to changes in application usage has to be done manually from command-line using the Heroku command-line utility. As Heroku does not currently support automatic scaling, it is likely that many Heroku applications have too many web and worker dynos running at certain times. In the case of the example application, keeping unnecessary resources up and running would instantly mean waste of money. If it is vital to make sure that the paid resources are never hanging loose, Heroku might not be the platform of choice.

Reese's (2009) scaling strategies are hard to implement in Heroku. To implement such strategy, a third party service or a custom implementation is required. For (experimental) automatic scaling purposes, there are several services available as listed in **Table 3.4**. There is also a Python wrapper available for using the Heroku API (Heroku 2012a). The wrapper can be used for managing a Heroku account via a REST API. Using the wrapper the web application gets access to the platform level and thus one could implement their own automatic scaling scripts or other optimizations.

Table 3.4. *Third party implementations for achieving automatic scaling in Heroku.*

Name	**Homepage**
Autoscale-bot	https://pypi.python.org/pypi/autoscalebot
Hero Scale	http://heroscale.com/
Hirefire App	http://hirefireapp.com/

Django's practices for code decoupling clearly support achieving elasticity. By making modules as independent as possible, it helps splitting the web application into separate projects. These projects could then be deployed as separate Heroku applications acting as services to each other.

3.7.2 Background processing

As a framework, Django does not suggest any practices for background processing. Despite that, there are several Python modules available for Django that can be used for implementing background processing. By separating the image processing into a back-

ground process, it became possible to scale the application parts in greater detail. The separation distributes the workload from the main web servers to the background, which leaves the web servers with more resources. As Django does not have any default methods for implementing background processing, it took time to find a suitable method for achieving what was desired. Although the implementation became exactly what was designed, most of the time was spent on researching how one could implement such processing within Django. The lack of default practices for such a common use case is likely to push developers into creating hacky implementations.

The current implementation of the background workers is not cost effective or robust in the long run. As the workers run if there are no jobs to process, they cause unnecessary bill. To prevent this behaviour, the Heroku API could be used to kickstart workers like described in Stoelinga's (2012) example. The implementation cannot be considered robust at this point, because the worker scripts implement the image manipulation logic using the modified version of the PIL library. It is a matter of time when the version of PIL is e.g. removed from Bitbucket.org or otherwise modified.

The need to perform background processing is vital in allowing a web application to scale up and down by distributing heavy processing to the background. The separation of pure user-interface and backend features requires analysing which features can be executed in parallel and which must be completed before returning any response to the client. In the example application, the separation was taken into account early when designing the architecture.

3.7.3 Content delivery

Django's default method for storing uploaded files to a web server's file system cannot be used in a load-balanced environment such as Heroku. Using the default method ends up in the same problem that was described earlier in Sub-section 2.5.5. In addition, Heroku's dynos empty the file system after restarts. The default method might seem to work when running the web application with one dyno, but everything user-uploaded would disappear after a while when Heroku decides to reboot the dyno. The behaviour cannot be avoided, as Heroku reserves the right to move running processes (dynos) in their infrastructure from a machine to another. For properly delivering user uploaded content, Amazon S3 was used. Using Amazon S3 requires that all bucket names are globally unique. Bucket name clashing can be avoided by choosing a long and random bucket name that is e.g. based on the application name. Once the bucket name is reserved, it stays reserved until the bucket is deleted.

Static content was included inside the Django applications. The practices used by Django seem confusing at first, but seem valid after a while. Django has a collector script for collecting static files from the project's Django applications. Each application holds its static files under a directory named "static". The collector script rewrites files with the same names resulting in that only one of them is accessible. The problem can be addressed by naming static files e.g. *appname_styles.css* to avoid same names.

4 PITFALLS AND WORKAROUNDS

This chapter describes the pitfalls that were encountered in the development of the example application. The pitfalls are described in detail in order to provide a comprehensive understanding. In addition, each pitfall is given a workaround when applicable. The pitfalls are divided into three categories: *known pitfalls, pitfalls in application design and implementation* and *pitfalls in development process.* The pitfalls are discussed in Sections 4.1, 4.2 and 4.3. The Section 4.1 is formed based on a set of pitfalls that were found in the literature to provide a basis for understanding which parts of a distributed web application can contain pitfalls. Sections 4.4–4.6 analyse the found pitfalls to provide insight into avoiding them.

4.1 Known pitfalls in distributed systems

A distributed system can get very complex. There are many points of failures that can cause problems in any phase of web application's lifetime. The problems can be related to any actor that is somehow participating in the environment. For example, electricity outages, thunderstorms, or construction workers near a core router can cause downtime to parts of a distributed system. Depending on the fault tolerance of the system, the system may be rendered partially or completely inaccessible. Although risks can be managed and minimized, but it is impossible to create a problem-free environment as seen in the case of Amazon Web Services (AWS) and the outages they have suffered (GigaOm 22.10.2012).

First-time developers make a set of assumptions about the working environment. According to Tanenbaum & Van Steen (2007, p. 16), the most common assumptions developers make include the following in no particular order:

(1) The network is reliable.
(2) The network is secure.
(3) The network is homogenous.
(4) The topology does not change.
(5) Latency is zero.
(6) Bandwidth is infinite.
(7) Transport cost is zero.
(8) There is only one administrator.

In the case of the AWS data center outages, replicating the web application to multiple data centers could have helped in minimizing downtime and the impact affecting

end-users. As listed in GigaOm's (22.10.2012) article, one of the services was Heroku. This means that the fault-tolerancy in cloud platforms has been abstracted from the developers deploying their web applications to the cloud platform. If the cloud platform does not take such risks into account that a whole data center could go down, it is again a matter of time when it happens.

The Tanenbaum & Van Steen's (2007) list gives a good starting point for understanding, which parts of a web application can fail in a distributed environment. On a cloud platform, developers cannot affect certain things that are e.g. related to a very low-level of the underlying platform. Because of the possibility to affect to things, there obviously is a lack of motivation and need to understand such. But when lowering the cloud abstraction level to IaaS, and building a web application from scratch using for example the AWS Elastic Computing Cloud nodes the Tanenbaum & Van Steen's (2007) list becomes completely valid.

4.2 Pitfalls found in design and implementation

This section introduces the identified pitfalls in application design and implementation phases. The titles of the sub-sections give a brief description of the pitfall. If applicable to the pitfall, a workaround is given.

4.2.1 Saving dynamic content using default methods

The example application allows users to upload their images to be shared. In the case of the example application, the user-uploaded content cannot be stored using the default methods that are found in the Django's documentation. The default method stores files in web server's local file system.

In the environment provided by Heroku, each web and worker dyno has an ephemeral file system. An ephemeral file system is non-persistent: In Heroku, a dyno's file system is reset at every dyno restart. Resetting a file system means that those files get deleted that are not included in the Heroku application's Git repository. If the dyno's file system would be persistent, it would mean problems with locating uploaded files as described in the examples of Sub-section 2.5.5.

In any environment, a need to serve notable amounts of content requires a proper storage solution. In a single-server environment, the pressure is on the web server's capability to serve both the web application and the content when using the local file system. In addition, the local file system must have proper backups in order to provide fault tolerancy. In a cloud environment, the local file system is not an option, which means that using an external storage service is a must. The storage service could be one's own or a service bought from a reliable service provider. The idea is to have the content stored in a persistent way. To overcome this pitfall in the example application, the Amazon S3 file storage service was used.

4.2.2 Authentication and in-memory sessions

Authentication is a key feature in many web applications providing any kind of personalized service or any other service that handles sensitive information, for example. These protection mechanisms are based on the idea of *sessions*. A session begins when a user logs in and it ends when the user logs out. Whenever the user navigates in the web application its session is validated before any request is made, because of the stateless nature of the HTTP protocol. The behaviour requires that the user's session is stored somewhere in a persistent place.

In a single-server environment, the server's memory is often used for storing the session information. This is the default behaviour in the Django web framework, for example. When moving to a cloud platform, the default method breaks. Problems caused by the method results in that the user gets unexpectedly logged out due to load balancing. The scenario is very similar to the uploading example described in Subsection 2.5.5. In this case, the resource that cannot be found is the session information that resides in either of the server's memory.

To overcome this problem in a cloud environment, the session information has to be placed into a storage that is accessible to every web server in the environment. Such storage methods are for example a database and a cookie or a cookie alone. Using a database and a cookie, the correct session can be found from the database using the information in the cookie. Always when a cookie is used, it exposes to security threats such as *eavesdropping* and *masquerading*. Eavesdropping is the act of obtaining copies of messages without permission and masquerading means sending or receiving messages using someone else's identity without their authority (Coulouris et al. 1994, p. 480). In many cases, just by owning the cookie one can get access to the service where the cookie is from. As the cookie-based solution introduces new problems, one should encrypt the connection between the client and the server using SSL encryption and signed certificates. Django provides a feature called Cross Site Request Forgery protection (CSRF) that can be used to strengthen the web application against the described security issues (Django 2013g).

4.2.3 Framework's documentation ignores cloud environment

Documentation was the main source of information when developing the example application with Django and Heroku. As a cloud platform provider, Heroku offers tutorials and small examples for running Django-based web applications on Heroku. The tutorials cover a very basic set of functionality and are a good starting point. On the other hand, Django's documentation does not currently take cloud platforms or distributed environments into account at all. This leaves many questions unanswered regarding the use of Django in a cloud environment. For example, Django provides methods for storing sessions both in the database and in Django's caching system. Despite the options, it is not mentioned that the cache-based method might not work in a load-balanced environment. Because of that, choosing the database-backed sessions requires specific

knowledge on when it should be used. However, Django's database-backed method can be used in a cloud environment as-is.

The lack of information results in that developers often have several alternatives of which some or none function correctly in a cloud environment. Such forces the developers to test each of the alternatives to be able to tell which of them can be used. Considering that cloud platforms are a rather new deployment option, not many developers have experience in developing web applications for the environment: Therefore, testing of alternatives and finding the tool for the job can be a troublesome task.

4.3 Pitfalls found in the development process

This section covers pitfalls encountered in the development process of the example application. As in the previous section, sub-section titles describe what the pitfall is about.

4.3.1 Use of randomly chosen plug-ins

Web frameworks do not offer all of the features for free that a decent web application implements. Therefore the wide selection of third party plug-ins comes in handy. There are robust options available for some of the most common tasks such as JSON and XML parsing, and caching. But there are also very bad options available. The problem is that how a developer can tell if a plug-in is better than its rival or why the plug-in cannot be used.

As described earlier, the example application uses the Python Imaging Library (PIL). Unfortunately, PIL falls into the category of suspicious plug-ins. As the plug-in is installed from a Bitbucket.org repository that is hosted by an unknown person, it cannot be guaranteed that the repository will remain as it is for the next year, for example. Using a modified version of a discontinued library includes many possible points of failure, despite that the plug-in works in the current implementation. For example, a likely situation is that the repository gets removed by the author and thus makes the source code unavailable. In a similar fashion, a future update to the cloud platform or Python runtime itself could break the plug-in. As the plug-in's development seems to be paused or discontinued, the future of the plug-in cannot be considered very bright.

Often the situation is that the available options are all far from the best. In the case of PIL, the plug-in was the least bad of nearly zero other options. In such cases, developers should take care of documenting the possible risks associated with the use of the plug-in.

4.3.2 Poor environment management

Before a new feature gets published, it has to be developed, tested and then published to the production environment. During these three phases, the same web application runs in different modes with different settings in different environments. The settings are often related to logging, error reporting, and overall performance. In development

mode, the web application might give the developer very detailed error reports for debugging purposes. In production mode, detailed error reports could reveal too much about the web application and its vulnerabilities. Therefore, the level of error reporting is usually decreased to give only a very limited set of information when encountering errors in production.

In Heroku, developers must build separate environments for allowing a Heroku application to run in different modes. It is not possible to copy settings and add-ons from a Heroku application to another. This means that the developer has to keep any separate environments manually in sync. The problem has been noticed in the official Heroku documentation but there are no guaranties such support would be included (Heroku 2013h). Heroku (2013h) suggests that different environments for this purpose should be handled using Git remotes, which means having one source code but several Heroku applications linked to it.

In the case of larger web applications, there might be a need to have also a staging environment. A staging environment can be used for getting approval for new features from the customer. When the version in the staging environment is approved, it should be possible to switch the staging environment to production. This kind of action does not have any support in Heroku.

As Heroku does not have multiple environments for each Heroku application by default, custom methods have to be used in order to achieve the feature. The lack of such feature causes the need for making unnecessary manual configurations. Manual configurations management is in most cases prone to fail. Different projects have different requirements; if the feature is needed, Heroku might not be a good choice.

4.3.3 Over-optimistic approach to cloud services

As reviewed in Chapter 2, cloud computing has been understood to provide massive scalability and infinite computing resources to customers. Adding the marketing hype to the false assumptions, it is not hard to find non-technical and technical people who would put everything to the cloud. The over-optimistic approach to cloud computing is most likely formed because of the naïve advertising. As the concept of cloud computing is rather new and not very well known, cloud services are often offered as a solution by many. But the problem is that not many really know what cloud services should be used.

Due to the abstraction provided by cloud services, the possibilities of the service may stay unclear. In the case of cloud platforms, Heroku provides a good service to start with. Such features as background job scheduling may be hard to achieve due to the Ruby-favouring nature of Heroku. By default, the scheduler functionality in Heroku is implemented for Ruby based scripts using Ruby (Heroku 2013i). Having more than one programming language in a project makes the development quite complicated.

Limitations of cloud platforms may not allow whatever kind of connections to the web application. For example, web applications that need to implement integrations using VPN connections can be deployed only to a platform that supports VPN connec-

tions. As in the case of Heroku, such connections cannot be established. The limitation can be explained by the relationship formed by the amount of configurability and the abstraction level as seen in earlier in **Figure 2.1**: The service that Heroku provides places to an abstraction level where the developer cannot configure anything that resides under the web application itself. This means things related to networking facilities, replication, backups of the Heroku application's Git repository and so on.

By knowing the possibilities a cloud service offers, one can avoid struggling with the service in the future. By investigating any available examples, documentation, tutorials and implementations, one can find how the service performs in various cases. If such proof cannot be found, it might not be a good choice to be the first one to test it.

4.4 Summarizing the pitfalls

In this section we analyse the found pitfalls create a basis for further discussion. The pitfalls were split into two categories: *design and implementation*, and *development process*. Despite the categories, similar root causes can be found why the pitfalls exist: As Reese (2009, p. 67) expresses it, "web applications share a similar general architecture – and architecture either makes or breaks an application in the cloud."

A web application handles information related to the application's state. If the state is handled in a way that locks the state or makes it inaccessible, the web application may not function correctly. In a distributed system the application's state should be stored in a way that is reliable and accessible to all servers that need the information. Not always the state has to be stored for a long period of time. As noticed earlier, loosing grip of e.g. user sessions can render the service pretty useless, as the user does not stay logged in while being bounced between web servers by the load-balancing tier.

Django is rather well prepared for the cloud environment but this has not been written to the documentation. Despite the fact that Django provides features that are vital in cloud environment, it does not advice the developer in any way when to use such features. The documentation does not take into account what kind of features should be used in a cloud environment. Although a framework's documentation is not expected to be a design guide for cloud applications, the documentation could still convince the developer that the cloud environment has been taken into account in the framework's design. There are multiple points of failures where a web framework can go wrong and knowing that the framework survives those is valuable knowledge.

The avoidable pitfalls were encountered due to the lack of relevant documentation, unsupported design choices, and use of features that do not work in a distributed environment. The lack of documentation is more a Django's problem than Heroku's, as it is the developer's responsibility to use the web framework correctly. It is very obvious that developers have to know exactly what they are doing to avoid causing extra work especially in cloud application projects. Web frameworks do not offer help regarding the chosen environment, which leaves the developer on its own. These pitfalls and the root causes signal that designing a cloud application or a web application for the cloud

requires up-to-date knowledge about the environment to be able to spot problems early. The move to the cloud environment is the key factor that causes problems to the developer, not the actual development.

By the analysis of the pitfalls, we can extend the Reese's (2009) idea a bit further by giving more responsibility to those who do the implementation, the developers. In the end, the developer's choices and decisions are a key enabler in being successful in the cloud environment. A step forward would be to find a way, how to guide developers to the right direction in the puzzled world of web frameworks and cloud platforms.

4.5 Coping with pitfalls

As pitfalls exist in all areas of web applications from development to maintenance, nobody can completely avoid them. Despite that, some of them can be predicted. To be able to know of what kind of pieces a web application is formed of, there has to exist documentation.

In order to overcome problems, developers have to create workarounds. In a web application project with many workarounds, the development gets easily hard to manage. Workarounds tend to create more work when the web application evolves, if they are documented poorly or in the worst case, not at all. To avoid such, workarounds should be documented at least in source code. Without knowing workarounds, the web application can be broken when updating plug-ins or the underlying web framework. That features are left without documentation is not just a problem with workarounds: It applies to every feature of a web application. For example, a custom fix to a third party plug-in makes it hard to update the plug-in without losing the fix. Similarly a feature that is bound to the environment may break when the environment is changed. In both situations, the pitfalls can be avoided by knowing that such can exist.

Reese (2009, p. 100) lists as a security issue that a cloud provider can face bankruptcy or take its business in another direction. The latter has realized in the case of Heroku, which during its early years dropped out features in search for their core business idea (Irwing 2012). Cloud providers may seem at first trustworthy. Their future visions may be very positive and advertisement promises a relationship for life between developers and the service. In the case of provider failure due to any cause, one should have a backup plan at least for the most business critical applications. For example, it might seem that everything is working flawlessly when a cloud platform provider promises to e.g. take backups and do everything related to maintenance. Cloud providers are still businesses as any other company: they do have rivals and they do have real risks. By minimizing exit costs, one could be prepared for problems that are caused by such scenarios. For example, in the case of cloud service provider's bankruptcy, one could port the web application to another platform. Depending on how portability was taken into account in the application's design, the porting may be either easy or very complicated.

Things that seem to be good at the first sight propably are not in the long run. In avoiding pitfalls, it is important to have a pessimistic approach to new software. In web development, developers find good-looking plug-ins great. For a plug-in to be good-looking, the plug-in needs a well-designed homepage with bright colors and a couple of examples that convince the potential downloader of the plug-ins ease of use. For example, such can be found in JavaScript development where many developers feel a need to create their own JavaScript libraries. The amount of available JavaScript libraries is huge, of which many do the same tasks in a bit different way. As there are few big, commercial players in creating software for web development and the software is mostly open source, the problem is that there is really nobody controlling the overall outcome. This has resulted in so-called "opinionated software" by which developers try to push their own ideas and practices forward to a wider audience.

4.6 An analysis method for choosing plug-ins

To identify pitfalls, an analysis method was created during the writing of the thesis. The analysis method is based on a simple set of questions. The method has been designed for analysing plug-ins by finding their strengths and weaknesses from several viewpoints. The checklist consists of questions related to *overall quality*, *future views*, *available support* and *popularity*. After the analysis has been done for the candidates, the results are compared to each other to find the best candidate. The analysis starts from finding answers to the questions found in **Table 4.1**.

The questions related to overall quality try to find if the plug-in can be considered mature and how the plug-in has been designed for the problem the developer is about to solve with it. The analysis takes into account *version numbering* and *known issues*. A version number tells about the maturity of the plug-in; generally a greater number means established practices. A list of known issues, or a lack of such, gives an idea of for which purposes the plug-in can be safely used for. To be able to know whether the plug-in supports the development, the documentation and the example should be analysed from the point of view of the use case it would be used in.

Table 4.1. *A set of questions used in the analysis.*

Question	**Area**
1) Has the plug-in's development been regular and is it currently active?	Future views
2) In which version the plug-in is currently and is a list of known issues available?	Overall quality, future views
3) Is the plug-in well documented and the documentation gives relevant examples?	Overall quality
4) Does the plug-in have an active community?	Support, popularity
5) Does the plug-in have an issue tracker? If yes, do the issues get solved?	Support, future views

Future views tell about the plug-in's development and how likely it is that issues get fixed. The future views are analysed by looking at how regularly the plug-in has been developed before, and if there is an issue tracker, how many open issues and solved issues there are currently. A regularly developed plug-in is likely to get regular updates in the future as well. An issue tracker is an even better way to determine how professional the development is.

To get an overview of what kind of support is available, the activity of the community around the plug-in is analysed. The available support is strongly related to the popularity of the plug-in; a popular plug-in most likely has a large community. In the world of web development, it is very common to ask help from web sites such as Stackoverflow (http://www.stackoverflow.com). A quick research reveals what kind of problems other developers have had with the plug-in and are there any workarounds available for the problems. The amount of related search results can be used as guidance in determining how popular the plug-in is.

In order to compare the analysed plug-ins in greater detail, each of the questions can be scored from 0 to 5 points. Giving a full score should be avoided to keep the scale open, meaning that there will always be a better plug-in at some point. Summing the given scores forms the overall score for the analysed plug-in. The greater the overall score is, the better the plug-in suits the need. If the developer wants to give more weight for some questions, the scores of such questions can be multiplied by e.g. a factor ranging from 0.5 to 1.5. It is important that the analysis is done from the viewpoint of the web application where the plug-in is going to be used in. By doing so, one can take the current working environment into account.

5 TOWARDS A BETTER CLOUD EXPERIENCE

This chapter discusses how to improve web application development in order to help developers with moving from a single-server environment to the cloud environment. Section 5.1 introduces development related improvements from a technical point of view. In Section 5.2 web application portability and exit costs are discussed. Section 5.3 goes through the requirements of an elastic cloud application in the form of cloud platform features.

5.1 Improvements to cloud application development

Often there is no relevant documentation available or the documentation is poor and perhaps limited to just a couple of examples. As always, development is dependent on the developer's skills, understanding and the ability to learn new. If there are no other options available, results can be achieved via trial and error –based methods. Such methods are time consuming and prone to failure in the long run. If a framework lacks basic functionality, it forces developers to create their own custom implementations.

Framework-based development has several areas where the work could be more efficient. According to the experiences from developing the example application, the areas in need of improvement are *development practices* and *cloud environment support in frameworks.* In the following sections we suggest improvements to the development.

5.1.1 Environment detection in web frameworks

The example application has two environments: *a local development environment* and *the cloud platform environment*. Holovaty & Kaplan-Moss (2009) provide an example for environment detection but the method used cannot be considered robust. The example uses computer's hostname to detect in which environment the application is running. In a dynamic environment the hostname may change, which would break the implementation.

Django does not offer any methods for detecting the current environment. Because of that, in the example application the environment is detected by using a custom script shown in **Program 5.1**. The simple script looks for the environment variable called *CUSTOM_DJANGO_ENV* that is set explicitly in both of the environments. At this point, the variable can be set to "production" or "development". If the variable is undefined, the application is not allowed to start.

The problem with the method in **Program 5.1** is that it is not built-in to the Django framework. The usecase is very common and exists in every web application project.

Even though the situation with Django is as described, it does not mean that all current frameworks ignore environments. For example, Grails-based web applications can be launched to different environments using the command line utility but in some cases it requires additional compiling and packaging. Grails is a Java EE –based framework that is written in the Groovy scripting language (see http://www.grails.org/).

Program 5.1. *A custom environment detection script in the example application.*

```
# Detect the current environment
try:
    CUSTOM_DJANGO_ENV = os.environ["CUSTOM_DJANGO_ENV"]
    AWS_ACCESS_KEY_ID = os.environ['AWS_ACCESS_KEY_ID']
    AWS_SECRET_ACCESS_KEY = os.environ['AWS_SECRET_ACCESS_KEY']
    AWS_STORAGE_BUCKET_NAME = os.environ['AWS_STORAGE_BUCKET_NAME']
except Exception, e:
    raise Exception("Validate your environment variables.")
```

Environment variables are a common method for configuring environments and especially passing parameters to applications. For example, applications launched from the command line often take parameters in for modifying their runtime behaviour. Parameters can be used to provide configurability. Passing parameters via environment variables enables the developer to change the runtime settings without touching any source code.

To avoid custom implementations we suggest that environment variable –based environment detection should be included as a built-in feature in web frameworks. If a framework provides a basic implementation, the most common usecases are supported automatically. By using the same framework and same practices across projects makes it easier for developers to move from a project to another, as they do not have a need to learn new ways to do the same thing.

5.1.2 Settings management in web frameworks

Database connection details such as credentials, server address, and database name are usually hard coded to a configuration file. The configuration file is then included in the code repository just like any other file. Managing configurations that way becomes difficult in a project with more than one developer. In addition to being difficult to manage, environment-specific configurations can be considered dynamic. For example, developers have their own development environments and the environments can use different configurations. Modifying files to suit the local environment is error-prone, as the environment-specific configuration files cannot be pushed to the version control system without breaking other developers' development environments. Reconfiguration is also needed when there are changes in e.g. service endpoint URLs, which is usual in a cloud environment such as Heroku.

Heroku stores application-specific custom settings and credentials in environment variables. The practice is easy to adapt and in addition it forces to keep the sensitive

information in a safe place. This is relevant in developing open-source projects, which tend to have public code repositories. Leaking database connection details could cause serious security issues. By using environment variables, the code that uses the settings and configurations can be kept generic. An obvious downside with environment variables is that variable names have to be unique to the web application and they need to be used in a way that works on all operating systems that are used for developing. For example, Windows machines do not have case-sensitive variable names. To overcome the problems with unique variable names, a virtual environment can be created in the local development environment. Virtualenv was used in developing the example application to create a sandboxed environment. There are several others available, including the Groovy enVironment Manager (GVM) for Grails application development (see http://gvmtool.net/).

Above we discussed environment detection. In the example application, the detected environment is used for defining which settings to use. As shown in **Program 5.2**, the correct settings module gets loaded based on the detected environment. The settings module can override any settings of the Django application by just redefining them. Importing custom settings is described in the Django's documentation (see Django 2013c), but it is up to the developer how to load settings based on the environment.

Program 5.2. *The settings loading script in the example application. A settings module is imported based on the detected environment. The module can override the default settings by redefining them.*

```
if CUSTOM_DJANGO_ENV == "development":
    print "*** Starting in development environment."
    from settings_development import *

elif CUSTOM_DJANGO_ENV == "production":
    print "*** Starting in production environment."
    from settings_production import *
```

The environment-based settings loading is implemented in the example application in a lightweight fashion. Django provides some advice how to do it, but it is still lacking proper practices. Therefore we suggest that a practice to load environment-based settings should be implemented or at least promoted in web frameworks.

5.1.3 Cloud support in web frameworks

Modern web development involves tools that help developers increase productivity when dealing with code. For example, a common feature is the possibility to automate repetitive tasks such as database schema syncing. Automating tasks ensures that tasks are always done identically. Automation also minimizes human error. In web frameworks, automated tasks are usually included in command-line utilities.

In Django, the command-line utility can start a local development web server for the current project. While running a development web server, changes to the source code

are reflected in real-time to the running version. Whenever a change is made, the source code is checked for syntax errors. If an error is encountered, the command-line utility displays the error message and a stack trace to show the developer where the error happened.

It is a common practice to use command-line utilities within an integrated development environment (IDE). By providing proper IDE support, the development could be made more efficient. Despite the fact that a web framework should be a toolkit and not a design or an implementation guide, it would be useful if the framework could assist the developer. By giving environment-specific hints whether to use certain features or not, the most obvious pitfalls could be avoided. This applies well to first-time use-cases. Such warnings and hints can be achieved by supporting multiple environments in the web framework. A feature that knows in which environments it can safely be used, could include information whether the feature is supported in the selected environment. By giving warnings just like those in the case of syntax errors, unsupported features could be spotted and avoided early.

5.1.4 Agile code reviews and architecture analysis

Quality assurance is an important part of software development. Haikala & Märijärvi (1999, p. 237) define that quality assurance means such actions that aim at assuring that the process and the final product fulfils the set requirements. In agile software development, those actions have to be kept lightweight. As stated earlier, pitfalls in cloud application development are mostly caused by human error. To prevent the consequences, noticing the errors early prevents from doing extra work later.

Because careful design and architecture are vital to cloud applications, code reviews and architecture analysis should be encouraged and committed regularly starting from the first days of development. To keep the review and analysis sessions agile, unnecessary bureaucracy and formalities should be avoided. Such sessions are also a good tool for sharing knowledge between developers. Usually developers work on different features and knowing about what others have done can be considered as a plus from risk management's point of view.

5.2 Portability and exit costs

Exit costs should always be examined whether choosing a platform for a web application or a cloud service from various alternatives. The exit costs increase when the application gets bind to the platform because of specific configurations, use of platform add-ons or platform-specific techniques. Minimizing exit costs increases the amount of custom configurations and maintenance. Exit costs get higher when moving up in abstraction: when an attribute is easily reached, reaching to other requirements is in exchange made very likely more difficult.

For example, the Heroku cloud platform offers many add-ons for doing varying on-demand tasks like video encoding or batch e-mail sending. The add-ons are always con-

nected to a Heroku user account, which is also used for billing the add-on's consumption. If a Heroku application utilizes add-ons, it is in a relationship with the platform provider and the add-on providers and it becomes dependent on those. This kind of relationship weakens portability. By analysing the example implementation's architecture, it can be clearly seen that the implementation is heavily tied to Heroku: The implementation uses Heroku's practices for scaling up and down, background workers, and message queuing. This tells that even a simple application can get tied to its surroundings, which makes it hard to move it to another platform. Moving to another platform has several reasons, like growing out of the platform, changed requirements, and cloud provider discontinuation of service.

Making portable web applications usually complicates the development process. Tanenbaum & Van Steen (2007, p. 8) define *portability* as follows: "Portability characterizes to what extent an application developed for a distributed system A can be executed, without modification, on a different distributed system B that implements the same interfaces as A." In the context of cloud platforms and web applications, the distributed system in the Tanenbaum & Van Steen's (2007) definition can be understood as a cloud platform: From this viewpoint, the challenge of portability lies in the search for cloud platforms with similar interfaces. As cloud platforms vary a lot and many of them have slightly different practices, the only way to make a cloud application as portable as possible might be to not use any platform-specific features. This is always a trade-off between custom implementations and production-ready platform features: It is up to the developer to decide which features to use and which not. To conclude, reaching for high portability means more work in development phase, but less work when porting needs to be done. Designing a web application for portability is usually not an action that would create instant value to the stakeholders, especially in smaller projects with a limited budget.

5.3 Essential cloud platform features

This section concludes what kind of features an elastic cloud application requires from the underlying cloud platform. The conclusions are made based on those Heroku's features that were found useful, on features that were missing, and on those pitfalls and risks that were discovered and analysed.

Easy deployments

Writing deployment scripts manually is not unusual, but considering that web application deployment is a process that is always done similarly, it can be automated. Heroku uses Git for deployments to the platform. The method was found very useful and straightforward. In addition to Git, Heroku uses buildpacks that are basically instruction sets tailored for deploying framework-based web applications. By customizing the buildpack process, a developer can install custom software to the virtual machine.

Secure credentials management

Most cloud applications depend on other services. These services may have restricted access that requires authentication. The credentials used for authentication need to be stored in a safe place. Heroku promotes the practice of storing credentials in environment variables. The practice was found secure, as it prevents the sensitive credentials from being hard-coded to configuration files. The separation of code and configuration helps keep such sensitive information away from code repositories.

Scalability and automatic scaling

One of the key benefits of cloud platforms is that they can be scaled easily without complex configurations: The low-level tasks related to the scaling process are externalized to cloud platform providers. Two types of scaling can be done: *horizontal* and *vertical*. In a nutshell, horizontal scaling is modifying the amount of servers whereas vertical scaling is modifying the amount of computing resources of a server (or servers). As the need to scale horizontally is more than obvious, vertical scalability allows executing e.g. memory-intensive processes that would otherwise run out of memory. In addition to the ability to scale both ways, automatic scaling is also required that allows developers specify rules how and when the web application is scaled. Providing configurable automatic scaling allows leveraging the Reese's (2009) scaling strategies. Heroku is currently limited to horizontal scaling and does not provide automatic scaling as a platform feature. Such applications that require more than 512 megabytes of memory cannot be hosted in Heroku at the moment. This need has been recognized as they are currently offering vertical scaling as a beta (Heroku 2013g).

Background processing

Efficient scaling requires, that it must be possible to distribute work to the background. In the example application, the image manipulation was pushed to the background by implementing a message queue and a background worker script. The message queue based implementation was found clear and expandable. To be able to implement a message queue, a queue service should be offered by the platform. The worker implementation in Heroku was found too simple. Having three background workers running would mean that they are running all the time. To prevent such behaviour and waste of resources, it should be possible to implement a master worker that can spawn child workers depending on the amount of queued jobs.

Platform API and command-line utilities

Heroku has shown excellent example in developing the command-line interface (CLI) for the platform. The Heroku CLI can be used for managing a Heroku user account in all possible ways. In addition to the CLI, Heroku provides an HTTP API for custom implementations. By providing an interface to the platform's features, the platform service can be extended to fulfil various needs.

Monitoring

Low-level monitoring is a key to tracking application performance issues. Heroku application's can be integrated with the New Relic monitoring service (Heroku 2013j), which provides very detailed information about everything related to the application e.g. memory consumption and database queries. The problem with New Relic is that it is a third party service and available as an add-on. As a platform feature, Heroku provides streamed logs via the Heroku CLI. The logs of a Heroku application can be viewed in real-time. The log feature was found useful while monitoring that the example application deployments were successful and that the correct amount of web servers and background workers were started.

Equal support for programming languages

If a platform is about to provide support for several programming languages, the support must be equal to all of the languages. The quality of documentation available for supported programming languages varies in Heroku. For example, Heroku provides many tutorials for Ruby but in some cases other programming languages are left out or the examples are not as complete as the Ruby version.

Support for multiple environments

The need to have separate environments for staging and production use is obvious. Platform support for creating and managing different environments helps the development process. Heroku creates only one environment for each Heroku application, which automatically represents the production environment. By custom practices one can create a staging environment in Heroku, but it can easily get confusing because of manual duplication of settings and configurations.

Connectivity

For example, Heroku allows only HTTP connections. The lack of VPN or any other advanced connection type limits the possible integrations that a web application can have. VPN connections are vital in connections to customer's internal systems. Without such support, the cloud platform does not conform to the requirements and thus cannot be used.

Geographical distribution

Cloud application downtime is usually caused by outages in the infrastructure of the cloud platform provider. In these cases, fault tolerance can be increased by geographical distribution. For example, Heroku hosts everything inside the same data center and in the case of outage there is no way to avoid downtime as everything is down (see GigaOm 2012). Therefore, a feature must exist that allows distributing the web application to other geographical locations.

6 CONCLUSIONS

Cloud platforms may feel inexpensive but cheap cloud computing does not simply exist. Somebody always has to pay the bill, and in a web application project it is usually the customer. Regarding the limitations of Heroku and other similar cloud platform providers, it is likely to encounter problems related to the platform's service offerings. For example, Heroku has already changed their business direction several times and each change has affected their offerings. These changes always affect certain customers (Irwing 2012; Somers 2013). According to Somers (2013), a recent modification has changed the behaviour of Heroku's load-balancing tier and has caused severe degradation of performance in large web applications. The load-balancing tier now forwards requests randomly as it had been previously based on a more intelligent solution (Somers 2013). Despite that the services have matured that were used in the example application, these risks still exist and especially within service providers that are start-up companies.

As long as the current frameworks stay the same, developers have to use workarounds to overcome pitfalls and missing practices. Building a Django application on top of the Heroku platform is well documented in Heroku's documentation, which makes first steps easy. The documentation provides an in-depth tutorial for running Django both with the development server and with the production-ready Gunicorn server (Heroku 2013b). In web development, problems start to arise when the implementation needs to step outside of examples and tutorials. This is typical in plug-in based implementations: The plug-in developers do not have all use cases covered in documentation nor do they have interest towards such. In the end, it is the developer's responsibility to use the plug-ins for only those purposes what they are meant for. A general misconception is that the web frameworks work well in the cloud environment just like they have done before in the single-server environment. As far as one understands what the frameworks and plug-ins are meant for, they can be safely used. Despite that not all of the solutions described in this thesis are directly applicable to other web frameworks, the problems behind them do exist when developing any web application for the cloud environment.

Due to the scope of the thesis, such as performance issues and technical details related to but not limited to caching, database distribution and multimedia content delivery were not discussed. These are obvious research directions for future work. The findings and the work still act as a good starting point for researching the subject further just by lowering the abstraction level.

BIBLIOGRAPHY

Amazon Web Services. 2013. Amazon Simple Storage Services. Accessed 13.4.2013. URL: http://aws.amazon.com/s3/

Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M. 2010. A view of cloud computing. Communications of the ACM. Volume 53, Issue 4, pp. 50–58.

Boto. 2013. Python interface to Amazon services. Accessed 13.4.2013. URL: https://github.com/boto/boto

Cardellini, V., Colajanni, M., Yu, P. 1999. Dynamic load balancing on web-server systems. Internet Computing, IEEE, Volume 3, Issue 3, pp. 28–39.

Coulouris, G., Dollimore, J., Kindberg, T. 1994. Distributed Systems: Concepts and Design. Second edition. USA. Addison-Wesley Publishing Company. 644 p.

Django. 2013a. Django homepage. Accessed 29.1.2013. URL: https://www.djangoproject.com/

Django. 2013b. Design philosophies. Accessed 25.2.2013. URL: https://docs.djangoproject.com/en/dev/misc/design-philosophies/

Django. 2013c. Custom default settings. Accessed 4.4.2013. URL: https://docs.djangoproject.com/en/1.4/topics/settings/#custom-default-settings

Django. 2013d. URL dispatcher. Accessed 4.4.2013. URL: https://docs.djangoproject.com/en/1.4/topics/http/urls/

Django. 2013e. Including other URLconfs. Accessed 4.4.2013. URL: https://docs.djangoproject.com/en/1.4/topics/http/urls/#including-other-urlconfs

Django. 2013f. Django-admin.py and manage.py. Accessed 6.4.2013. URL: https://docs.djangoproject.com/en/1.4/ref/django-admin/

Django. 2013g. Cross Site Request Forgery protection. Accessed 9.4.2013. URL: https://docs.djangoproject.com/en/1.4/ref/contrib/csrf/

Fardone, G. 2012. Cloud elasticity and cloud scalability are not the same thing. Accessed 17.3.2013. URL: http://blog.evolveip.net/index.php/2012/05/24/cloud-elasticity-and-cloud-scalability-are-not-the-same-thing-2/

Gartner. 2008. Gartner Says Contrasting Views on Cloud Computing Are Creating Confusion. Accessed 18.1.2013. URL: http://www.gartner.com/newsroom/id/766215

Gartner. 2009. Gartner Highlights Five Attributes of Cloud Computing. Accessed 18.3.2013. URL: http://www.gartner.com/newsroom/id/1035013

GigaOm. 22.10.2012. Amazon problems take down Reddit, other sites. Accessed 10.4.2013. URL: http://gigaom.com/2012/10/22/amazon-problems-take-down-reddit-other-sites/

Grails. 2013a. Grails homepage. Accessed 29.1.2013. URL: http://grails.org/

Grails. 2013b. Grails and Hibernate - Reference documentation. Accessed 19.4.2013. URL: http://grails.org/doc/latest/guide/hibernate.html

Grails. 2013c. Configuration - Reference documentation. Accessed 19.4.2013. URL: http://grails.org/doc/latest/guide/conf.html

Haikala, I. & Märijärvi, J. 1998. Ohjelmistotuotanto. 5th edition. Jyväskylä, Finland. Gummerus Kirjapaino Oy. 385 p.

Heroku. 2012a. Heroku.py API wrapper for Python. Accessed 2.3.2013. URL: https://github.com/heroku/heroku.py

Heroku. 2012b. Background Tasks in Python with RQ. Accessed 6.3.2013. URL: https://devcenter.heroku.com/articles/python-rq

Heroku. 2013a. Workers and Queuing. Accessed 5.2.2013. URL: https://addons.heroku.com/#queues

Heroku. 2013b. Using a different WSGI server. Accessed 24.2.2013. URL: https://devcenter.heroku.com/articles/django#using-a-different-wsgi-server

Heroku. 2013c. Specifying a Python runtime. Accessed 24.2.2013. URL: https://devcenter.heroku.com/articles/python-runtimes

Heroku. 2013d. Django settings. Accessed 25.2.2013. URL: https://devcenter.heroku.com/articles/django#django-settings

Heroku. 2013e. Stacks. Accessed 6.4.2013. URL: https://devcenter.heroku.com/articles/stack

Heroku. 2013f. Usage & Billing. Accessed 6.4.2013. URL: https://devcenter.heroku.com/articles/usage-and-billing

Heroku. 2013g. 2X Dynos in Public Beta. Accessed 10.4.2013. URL: https://blog.heroku.com/archives/2013/4/5/2x-dynos-beta

Heroku. 2013h. Managing Multiple Environments for an App. Accessed 10.4.2013. URL: https://devcenter.heroku.com/articles/multiple-environments

Heroku. 2013i. Heroku Scheduler. Accessed 12.4.2013. URL: https://devcenter.heroku.com/articles/scheduler

Heroku. 2013j. New Relic. Accessed 13.4.2013. URL: https://devcenter.heroku.com/articles/newrelic

Heroku. 2013k. How it works. Accessed 13.4.2013. URL: https://www.heroku.com/how

Holovaty, A. & Kaplan-Moss, J. 2009. The Django Book. Accessed 25.2.2013. URL: http://www.djangobook.com/

Irving, F. 2012. Heroku's early history: 4 home pages that made $212 million. Accessed 6.4.2013. URL: http://www.flourish.org/blog/?p=687

Java.com. 2013. Helpful Concepts and Definitions Glossary. Accessed 6.4.2013. URL: http://java.com/en/download/faq/helpful_concepts.xml

Koskimies, K. & Mikkonen, T. 2005. Ohjelmistoarkkitehtuurit. Accessed 28.1.2013. URL: http://www.cs.tut.fi/~kk/Ohjelmistoarkkitehtuuri.pdf

Lamport, L. 1987. A message regarding distributed system definitions. Accessed: 10.1.2013. URL: http://research.microsoft.com/en-us/um/people/lamport/pubs/distributed-system.txt

Mell, P. & Grance, T. 2011. The NIST Definition of Cloud Computing. USA. National Institute of Standards and Technology. Accessed 9.1.2013. URL: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

Microsoft. 2013. ASP.NET MVC homepage. Accessed 16.3.2013. URL: http://www.asp.net/mvc

Nebula. 2013. Service listing and pricing. Accessed 1.4.2013. URL: http://nebula.fi/fi/palvelut/pilvipalvelut/webhotellit

Redis Queue. 2013. Redis Queue Python library homepage. Accessed 8.4.2013. URL: http://python-rq.org/

Reese, G. 2009. Cloud application architectures. USA. O'Reilly Media. 189 p.

Rellermeyer, J., Duller, M., Alonso, G. 2009. Engineering the Cloud from Software Modules. CLOUD '09 Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing. Washington DC, USA. IEEE Computer Society. 32–37 pp.

Rimal, B., Jukan, A., Katsaros, D., Goeleven, Y. 2010. Architectural Requirements for Cloud Computing Systems: An Enterprise Cloud Approach. Journal of Grid Computing. March 2011, Volume 9, Issue 1, pp. 3–26.

Ruby on Rails. 2013. Ruby on Rails homepage. Accessed 29.1.2013. URL: http://rubyonrails.org/

Schlossnagle, T. 2007. Scalable Internet Architectures. USA. Sams Publishing. 262 p.

Sitaram, D. & Manjunath, G. 2012. Moving to the Cloud: Developing Apps in the New World of Cloud Computing. USA. Elsevier Inc. 448 p.

Sodhi, B. & Prabhakar, T.V. 2011. Application Architecture Considerations for Cloud Platforms. 2011 Third International Conference on Communication Systems and Networks (COMSNETS), 4–8 January 2011, Kanpur, India, pp. 1–4.

Somers, J. 2013. Heroku's Ugly Secret. Accessed 10.4.2013. URL: http://rapgenius.com/James-somers-herokus-ugly-secret-lyrics

Stackoverflow.com. 2012. Python, PIL and JPEG on Heroku. Accessed 10.4.2013. URL: http://stackoverflow.com/questions/10213509/python-pil-and-jpeg-on-heroku

Stoelinga, S. 2012. Only use a worker when required on Heroku with Django/Python. Accessed 10.4.2013. URL: http://samos-it.com/only-use-worker-when-required-on-heroku-with-djangopython/

Tanenbaum, A. & Van Steen, M. 2007. Distributed Systems: Principles and Paradigms. Second edition. USA. Pearson Education Inc. 686 p.

Toffetti, G. 2012. Web engineering for Cloud computing. Accessed 8.1.2013. URL: http://mdwe2012.pst.ifi.lmu.de/wp-content/uploads/2012/07/mdwe2012_submission_3.pdf

VMWare. 2006. Virtualization Overview. Accessed 6.4.2013. URL: http://www.vmware.com/pdf/virtualization.pdf

Zhang, Q., Cheng, L., Boutaba, R. 2010. Cloud computing: state-of-the-art and research challenges. Springer-Verlag. Journal of Internet Services and Applications. Volume 1, Issue 1, pp. 7–18.