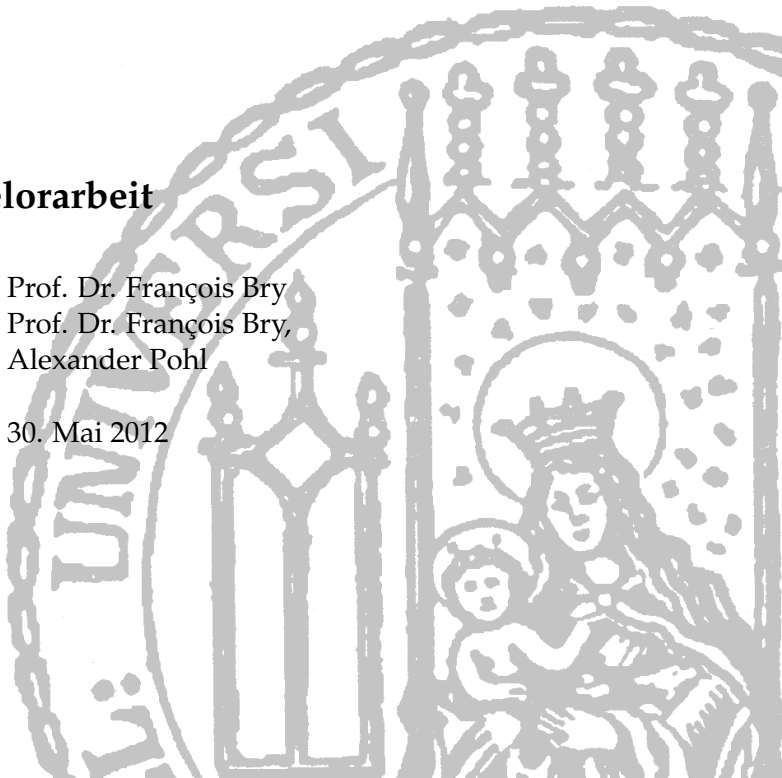
INSTITUT FÜR INFORMATIK
der Ludwig-Maximilians-Universität München

Making the Backchannel Backstage Failure-Resistant

Towards Better Scalability

Max Kleucker



Bachelorarbeit

Aufgabensteller	Prof. Dr. François Bry
Betreuer	Prof. Dr. François Bry, Alexander Pohl
Abgabe am	30. Mai 2012

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Hilfsmittel verwendet habe.

München, den 30. Mai 2012 Max Kleucker

Abstract

Modern web applications are expected to provide a good or even outstanding user experience. Besides an usable interface this also involves a quickly responding and seamless working web service. So the distance between user and server as well as any potential delays should remain unascertainable. A highly-available and scalable service is necessary in order to provide such a user experience regardless of the number of users.
This work is based on the existing backchannel software Backstage. This web application is designed to be used in large lecturers where it enables digital communication in the manner of microblogging while indirectly providing a backchannel for the lecturer and the students. On the one hand the communication has the goal of letting students aid one another and on the other hand the lecturer is able to gain instant feedback from analyzing the students information exchange.
Upon ensuring the functionality this thesis analyzes the application and its implementation to develop strategies to increase the failure resistance and to gain scalability. Those strategies are directly applied to the prototypical implementation of Backstage and evaluated.

Zusammenfassung

Von moderne Webanwendungen wird verlangt, dass sie ein vorzügliches Nutzungserlebnis bereitstellen. Neben einer durchdachten Benutzeroberfläche umfasst das auch eine schnell reagierenden und makellos funktionierenden Webdienst. So sollen die Entfernung zwischen Nutzer und Server unbemerkbar bleiben, ebenso wie mögliche Verzögerungen. Ein hochverfügbarer und skalierbarer Dienst ist nötig, um solche ein Nutzungserlebnis unabhängig von der Anzahl der Nutzer zu ermöglichen.
Diese Arbeit basiert auf der existierenden Rückkanal-Software Backstage. Diese Webanwendung ist daraus ausgelegt in großen Vorlesungen genutzt zu werden. Dort ermöglicht sie digitale Kommunikation in Form von Microblogging und dient dem Vortragenden indirekt als Rückkanal. Auf der einen Seite hat diese Kommunikation das Ziel, dass sich die Studenten gegenseitig unterstützen. Auf der anderen Seite kann aus dieser Kommunikation direktes Feedback für den Vortragenden gewonnen werden, indem der Informationsaustausch der Studenten analysiert wird.
Ausgehend von der Sicherstellung dieser Funktionalität analysiert diese Arbeit die Anwendung und deren Implementierung, um davon Strategien für eine Verstärkung der Fehlerresistenz abzuleiten und daraus die Skalierkeit zu verbessern. Diese Strategien werden direkt auf den Prototypen der Anwendung Backstage angewandt und evaluiert.

Acknowledgements

First and foremost I want to thank Prof. Dr. François Bry who made this thesis possible and supervised it. His open-minded and valuable thoughts regarding technical implementation and integration patterns broadened my general perspective on those. Furthermore I want to thank Alexander Pohl for advising this work and accompanying the creation process. His highly competent remarks together with his creative ideas had a sustainable influence on this work. Overall the pleasant atmosphere in multiple meetings and conversations created an enjoyable environment for working on the project and finally deliverin this thesis.
Finally I would like to thank my family and friends for their support along the way.

Contents

1 Introduction . . . 1
1.1 Motivation . . . 1
1.2 Goals of this Thesis . . . 2

2 Failure Resistance and Scalability . . . 3
2.1 Failure Resistance . . . 3
2.1.1 Software Failure . . . 4
2.1.2 Failures in Web Applications . . . 4
2.1.3 Network Errors . . . 4
2.2 Scalability . . . 5
2.2.1 Vertical Scalability . . . 5
2.2.2 Horizontal Scalability . . . 6
2.2.3 Scalability in Web Applications . . . 6
2.3 Failure Resistance and Scalability for Backstage . . . 6
2.3.1 Definitions and Assumptions for Backstage . . . 7
2.3.2 Use Cases . . . 7
2.4 Qualities of Service . . . 8
2.4.1 QoS for Backstage . . . 8

3 Current State of Backstage . . . 9
3.1 Architecture . . . 9
3.1.1 Stateful Application versus Scalability . . . 9
3.1.1.1 Sticky Sessions . . . 10
3.1.2 Backstage and External Components . . . 10
3.1.3 Looking inside Backstage: The Architecture of Backstage . . . 10
3.2 The Technical Specifications . . . 11
3.2.1 Tearing apart Backstage . . . 11
3.2.2 Application Framework: Grails and Groovy . . . 12
3.2.2.1 Views and Controllers . . . 13
3.2.3 WebServer: Using Jetty as the Application Server . . . 13
3.2.4 Database: MongoDB . . . 13
3.2.4.1 Providing the data from MongoDB to Backstage: Morphia 14

3.2.5 Communicating with the Clients: AJAX and Views 14
3.2.5.1 AJAX and Reverse AJAX 14
3.2.5.2 DWR 15
3.2.6 Lecture Context 15
3.3 Tasks and Information Flow 16
3.3.1 Processing an incoming Post 16
3.3.2 Creating Slides 17
3.3.3 A Quiz-Round 17

4 Taking Backstage to the Cloud 19
4.0.3.1 Additional Requirement: Local Data Storage 20
4.1 What is Cloud Computing 20
4.1.1 Cloud Computing as an approach of horizontal scalability 21
4.1.2 Cloud and Traditional Hosting 22
4.1.3 Cloud and Grid Computing 22
4.1.4 Types of Cloud Computing 22
4.1.4.1 Platform as a Service – Google AppEngine 22
4.1.4.2 Infrastructure as a Service – Amazon Web Services 23
4.2 Backstage in the Cloud 23
4.2.1 Providing Backstage 23
4.2.2 Requirements 24
4.3 The process of taking Backstage to the Cloud 24
4.3.1 Decision for Amazon Web Services 24
4.3.1.1 Quick introduction to EC2 25
4.3.2 Preparing Backstage 25
4.3.3 Deploying Backstage into the cloud 25
4.3.3.1 Configuring an EC2 instance 25
4.3.4 Running Backstage on AWS 27
4.3.4.1 Enabling external access 27
4.4 Further thoughts on a cloud based Backstage 27
4.4.1 Automated Scaling 27
4.4.2 Quick Recovery 28

5 Improving Backstage 29
5.1 Problematic areas of Backstage 29
5.1.1 Internal Processes 29
5.1.1.1 Processing Posts 29
5.1.1.2 Rendering Slides 30
5.1.1.3 Synchronous processes 30
5.1.1.4 Aggregating Data 30
5.1.2 Extensibility 31
5.1.2.1 Further Plans for Backstage 31
5.1.3 Challenges for Backstage 31
5.2 Rethinking Backstage 32
5.2.1 A Different Integration Pattern: Messaging 32
5.2.1.1 Messaging in a Restaurant 32

5.2.1.2 Introducing JMS and ActiveMQ 33
5.2.1.3 Messaging Objects 33
5.2.1.4 Benefits of using JMS 34
5.2.1.5 Message Driven Components 35
5.2.2 Integrating JMS in Backstage 35
5.2.2.1 Preparation: Extracting the Entry Points 36
5.2.2.2 Preparation: Uncoupling the MicroblogBean 36
5.2.2.3 Preparation: Defining a Message Format and a Queue . . 36
5.2.2.4 MicroblogDWRBean: Forwarding Messages to the Queue 38
5.2.2.5 MicroblogBean: Reacting on Messages from the Queue and Pushing Them to the Clients 38
5.2.2.6 Dispatcher: Sending the Post to the Clients 38
5.2.2.7 Configuration: Parallel Beans through Messaging 38
5.3 Analyzing the modified Backstage 39
5.3.1 A More Robust Backstage 39
5.3.2 Better Extensibility 40
5.3.3 Potential Next Steps Regarding Messaging 40
5.3.3.1 Status Reports on Tasks 40

6 Conclusion 41

Bibliography 43

CHAPTER 1

Introduction

This thesis is located in the context of the larger project Backstage. Backstage is a digital backchannel for large lecture classes developed at the Ludwig-Maximilians-University of Munich. The purpose of this application is to serve as way to improve communication between students and the lecturer during lecture sessions. It offers students and the presenter non-disruptive real-time communication where students exchange information in order to gain a better understanding of the topic. Teachers gain a valuable backchannel to a classic frontchannel form of teaching with a low level of active participation and feedback.
All communication is centred around the topic of a lecture providing the students an annotatable set of the current slides where they can exchange information and questions in a style comparable to micro-blogging services. The lecturer can gain feedback but also use the system actively to provoke participation of the students e.g. by starting a quiz using Backstage. This way students shall both be guided and triggered help one another by exchanging information pieces on the slides.
Technically Backstage is a web application and follows the basic client – server architecture. This implies that Backstage keeps all data centrally on this server and processes incoming data from the clients as well as serving the clients new data.

1.1 Motivation

Already at an early stage the prototype of Backstage should be used in studies to evaluate its functionality and the acceptance among the users. In order to conduct meaningful and convincing experiments it is necessary that the software Backstage can handle a large number of users.
Failure resistance as the basis of a highly available web-application can be illustrated from two different angles: On the one hand the user expects a flawless behavior from the application. Part of which is to be able to interact with it without delays and to rely on the data the application shows. Speaking of a nearly real-time

interaction model like Backstage supposes it, the user even demands to be always presented with the most recent data while being able to carry out his message to everyone at the very same time.
On the other hand there is the technical staff whose task it is to run and maintain Backstage. To them it is important to rely on a well-running instance of Backstage as well. Being able to calculate the server requirements based on the expected number of simultaneous users gives them a tool to ensure that the server environment is always well-managed for a seamless running of Backstage.
Both points of view are strongly connected. Having administrators providing an appropriate server-environment reduce the risk of both themselves and users having to deal with unexpected of the behavior of the application.
This expectations require Backstage to ensure a high availability by being a failure resistant and therefore robust application.

1.2 Goals of this Thesis

This work develops practical approaches to increase the failure resistance of Backstage. Those are filtered from general concepts regarding the architecture of a web-application and further developed and adapted to the specific circumstances of the project Backstage. The final goal is to create a more robust application and to lay the foundations for a Backstage used by multiple hundreds of users.
Firstly the basic principles of failure resistance are presented with the goals of ensuring a high availability and increasing the scalability. Different concepts will be examined and introduced taking a look at the used technical components and the architecture of Backstage as an application.
The practical aspiration of this work are pursued by implementing such concepts into Backstage and documenting this work to gain more general findings by implication which on their behalf can be relevant for other web applications as well.

CHAPTER 2

Failure Resistance and Scalability

Backstage is designed to be available a large number of users. Up to two hundred users are expected to take part in the first studies. In those experiments shall the acceptance by and the usage of Backstage be tested. A flawless functioning prototype is therefore essential to ensure convincing and reliable results. This demands a high availability from the application Backstage. The availability concerns the general system at any point and the specific functionalities during the time of use. This requested availability is directly connected to failure resistance as it is a positive effect thereof. Availability as such is the targeted state of a failure resistant Backstage and used in this reference throughout this thesis.

2.1 Failure Resistance

Failure Resistance describes the ability of any kind of system, material or structure, to withstand any influence while functioning as expected. The positive and practical effect of a high level of Failure Resistance is stability[4]. Speaking of a software system Failure Resistance summarizes its characteristics and abilities of withstanding unwanted effects.
Analyzing the occurrences and types of failures helps to gain a better understanding of those. This is the foundation to create strategies for increasing failure resistance.
Dealing with the web it is likely that most Internet users have encountered failures using the web. A common categorization for failures in web applications is [17]

- Software Failure
- Human Operator Errors
- Hardware and Environmental Failures
- Security Violations

In order to keep this thesis focused I narrow the field to the subject of *Software Failure*.

2.1.1 Software Failure

In order to make assumptions on the failure resistance of a system it is vital to find the situations in which the system fails and the reasons for particular failures. Failures in software system might either come abruptly caused by a single action or it might announce itself by a precedent slowing down.
Most computers are likely to have encountered these types of failures of a software system: Applications which crash all of a sudden or applications responding slower and slower until they freeze or crash. Assuming that this behavior is either not replicable and the application was used as intended, this falls into our category of *Software Failure*.
For web applications it is possible to recognize different domains of Failure Resistance [17]:

Resource Exhaustion The resources of the machine running the application don't cover the application's need. This can be caused by errors within the application.

Computational / Logic Errors Errors within the application result in unwanted behavior.

System Overload In difference to *Resource Exhaustion* which is caused by external limitations, *System Overload* is the result of too many requests the system logic cannot handle.

Recovery Code The intended goal of *Recovery Code* is to enable a fast and reliable recovery after a failure. But faulty recovery code can have the opposite effect.

Failed Software Updates Problems and failures during the update process might cause further outages and a failures of the whole system.

2.1.2 Failures in Web Applications

Web applications in general suffer from the impression that a large amount of them is considered poorly developed, to which some refer as "Web Crisis" [8].
In order to improve failure resistance it is important to know potential sources of failures. Speaking of larger applications automated tests are essential to have a continuous overview about the application's stability.
There is a variety of tools which can conduct regular functionality testing of a web application. [18]

2.1.3 Network Errors

Failure resistance in web applications has a special external dependency looking at the components between user and application which we aggregate as network. As

mentioned earlier we will mostly inspect the software although those network are responsible for up to 81 percent of service failures depending from the read-write ratio of a service [15].
Ensuring failure resistance is a distributed task affecting all many components between the user and the application itself. For example in the situation of providing Backstage for a LMU-based lecture this includes the Leibniz-Rechenzentrum (LRZ) for maintaining the basic IT infrastructure, the system administrator of the LFE für Programmier- und Modellierungssprachen (PMS) for maintaining the local server running Backstage, the teams behind the external components Backstage relies upon for providing a functioning software and the developers of Backstage who are responsible for ensuring the correct behavior of the application itself.
Therefore this work focuses on ensuring failure resistance on the application level. Topics emerging from different areas are neglected.

2.2 Scalability

Scalability describes the ability to handle an increase of work or to be accommodated to a larger environment [2]. It is a key characteristic of highly available applications. Looking at the goal of the development of Backstage itself: to be usable for a single lecture as well as for a whole university we can see a clear increase in requirements.
Regarding scalability consider a small restaurant. The service room has 30 spots to admit persons which are served by two waitresses. The kitchen is relatively tight and only able to cook only one serving of each of the meals they offer. Given the situation everything works out just fine.
The process of when a new guest enters the restaurant starts with the waitress taking the order. This order is placed on a tray where all open orders are stacking up. The kitchen takes the orders as soon as there is enough space to cook them. Once a meal is done it is passed to waitresses again which serve it to the guest.
In analogy to this restaurant the application is subject to a similar structure and system. Different components interact with each other and they all have their own limitations and features. The guests are now the users of the application. The waitresses are the communication components between the users and the application. While the kitchen is the application itself with its functionality.
Changing the demand – the number of guests in the restaurant or the number users of the web application – results in changed requirements. Meeting the requirements for this new situation is scaling.

2.2.1 Vertical Scalability

There are two specific subsets of scalability: horizontal and vertical scalability. They present different approaches on how to handle an increase in demand.
The later one describes the extension of the resource for the existing system. For the restaurant this would mean to extend the kitchen so it can serve more meals in the same time. For a software this extension equals in increasing the power of the machine: adding CPUs or memory.

2.2.2 Horizontal Scalability

Horizontal scalability leaves the existing system unchanged. Instead new instances of the same system are added. The increase in demand is then met by multiple systems which essentially all do the same thing.
Again in reference to the example of the restaurant this equals opening a second restaurant right besides the existing one and offer the very same menu. But it is necessary to hire another person, a receptionist, who distributes the guests among the restaurants to ensure a balanced workload for both restaurants.
From the perspective of a developer of web applications this would result in setting up a second server with the web application on it. The role of the receptionist is taken by a load-balancer which distributes the users upon the different servers.

2.2.3 Scalability in Web Applications

The circumstances under which a web-application is used vary from minute to minute as the number of clients is fluid and the actions by the clients are largely unpredictable. Yet Backstage has got a little advance regarding the distribution of simultaneous users: Lectures usually take place on a fixed schedule allowing a more precise prediction of the number of users. Therefore also the activity of the users and the resulting load for Backstage can be certainly assumed to be higher during those lectures.
Beforehand let me give a quick introduction on the different levels and possibilities of scaling a web application.
At any given moment the web-application is scaling on the smallest level: Accommodating itself to single additional users. Every single users increases the need for resources of a web application. While the necessary infrastructure is provided by environment, it is up to the application to ensure a correct behavior.
Going up from single users to dozens or hundreds of users a web-application we can look at scaling single components. Grown applications consist of different components which work together to provide the application's functionality. In most cases some of those parts are more stressed then others. And some parts are more critical to provide the service. Those two selections may overlap and give a hint on which components should be evaluated for improvement.
Once the limits of a single instance of an application are reached one may look at scaling the runtime environment. In analogy to horizontally scaled restaurant this means creating a further instance of Backstage and dividing the users between them.
We will come back to these different levels and approaches during the thesis.

2.3 Failure Resistance and Scalability for Backstage

This understanding of the terms Failure Resistance and Scalability and their importance brings up their role in Backstage. For a better evaluation there are additional definitions and assumptions to be done in the following part.

These points will be the base for later analysis of the challenges for Backstage and are considered with the idea of high availability in mind.

2.3.1 Definitions and Assumptions for Backstage

There are different key terms to describe the usage of Backstage from an organizational perspective:

Courses A single course describes the situation of a teaching person holding a class over a longer period of time with periodically meetings in which Backstage is used.

Session A single session is an organizational for course sessions. It usually starts and ends with the administrative steps the teacher takes using the Backstage interface.

User / Client A user or client is any to Backstage connected entity who can take the role of a lecturer or a student passively following a session.

2.3.2 Use Cases

Backstage is meant to be an application that fits different courses: While on average a course is unlikely to exceed 400 students, there are courses reaching 800 students.
Organizationally Backstage is based on the idea of courses with sets of sessions. As there is no connection between different courses the data of a single course must be only available within this particular course.
Furthermore each session takes place in a virtually encapsulated environment. Users are only interacting within these sessions.
Starting from there we can see the simplest use case as a single course with a sequent series of sessions. One goal of Backstage is to provide its service to whole universities with multiple hundred courses and at peak times almost as many parallel sessions. In between almost any constellation shall be possible.
Deriving from the three key factors we declared in the precedent part there are three more precise numbers which are the foundation to our estimations:

1. *Number of Courses* illustrating the general amount of data Backstage has to cope with.

2. *Number of concurrent Sessions* which has a main impact on the load Backstage deals with during the time of execution.

3. *Number of connected Clients per Session* multiplied with the *Number of concurrent Sessions* it generates the total number of connections Backstage has to maintain.

2.4 Qualities of Service

We summarize the levels to which Backstage satisfies different requirements as Qualities of Service (QoS). For web applications and web services in general there exist multiple sets of QoS in literature [14, 11, 12] mostly differentiating in the tasks of the application they are applied to.

2.4.1 QoS for Backstage

The World Wide Web Consortium suggests twelve very generic characteristics [11] for web services which are listed below with their specific reference to Backstage:

Performance — describes how fast Backstage can process requests.

Reliability — describes the ability of Backstage to provide the expected functionality.

Scalability — described the ability of increasing the computing capacity in order to handle more users.

Capacity — is the maximum number of user requests Backstage can handle simultaneously.

Robustness — summarizes Backstage's capabilities to deal with incorrect data while still functioning correctly.

Exception Handling — describes how Backstage handles errors which occur while processing requests.

Accuracy — aims for keeping the number of errors Backstage creates to bare minimum.

Integrity — stands for avoiding unauthorized access on the one hand and ensuring the correctness of data on the other.

Accessibility — represents whether Backstage is able to serve a client's request.

Availability — can be measured in the time it takes until a client can use Backstage.

Interoperability — means that the functionality Backstage provides can be used by other developers without restrictions regarding programming language or operating system.

Security — is a large topic summarizing all aspects it takes for Backstage to keep the data safe and only accessible for their permitted use.

Network-Related QoS Requirements — describes all factors regarding the actual connection between Backstage and the user which we already discussed in 2.1.3

These QoS give the larger reference for the characteristics of Backstage and are used to highlight the practical purpose of the different steps taken in this thesis.

CHAPTER 3

Current State of Backstage

Backstage has been initially developed with the aspiration to quickly have a functional software. Many design choices were affected by this goal and therefore many of the aspects regarding scalability and failure resistance were left unnoticed in the first place.
The following analysis presents the current state of Backstage and the starting point for the further actions.

3.1 Architecture

Students and lecturers in their role as the end-users of Backstage will only see the website rendered in their browsers. As every user expects an individual and customized view in order to use the full sets of function of Backstage it is necessary that the data sent to the users is constantly generated and always up-to-date.
Speaking only of the clients or users on the one side and Backstage on the other is a strong simplification. The architecture of Backstage consists of many different components and modules on different levels.

3.1.1 Stateful Application versus Scalability

Backstage is a stateful application. It keeps track of different events which happen during it lifecycle. With every new event the state of the application changes. For Backstage this means it gathers and aggregates all events, such as new postings, internally. The users are served with information depending on this state.
In contrast a stateless application would fully rely on the information the clients sends in and compute the response based only on this information.
The ability to scale Backstage horizontally suffers from this statefulness. As the state is not shared among multiple instances only a single instance can ensure the correctness of its data.

3.1.1.1 Sticky Sessions

In view of the restaurant example there are two restaurants beside each other: A and B. A guest walks into A and orders dish number 113. Restaurant A holds the current state regarding this guest now. In the meantime she steps out of the restaurant to have a cigarette but then enters restaurant B. But B can't serve her dish 113 has it doesn't know about the order. To prevent this there is our receptionist in front of the restaurants who ensures that guests are distributed and that each guest will find his table again.
The technical counterpart to this receptionist are `Sticky Sessions`: The routing instance which distributes the requests among the different instances also ensures that clients are always connected to the same instance.

3.1.2 Backstage and External Components

The first level helps to separate the application from external components Backstage needs to be functional: The actual webserver and the database.

Webserver / Application server in this context does not describe the physical machine Backstage might run on but rather the software component to connect the users with the application Backstage. It takes incoming requests from the users and forwards them. Afterwards the generated answers are delivered through it from Backstage to the clients.

The Database at the other end is a storage to persist the data Backstage creates. It is the long-term memory of Backstage so that the data is still available if the system is restarted. It also allows us to use the data outside of Backstage for example in order to analyze the data for further scientific use or evaluation.

Backstage at this level stands for the actual web application as it is built and functioning as a unit. This separation is necessary as both database and webserver are usually not controlled by the web application. They are independent parts that serve Backstage with their functionality over clearly defined interfaces.

3.1.3 Looking inside Backstage: The Architecture of Backstage

While the previous view might be applicable for almost any web application from here on the peculiarities of Backstage increase noticeably (Figure 3.1).
In general Backstage is built following the the basic model-view-controller (MVC) architecture. Accordingly the *Views and Controllers* are dealing with the incoming requests from the clients and the rendering of particular informations which are sent to them in return. The *Application Logic* is decoupled from this direct communication with the clients and contains the actual business logic. Furthermore a *Lecture Context* is present within Backstage which stores and organizes temporary data.

The Views and Controllers have the purpose of taking incoming request and processing them. Furthermore they contain the logic for creating the responses sent to the clients and are therefore responsible for what is finally displayed in the users browser window. This functionality is provided by the used web framework.

The Application Logic is the part which aggregates everything and implements the actual functionality of Backstage. Incoming data is processed and calculations are made and data sets for the clients are built here. It is the core of Backstage and subject for the next levels.

The Lecture Context stores and organizes the information regarding the different lecture sessions. This way it reduces the need for looking into the database and works like an additional cache this way.

3.2 The Technical Specifications

All technical choices are driven by the decision to use Java as the basic programming language. Upon this foundation the following part gives an overview over the specific technologies and components used inside Backstage.

3.2.1 Tearing apart Backstage

The previously established schema is again a simplification when the interest is upon the actual implementation, upon the actual code. Looking at it again there

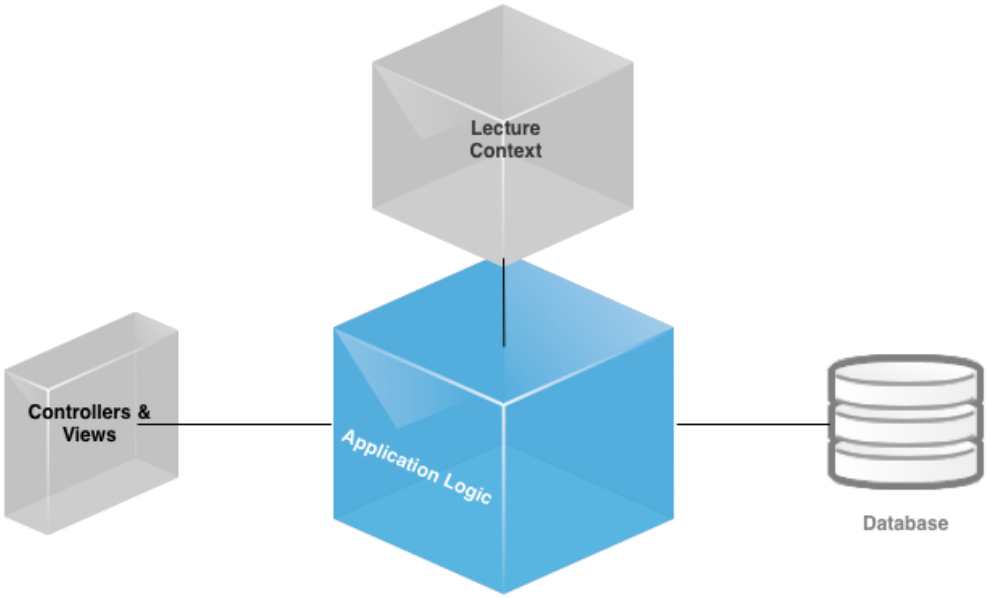


Figure 3.1: Basic Application Layout

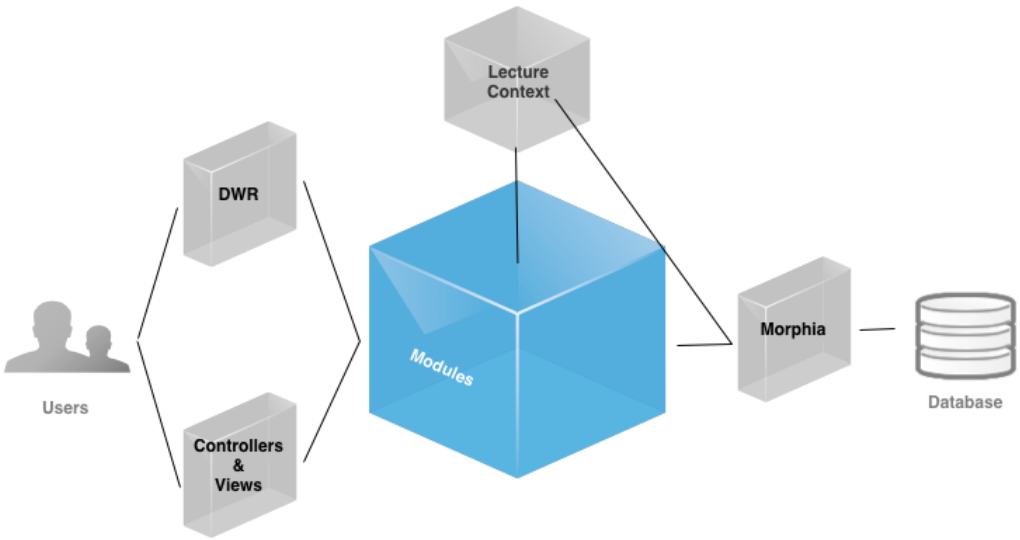


Figure 3.2: Extended Application Architecture

are additional components appearing. They have a distinct function which will be explained hereafter.
To simplify the development process and in order to quickly create a functioning version of Backstage it uses a variety of third-party frameworks and libraries [3].

3.2.2 Application Framework: Grails and Groovy

Grails[1] is an open-source framework built for web application and based on Groovy[2]. Groovy on the other hand is a programming language written for the Java Virtual Machine.
Grails is designed to support developers by offering the basic functionalities of a web-application, such as request handling, and thereby only the concrete tasks need to be implemented which reduces implementation overhead and allowing faster results. In conjunction with Groovy it allows the dispensation of programming in Java to certain level by offering a syntax more similar to Ruby[3] and Python[4].
Given the constellation that all code will be executed in the Java Virtual Machine there is also the option to use components written in Java itself. Thanks to this interoperability Backstage uses all three options: For the components which produce the views a student or teacher will see in his browser we rely on the specific components of Groovy. While the application logic and the processing of data are mostly written in Java.

[1]Grails - the search is over, `http://grails.org/`
[2]Groovy - Home, `http://groovy.codehaus.org/`
[3]The programming language Ruby, `http://www.ruby-lang.org/`
[4]Python Programming Language - Official Website, `http://python.org/`

3.2.2.1 Views and Controllers

These frameworks already come with a basic logic for controllers and views. While the controllers' role is to process incoming commands the views build the actual code which is later sent to clients and rendered in the browser.

3.2.3 WebServer: Using Jetty as the Application Server

Tomcat[5] is considered the most prevalent Java application server[6] and is currently the default container shipping with Grails. Therefore it was used at the beginning of development.
But Tomcat is also known for being "heavy" in terms of configuration, maintenance and is likely to need more resources[7] opposed to Jetty[8] which is known for using less resources and being "lighter" regarding configuration and maintenance[9].
Additionally Jetty provide preemption of threads used for he handling of HTTP requests by so-called continuations which improve the scalability and higher performance when using long lived HTTP (see 3.2.5). Though current versions of Tomcat provide similar functionalities the library used for realizing this customization of HTTP requests does only support the continuations offered by Jetty.
With Backstage being in the early testing phase Jetty allows us to make quicker adaptions regarding the configuration and is supposed to need less resources in our testing environment.

3.2.4 Database: MongoDB

MongoDB[10] is an NoSQL database which is designed to be scalable and high-performant. Backstage uses MongoDB to store all data, like postings from users, as well as binary files, like slides.

NoSQL is a paradigm opposing to relational databases, such as MySQL[11] and IBM DB2[12]. NoSQL databases don't require fixed schemas which is the biggest difference. They follow other paradigms such as key-value-stores or document stores. MongoDB falls into latter category: It stores data as JSON-like, a specific representation of data, documents.

MongoDB puts a high emphasis on replication and sharding [19]. Two interconnected approaches of MongoDB to distribute the database across multiple ma-

[5] Apache Tomcat, `http://tomcat.apache.org/`
[6] PoweredBy - Tomcat Wiki, `http://wiki.apache.org/tomcat/PoweredBy`
[7] Tomcat vs Jetty - Stack Overflow, `http://stackoverflow.com/questions/302977/tomcat-vs-jetty`
[8] Jetty WebServer, texttthttp://jetty.codehaus.org/jetty/
[9] InfoQ: Google choose Jetty for App Engine, `http://www.infoq.com/news/2009/08/google-chose-jetty`
[10] MongoDB, `http://www.mongodb.org/`
[11] MySQL, `http://www.mysql.com`
[12] IBM DB2, `http://www-01.ibm.com/software/data/db2/`

chines. Failures can be better handled this way and it helps maintaining a highly available database.
Replication describes the ability of a database to duplicate itself on another database server. Hereby redundancy is created: Not only a single database contains the data and can serve requests but multiple can.
Sharding on the other hand is an approach to horizontal scalability: The database is partitioned based on logical specifications and distributed across multiple servers (shards). Requests to the database are automatically routed by the master server to the shards containing the specific dataset. In reference to the restaurant example the master takes the role of the receptionist by distributing the requests.
Internally both features fall back on each other as MongoDB uses sharding internally for replication while shards are replicated as well. Most configuration is done by MongoDB automatically and interferes only rarely with the application.

3.2.4.1 Providing the data from MongoDB to Backstage: Morphia

Backstage is designed under object oriented principles. This basically involves the logical entities of the real world and a set of their properties which are relevant to Backstage. For example the the users and lectures have a virtual representation. The resulting objects are likely to have additional properties and functions which are needed to provide the functionality of Backstage. These objects are stored within the MongoDB database.
To simplify the translation between the data object contained in the database and the virtual objects in the application, Backstage uses Morphia to automatically translate data into an object and saves changes of an object to the database.
Morphia maps back and forth the Java objects and the MongoDB documents so that changes made to our virtual object can be persisted in the database and the raw data from the database can be accessed in the virtual object, this is also called object-document-mapping.

3.2.5 Communicating with the Clients: AJAX and Views

The main task of Backstage is to provide the clients, students and teachers, an interface through which they can communicate. Therefore the interface must be likable and the functionality shall be usable with ease.
While the design of the frontend is not part of this work the connection between functionality in the frontend and the backend is. The rise of the Web2.0 was also the start of web applications becoming mainstream. The most significant change is within the interaction process: The user stays on the same page while interacting with it while previously every interaction led to a new page with the desired change.

3.2.5.1 AJAX and Reverse AJAX

This is achieved by Asynchronous Javascript and XML (AJAX) which uses JavaScript to asynchronously send a request without the browser changing the document. The

response coming from the server is per se hidden from the user but can be used to manipulate the document. From a user's perspective this is comparable to using an ordinary desktop application.
Looking at applications like Backstage where one can find a collaborative environment it is not only important what you do but especially what others do.
The communication between client and server follows the pull-principle, like specified in HTTP[13]. The client sends a request and the server responds. In our environment that is not practical at all since the client does not know when the server has new data ready for the client to fetch.
Since this problem is not unique for Backstage there are already workarounds which emulate a push-alike behavior: the server can actively say to its clients that there is new data. This is known as *Reverse AJAX*. The technical implementation is still based on usual AJAX.

Sidenote: WebSockets Since 2011 there is an officially proposed standard for enabling a general direct communication between server and client: WebSockets[14]. WebSockets creates a distinct communication channel between a single client and the server. This channel is fully bi-directional allowing both ends to send and receive information. In contrast to lower protocols like TCP it uses text messages instead of byte messages.
While the support for WebSockets is not consistent among the different browser vendors and the implementation specification is also not final yet[15] there are workarounds to emulate the targeted behavior.

3.2.5.2 DWR

Direct Web Remoting (DWR)[16] is a library which not only enables such Reverse AJAX calls. Furthermore DWR is a mature Remote Procedure Call (RPC) framework. The subset of functionalities Backstage uses allows the direct communication between JavaScript in the browser and the Java program on the server. Therefore the library is integrated at both ends: the client and the server.
For example internal functions of the web application can be exposed to clients. On the one hand DWR creates the necessary functionality on the client which familiarizes the client with the function signatures and the URLs under which those functions are actually available. On the other hand it creates those entry points in the application itself and ensures the requests are forwarded to the specified functions.

3.2.6 Lecture Context

The lecture context keeps objects available which are currently used in a session and therefore likely to be accessed often. This way all data which are relevant to a

[13] RFC 2616 - Hypertext Transfer Protocol, `http://tools.ietf.org/html/rfc2616#section-5`
[14] RFC 6455 - The WebSocket Protocol, `http://tools.ietf.org/html/rfc6455`
[15] The WebSocket API, `http://dev.w3.org/TR/websockets/`
[16] Direct Web Remoting, `http://directwebremoting.org/dwr/`

session are centralized. Which again simplifies the handling of lecture-wide events, such as processing an incoming post and delivering it to all clients.
Besides the organizational function of encapsulating the data of a session it also functions as a cache: By storing the information closely to the computational components it reduces the need to retrieve the same information multiple times from the database. The objects used in Backstage are persisted in the database. Therefore if an object is missing, it will be retrieved from the database. Usually these objects are disposed once there is no use of them. The need to retrieving them from the database multiple times is reduced by storing them in the lecture context.

3.3 Tasks and Information Flow

In showing the process of different tasks and examining which components are actually involved the pieces come back together. This involves the path different pieces of information take on their way to the client.
The focus of this chapter to show the illustrate the function of Backstage upon tasks that require actual aggregation of data compared to basic requests asking for a specific file. However these processes might be simplified to the relevant parts and suppose correct data at any stage.

3.3.1 Processing an incoming Post

The microblogging component is the basic communication function and therefore one of the most used in Backstage. On Backstage, most interactions result in a microblog post. Thus, not only the information exchange among students but also ratings and quiz responses are medelled as microblog messages. Hence this is the most important part regarding the processing of user data and for providing the Backstage specific functionalities.
Technically a post is a note containing a category, like question or answer, and some content. It is usually sent in by a end-user who positioned it upon a slide, so that these coordinates belong to a post as well.
DWR takes incoming posts and validates them before they are passed to the MicroblogBean whose function it is to store the post in the database and to determine which users shall receive this post. The database connection and the aggregat-

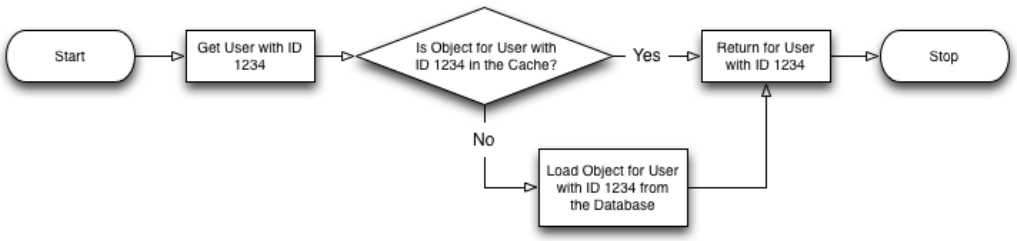


Figure 3.3: Retrieving an object from the Cache / Lecture Context

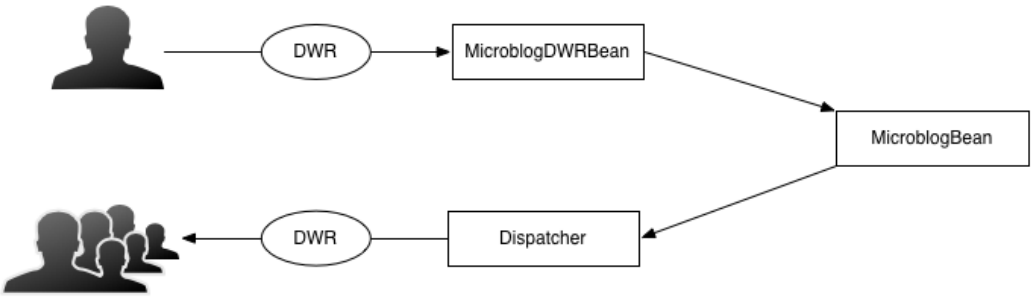


Figure 3.4: Default Path for Processing an Incoming Post

ing a list of potential recipients involved in this step are computation requesting additional resources and data and are therefore potential sources for lags. While sending the new post to all recipients can be done by the Dispatcher with the information computed by the MicroblogBean.

3.3.2 Creating Slides

Central part of Backstage are the slides of th lecturer. Therefore the lecturer is asked to upload slides beforehand as a PDF. In contrast to other fileformats for presentations such as Microsoft PowerPoint or Apple Keynote PDF is an open standard and there are multiple libraries allowing to import and export them. This is important as a PDF cannot be used in the presentation mode because browser don't allow to present and interact with PDFs the way Backstage wants it to do. So all slides are translated into plain graphics files which can then be served using standard HTML to the client.
Given the situation a new set of lecture slides are uploaded to the server Backstage takes the PDF and translates each PDF page into an image file which is then saved to the database.
The process of extracting the single slides and converting them is very resource-intensive. For the conversion process itself the server's filesystem is used at is necessary to create temporary files. These are then saved to the database which is a costly action because of amount of data binary files contain.
The decision of using the database to store the slides was based on the idea of having all data centralized at one place and because MongoDB offers native support for saving binary files[17].

3.3.3 A Quiz-Round

The quiz functionality is another important part in the concept of Backstage: The lecturer is able to create quizzes which can be started and evaluated during a lecture. Most basically these are multiple choice quizzes created by the lecturer ahead of the session. Students can then choose their option.

[17] GridFS - MongoDB, `http://www.mongodb.org/display/DOCS/GridFS`

This process uses parts of the above task: When the data for a quiz is saved there is a slide created containing the question and potential answers. Once answers coming in there is slide created containing the results.
A response to a quiz takes a similar route a microblog-post takes but ending up at the QuizBean where the results are computed and not pushed to all users.

CHAPTER 4

Taking Backstage to the Cloud

High availability and therefore scalability is a key factor regarding providing Backstage. But it is not yet possible to communicate specific requirements regarding the necessary resources. This is mainly due to a lack of reliable observations under real-world conditions.
Deploying the application upon an infrastructure yielding all the resources needed on demand is the prerequisite to ensure the availability for a prototypical implementation of Backstage. The cloud is an option providing such an infrastructure with adjustable resources.
The decision to take Backstage to the cloud in the first place was based on the requirement to have a backup solution for user studies that need to be conducted in real learning settings with larger audiences in order to show the usefulness of Backstage. Even the prototype of Backstage which is used in those studies must be able to cope with a large number of students.
While using the cloud with a strong demand in resources is costly, it is a valuable fallback solution to ensure the availability of the Backstage prototype during the time of the study.
On contrary running Backstage on a server provided by the university limits the scaling of Backstage strictly to the static specifications of that particular server.
Because of the unknown requirements and the different purposes of Backstage in the latter development process, there is a demand for a more flexible infrastructure which allows to dynamically allocate more resources. Yet this solution must fulfill a set of requirements in order to keep any needed adaptions as small as possible:

Java and used Frameworks In the first place it must be able to run the Java and more precisely it needs to provide deployment capabilities of servlet-based web applications like Backstage.

MongoDB To ensure the compatibility the solution must also be able to keep the database driver and access in their current implementation.

4.0.3.1 Additional Requirement: Local Data Storage

On the look for external solutions it became obvious that the data created with Backstage requires special attention due to privacy concerns.
Universities are public institutions and therefore subject to strict rules regarding data privacy. To adhere potential privacy restrictions the data gathered in Backstage needs to reside within the university network. For this reason the application stack needs to be separated into the data storage and the remaining application part that may be deployed to servers operated by external companies.
Rather then having the database, in which all data the users exchange is stored permanently, on the same machine as the application logic of Backstage, the database is now extracted and runs on a separate machine within the university.

4.1 What is Cloud Computing

Cloud Computing follows *Web2.0* as one of the most used buzzword regarding web technology. But what hides behind this rather imaginary term?
Cloud computing is a distributed computing paradigm: Instead of executing your computing on a single machine multiple ones are used. This idea isn't new at all but refers back to the 1960s and the idea of *Utility Computing* : Aggregating multiple machines and offering their resources, computing power and storage, through an interface.
It abstracts running an application or service from maintaining the necessary hardware. The configuration and monitoring and maintenance of the underlying hardware is completely beyond the customer's control. Virtually there are no boundaries regarding the use of resources, they are limitless.
An important advantage of cloud computing is scalability: Having a web application it is likely that the demand varies of the course of a day or throughout the week. Traditionally you are supposed to always have the resources at your back to handle peak times when your application is heavily used and under high load. This results in unused resources during the rest of time and therefore unnecessary costs.
A cloud-based application shall be able to respond immediately to different loads: Either by increasing the resources the application can access (Vertical Scalability, 2.2.1) or setting up another instance (Horizontal Scalability, 2.2.2) to cope with higher loads.
As *Cloud Computing* describes rather an idea then a exact implementation the definitions vary as widely as the actual implementations [7]. To cover the generic thought originally behind the cloud I refer to the definition of Foster[6]:

> A large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet.

4.1.1 Cloud Computing as an approach of horizontal scalability

The definition of cloud computing itself stresses scalability as one of the main goals. Different parts come to play when you take a closer look at the cloud and its nature. Let's have a look at the definition

Economies of Scale Originating from business studies the costs for offering a product can be split in fixed costs which are independent from the actual number of units shipped and variable costs which are directly dependent from the number of units. The basic statement is that with an increasing number of units the costs for a single unit decrease as the fixed costs can be split among more units [10].
Applying this to providers offering cloud-based services the costs for single users shrink because it allows full exhaustion of the available resources by dynamically allocating them to users.

Abstracted Using a cloud service you have no direct access to the hardware and can only use the provided interfaces. On the one hand this takes the possibilities of configuring every little detail down to machine level but on the other hand it also takes the responsibility to ensure the virtual machines are running correctly and reduces the complexity of handling different hardware.
Using the abstraction layer a cloud service provides you it ensures the compatibility regardless any changes behind the interface.

Virtualized In order to sustain the image of limitless resources regarding computation, storage and networking it is necessary that those are virtualized. That way it is possible to hide the actual technical implementation from the user [1].

Dynamically-scalable While the ability to scale an application might already be given when you are able to setup the application for usage at different scales. Being able to dynamically scale an application extends this to scaling while the application is already running and doing this within very short timespans. Ideally this can be done automatically and does not need human supervision.
On the provider side this requires the necessary interfaces to allocate and deallocade resources dynamically.

Managed As already mentioned earlier the technical duties regarding maintaining servers and ensuring correct network connectivity are up to the provider of the cloud service.

Cloud Computing reflects the idea of horizontal scaling and allow a dynamic resource allocation for web application. Looking at Backstage Cloud Computing is a possible solution in the context of thesis because the risks which are primarily the costs are manageable and at a minimum. Additionally it does not involve any upfront and binding investments other then a purchase of additional servers would.

4.1.2 Cloud and Traditional Hosting

Whenever I speak of traditional hosting or traditional environments you can picture it as a single server with fixed specifications running the application. Additional services such as the database might be located on a additional machine.
The resources are limited to the resources the single machine has got and adapting those to different situations is costly, e.g. buying new hardware and having a downtime while changing parts on the server, and time-intensive as the hardware must be changed and configured.
The flexibility and the idea of unlimited resources are the most appealing factors which might drive web applications towards cloud-based environments although there might be concerns against using the cloud [13].

4.1.3 Cloud and Grid Computing

Grid computing is not only similar to cloud computing as it also describes an approach towards distributed computing [5]: It is often an essential component in providing the technical infrastructure for cloud services[6]. The optimization towards a service available to a large audience is more specific for cloud computing. Grid systems on the other hand are often built and used for very specific and unique tasks rather then maintaining a continuous service [6, 20].

4.1.4 Types of Cloud Computing

The common trait of all cloud computing system is their function of shadowing the actual hardware behind a layer. So the users of such a system, like in this case the programmers of a web application, operate only on a virtual system.
Based on the open definition of Cloud Computing different types of services have evolved. These types are also a common categorization ("as a Service"). The following introduction of two different services also shows two different approaches: Amazon Web Services[1] (AWS) and Google AppEngine [2]. These providers are the two of largest ones but have divergent services.

4.1.4.1 Platform as a Service – Google AppEngine

One type of cloud computing providers offer a complete environment consisting of runtime environments of the desired programming language and often a database connection. Furthermore you often can access a set of custom APIs for additional functionality and administration tasks. This paradigm is called "Platform as a Service" (PaaS).
As example Google, one of the main competitors offering PaaS cloud computing, offers different environments based on the choice of the programming language and has integrated database interfaces which can be used like regular databases but the logical and physical internal representations are unknwon[3].

[1] Amazon Web Services, http://aws.amazon.com/
[2] Google AppEngine, https://developers.google.com/appengine/
[3] See https://developers.google.com/appengine/docs/whatisgoogleappengine

This approach has some clear advantages: 1) more computing power can be directly assigned as needed and the application can virtually grow without limits. 2) Database administrational tasks such as replication and sharding are done by Google. 3) Components up to the language environment are optimized by Google. These advantages come along with some restrictions regarding the choice of internal components as everything is already prepared.

4.1.4.2 Infrastructure as a Service – Amazon Web Services

At the other end of the spectrum there are services offering Infrastructure as a Service (IaaS). It differs to PaaS by not offering a complete software basis but only the infrastructure to use the resources for cloud computing. It is upon the customer to provide her own runtime environment.
Another internet giant provides such functionality: Amazon. Within their Amazon Web Services they unite a variety of different services so to be more precise Elastic Cloud Computing[4] (EC2) is their IaaS implementation.
The idea comes from the way you handle single machines: On EC2 you are able to set up a instances on which you can install and configure every component as you need it. All responsibilities for integrating the necessary services are up to the user. The advantage compared to traditional hosting is that flexibility. While having everything in an instance might feel static you can easily adjust the number and size of instances and access tools which allow you to automate this process.
Compared to PaaS you gain much more control over the software stack you use within your application and the configuration while keeping the theoretical access to unlimited resources.

4.2 Backstage in the Cloud

Cloud Computing is a way to ensure a high level of scalability for web applications. With the ability to easily adjust the computing power to changing demands it seems like the perfect foundation for Backstage and its various use cases.
Therefore this work takes now a try at taking Backstage to the cloud covering the process and arising questions and problems along the way.
This proof-of-concept ensures that Backstage can be run in the cloud and gives starting points for further work on fully integrating Backstage into a cloud based environment but does not cover the actual implementation of this. With regard to the topic and the other parts of thesis this would exceed the scope of this work.

4.2.1 Providing Backstage

The motivation behind looking into the possibility of running Backstage in the cloud is the idea of providing Backstage as a service. For example in situations when interested institutions don't have a sufficient IT department it to administer

[4] http://aws.amazon.com/ec2/

Backstage on their own, a cloud-based Backstage service could enable them to use it nevertheless.

4.2.2 Requirements

Backstage is already a fully grown application using a variety of third-party components[5] on the inside and has it's very own logic as it was built form scratch.
One constraint in the process of getting Backstage to the cloud has been to keep this state of Backstage as good as possible and to avoid larger adoptions for the initial deployment.

4.3 The process of taking Backstage to the Cloud

The process itself started with evaluating the approaches of different cloud services and then moved on to preparing and deploying Backstage for a test run in a cloud based environment.

4.3.1 Decision for Amazon Web Services

As already mentioned there are two major paradigms currently offered by cloud providers:

1. Infrastructure as a Service (IaaS)

2. Platform as a Service (PaaS)

Given the requirement to make as few adoptions as necessary to Backstage itself the decision went for an IaaS provider. This will give us the possibility to exactly determine which technologies and system components we chose in order to surround Backstage with them.
Amazon EC2 has already been presented as a IaaS provider but there is a larger choice of different services having a similar offer: CloudSigma[6], Rackspace[7], CloudFoundry[8] and 1&1[9]. Following the recent hype around cloud computing there are more and more services providing such functionality and this is only a small selection of them.
The decision towards s popularity there is a large set of resources, such as instructional material and in-depth reports regarding specific features, which might help to solve potential problems quicker.

[5] See Section 3.2 for a complete list
[6] http://www.cloudsigma.com/
[7] http://www.rackspace.com/
[8] http://www.cloudfoundry.com/
[9] http://1und1.de

4.3.1.1 Quick introduction to EC2

Elastic Cloud Computing (EC2) is one of multiple services Amazon offers around the larger subject of cloud computing within their department Amazon Web Services. Besides services for storage (S3) and databases (RDS, DynamoDB) EC2 is specialized in computing itself. The basic concept is built around instances which you can pick for your tasks and are available in different sizes.
They differ in terms of[10]:

1. Number of Computing Units
2. Available memory
3. Local Storage

Except for highly specialized instances all pricing is modularized and is calculated by the time an instance is actually running and the bandwidth which is used and optional services.[11]
The user has full control over these instances and can choose from a variety of operating systems like Ubuntu Linux and Microsoft Windows as a starting point or even use his custom operating system. Through remote access the user can control such an instance like a normal computer and configure it according to his requirements. Additionally those instances can be started, shutdown and copied to have multiple instances running at once.

4.3.2 Preparing Backstage

As the whole software environment on an EC2 instance can be fully customized all requirements Backstage has got can be met and no further steps are necessary besides the transition from a local database to a remote one as stated in Section 4.0.3.1.
The changed location of the database can be easily configured by adapting the `DataSources.groovy`. The database connectors automatically adopts the changed settings and tries to connect to the new location of the database.

4.3.3 Deploying Backstage into the cloud

Because of the raw software system a new EC2 instances comes with, the actual setup process covering all configurational steps equals the one on a local machine[12].

4.3.3.1 Configuring an EC2 instance

As mentioned above instances are the distinct units within EC2. Therefore it is necessary to configure only one for the initial deployment of Backstage. Later on a

[10] List of all instance types and their exact specification: http://aws.amazon.com/ec2/#instance
[11] http://aws.amazon.com/ec2/#pricing
[12] "Setting up Backstage": `http://www2.pms.ifi.lmu.de/blogs/thesisthinktank/?p=71`

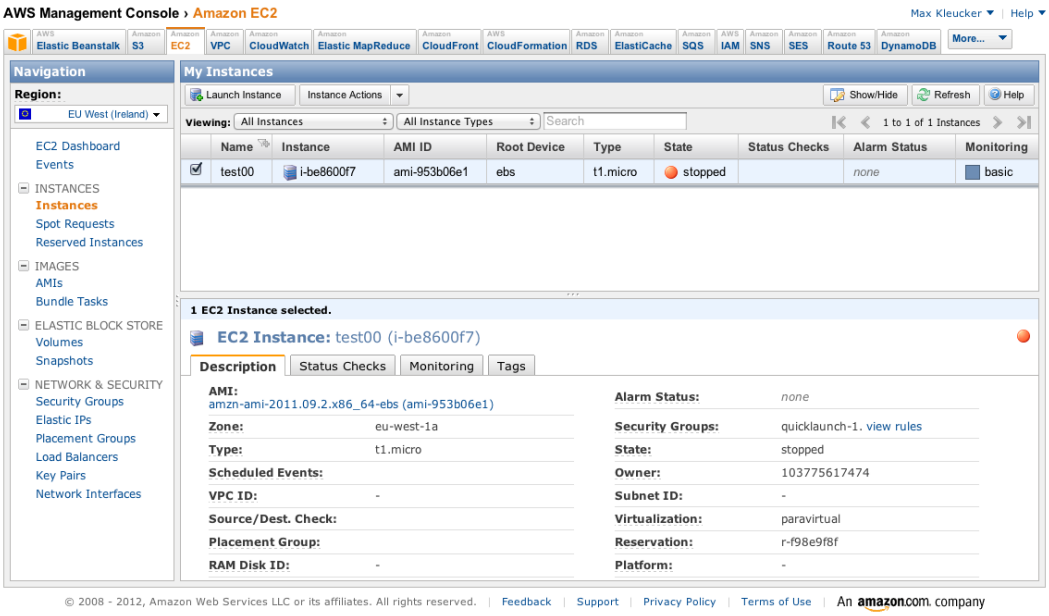


Figure 4.1: AWS Management Console. The webinterface for controlling EC2 instances.

snapshot of this instance can be used in order to start new instances without going through the whole configuration process.

For these testing purposes I decided to choose a `Micro Instance`[13] in the European data-center in Ireland. The instance comes with very low profile specifications but is offered for free use within AWS Free Usage Tier[14]. Nonetheless such an instance has proven to be powerful enough to run Backstage and it is sufficient to evaluate the basic functionality of Backstage in the cloud.

As a base system I decided to go with an Amazon Linux AMI which is a Linux imaged created and maintained by Amazon itself. During the time of evaluation the base image's identifier was `ami-e1ba8695`. The initial creation of an instance is done via the AWS Management Console (figure 4.1) through which all administrative functionality can be accessed.

Starting an instance is one of these functions and is necessary to gain access to the instance via SSH in order to continue with the process. To be able to run Backstage on the instance it is necessary to re-create the software environment which basically required the following packages to be installed on the instance:

- Java[15]
- Grails[16]

[13] http://aws.amazon.com/ec2/instance-types/
[14] http://aws.amazon.com/free/
[15] Java Programming Language, `http://www.java.com/`
[16] Grails, `http://www.grails.org`

- Jetty [17]

I found that using the automatic installers all these packages offer already cover the configuration necessary to run Backstage. Nonetheless checking the configuration ahead of a productional use is advised.

4.3.4 Running Backstage on AWS

Once all preparations are made and the EC2 instance is equipped with the necessary software components it is time to get Backstage onto the instance.
The Grails framework offers support for Web Application Archives (.war) which is a file format designed for running web applications and therefore contains all necessary files.
To use these advantages we create a `war` file for Backstage and load this one on to our EC2 instance. On the system itself the servlet container, Jetty, will automatically deploy and run the application.
Backstage is now up and running in the cloud.

4.3.4.1 Enabling external access

EC2 instances are per default only accessible over SSH for security reasons. Backstage on the other hand is designed to be used as a web application and therefore should be accessible.
Communication between your instance within the Amazon cloud and the web must be explicitly allowed through the Management Console. Under `Network & Security` inbound traffic must be allowed for the security group `quicklaunch-1` on port `8080` respectively `80`, depending on the configuration of Jetty.

4.4 Further thoughts on a cloud based Backstage

Backstage can be now accessed in the cloud and offers all its functionality. This situation offers little advantage over a hosted Backstage on a regular server: On the one hand it is only necessary to pay for the actual time Backstage is used in comparison to a server which has a fixed monthly price. On the other hand the switch towards a bigger instance with more resources can be easily done in order to serve more users.
So far it has been shown that Backstage can be run in a cloud based environment. Yet only few possibilities of the cloud are actually used which leaves further steps to integrate Backstage more deeply in a cloud-based environment.

4.4.1 Automated Scaling

For now scaling only exists of the opportunity to choose different sized EC2 instances. One goal would be to decentralize Backstage to a point where multiple instances are serving Backstage to the users.

[17] Jetty WebServer, `http://jetty.codehaus.org/jetty/`

An automated system decides on metrics like system load and round-trip time [12] whether there should be a new instance initiated. And vice-versa when load decreases overhead can be decreased by reducing the number of instances.
Having a single data store like the database which has been extracted from the application server is the first step into this direction. Yet further adaptions to the logic are necessary to reach the described state.

4.4.2 Quick Recovery

In cases when Backstage crashes or becomes unresponsive a supervising tool can be used which automatically disables a corrupted instance of Backstage and initiates a fresh ones.
In a single instance this behavior might still lead to an outage that is noticeable by the user. But the time such an outage lasts is decreased significantly compared to a Backstage environment maintained and supported only by human supervision.
In an environment where there are multiple instances of Backstage running like described in 4.4.1 this adds another factor for stability. Only connections originally communicating with a faulty instances need to be redirected which can be ideally done without the user notice any major hiccups or errors.
Most cloud provider offer additional services which cover exactly such control mechanisms. Within Amazon's EC2 system they are referred as Elastic Load Balancing (ELB)[18].

[18] Elastic Load Balancing - Amazon Web Services, `http://aws.amazon.com/elasticloadbalancing/`

CHAPTER 5

Improving Backstage

5.1 Problematic areas of Backstage

Before jumping right ahead and starting changing things at Backstage it is necessary to take a look at the challenges and problems Backstage faces currently and in the near future.
The goal is to create a more failure-resistant and scalable version of Backstage which suggests to solve current shortcomings as well as preventing potential problems and thereby get Backstage closer to the profile of a highly available web application. Considering the fact that Backstage is currently a prototypical implementation with the targeted use case of different studies this task requires foresight and solutions which are extensible beyond the current implementation of Backstage.
To narrow the actual field of focus, this part of the thesis is not about recreating Backstage without all problems found but about improving the current state by developing solutions for potentially problematic areas.

5.1.1 Internal Processes

The tasks introduced in Section 3.3 symbolize the basic functionality which should be functional under any circumstances. Apart from the general availability of Backstage the absence of these features, especially Sections 3.3.1 and 3.3.2, would cause at least a serious disruption in the designed way of using Backstage, if not even make Backstage unusable.
Therefore these are the starting points for tracking potential problems within Backstage.

5.1.1.1 Processing Posts

As explained processing new posts consists of two major functions: a) Saving a post to the database and setting all internal relationships. b) Aggregating a list of

users which are supposed to receive this post.
A scenario in which a few hundreds users are supposed to received an example post is not unlikely. While the costs for the first part actually are fixed, the costs for b) increase with every single user. Therefore the time it takes to process this post will increase as well.
At the same time the volume of posts written is likely to increase with an increasing number of clients. Both behaviors reinforce each other.
Additionally the MicroblogBean which deals with all these functionalities can only handle one post at a time. This results in unwanted effects like unresponsiveness of the whole application

5.1.1.2 Rendering Slides

It has already been mentioned that the translation of a PDF file containing the slides into single graphics is a resource-intensive process. Currently this process is started with the request the lecturer sends in order to create a session. This request does not get a response until the process is completed. While this behavior only affects the lecturer directly it is still far from being a desirable one.
Additionally quiz related slides put load onto this component. During a quiz intermediate results are periodically rendered as a slide and with closing a quiz session there is a slide of the final results rendered. Given a situation of multiple lecture sessions conducting parallel quizzes this might cause critical situation regarding the available resources. Because resources allocated for this particular function my lacking for other components.

5.1.1.3 Synchronous processes

All these functions have something in common: They are executed sequentially. As already implied the actions are invoked by the a request of the client. While the end-user expects that the task is completed, the request does not expect an immediate response containing a result.
But because of the current implementation the client waits for a response until Backstage has completely finished the task. In the meanwhile the user's browser is blocked until it receives the awaited response.
The longer a task takes the more inconvenient and opaque is this behavior to the user.

5.1.1.4 Aggregating Data

What comes to play at quizzes (3.3.3) and other parts is the aggregation of large data sets. To clarify the term itself, aggregation stands for the task of analyzing a larger set of data with the goal of deriving concrete values from them.
This is for example constantly done to provide statistics about the current use of Backstage for the lecturer. But it also comes into effect when quizzes are evaluated. The benefits for the lecturer in using Backstage lies in these possibilities and must be taken care off.

5.1.2 Extensibility

Beyond focusing on these issues Backstage faces in its day-to-day use, it is important for a successful work to also incorporate current plans for extending the functionality of Backstage.
In its current state all parts of Backstage are deeply integrated with each other. This makes it hard to integrate new functionality or components as a clear process structure is missing.

5.1.2.1 Further Plans for Backstage

At the time of writing this thesis there are different other projects evolving around Backstage which may be integrated into Backstage in the near future. They are used as examples to show the diversity regarding future features and different sets of requirements they might have towards Backstage.
Yet all assumptions about those extensions are not final but only based on short description of the particular project.

Statistics for Students All statistical functions are currently determined to offer the lecturer basic information about the students' information exchange.
It is currently planned to offer personalized statistics for students as well: The student's activity using Backstage shall be analyzed and visualized in relation to all other students. Those values need to be computed close to real-time for accurate information.
The challenges regarding an implementation of such a feature especially lies within the comparison among all students and time. It is conceivable that this is a very resource-intensive process which is preferable decoupled from the Backstage application core to avoid negative impacts on existing functionality.

Audio & Video Streaming On a different level it is planned to not only show the current slides but take it a step further and integrate video streaming. This way students are enabled to follow the complete lecture in real-time without attending it on-site.
While the implementation of video streaming is geared to run over a standalone solution it is necessary to connect both services. So the video stream is accessible within the interface of Backstage.
This example shows the potential need for some sort of API which allows external services to integrate within Backstage.

5.1.3 Challenges for Backstage

The illustrated challenges and potential sources for problems show that the processing structure in its current form lacks the flexibility to meet the different requirements which are expected in order to provide a highly available service and laying the foundations for extending the application.

Especially the synchronized invocation of heavily used parts of the application illustrates a potential bottleneck. Therefore, establishing a more asynchronous behavior must be part of the efforts to create a more robust Backstage.

5.2 Rethinking Backstage

The different components within the application core of Backstage are the present as Beans for the specific functionality. The different functions of these beans are directly invoked by others and this happens partially synchronized. A synchronized invocation causes a lock of the part of the application which is invoked and all related data. This lock avoids other race-conditions until it is removed and ensures a correct processing and the integrity of data this way. On the downside it also blocks the application partially leading to a decrease in performance
A potential solution is required to avoid these locks and offer a way to handle the processing asynchronously (5.1.3) while also ensuring the integrity of the results. Furthermore a possible solution should ease the process of adding new functionality to Backstage both internal and external.

5.2.1 A Different Integration Pattern: Messaging

Messaging is such a pattern for internal communication [9] between components of a web-application. It is an additional component which replaces direct communication between different other components. Instead of calling functions of other components directly in order to fulfill a task, this instruction is now handed to the messaging service which will pass it to the targeted component.
This way the requesting thread ends once the message is passed to messaging service. The time this thread is locked decreases.

5.2.1.1 Messaging in a Restaurant

Going back to the example of the Chinese restaurant the classical behavior would be each customer lines up in front of the kitchen and must be served before the next customer can place his order. In this case the process of serving guests runs completely synchronously: Only one customer can be served at any given time. Therefore the risk of longer waiting times for the other customers increases which my lead to discontent among them.
The ordinary process within restaurants already embodies solutions for this situations: The customer gives her order to the waitress. The waitress on her behalf forwards the order to the kitchen. Where it is filed on a tray containing all open bons. The kitchen starts processing the order once enough resources are available and then hands out the cooked meal to the pass. The waitress picks the dish up again and takes it to the guest. Until the waitress serves the meal the customer can move on to do different things in the meanwhile. As all included actors are working asynchronously on their specific tasks it is possible to serve multiple guests simultaneously.

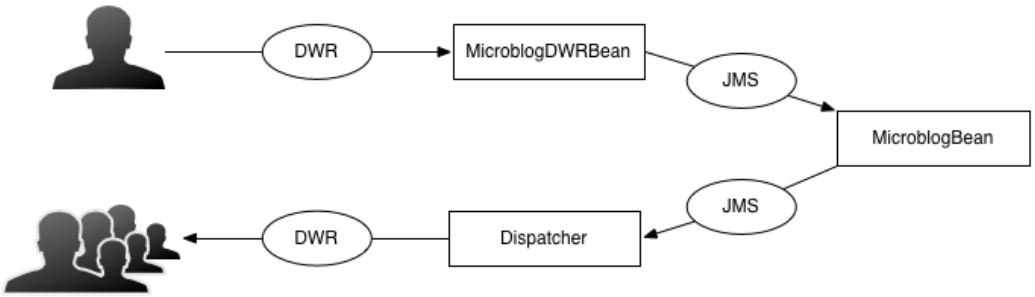


Figure 5.1: Path for Processing an Incoming Post in the JMS version

Messaging would take the role of the bon tray and the pass in Backstage: All jobs for the kitchen – the application core of Backstage – come in on the bon tray which will be subsequently worked off and then the meal – the computed result – will be handed over to the pass.
Another requirement that is here already fulfilled but is a potential challenge: All jobs must be able to be represented in a simple structure. While at programming one usually deals with complex objects it is hard for a waitress to get these down on her block of paper.

5.2.1.2 Introducing JMS and ActiveMQ

The Java Messaging Service (JMS) is an API designed for enabling messaging in Java applications. It is part of the Java Enterprise Edition[1] and freely available.
Yet JMS only provides a standard for the different parts of messaging and their behavior. It is no fully functional implementation. ActiveMQ[2] is such an implementation of the JMS standard but also offers additional proprietary features.
The decision in favor ActiveMQ among other JMS services is based on various factors. On the one hand ActiveMQ is open-source and licensed under the Apache2.0 which allows its use for Backstage. On the other hand it is a grown and popular project which suggests a certain level reliability among offering a rich set of features.
Especially looking at the extensibility of Backstage ActiveMQ offers connectors for other programming languages like Python, C or PHP. This way the the work around implementing messaging into Backstage supports also the extensibility regarding future external components.

5.2.1.3 Messaging Objects

To create a messaging infrastructure it is important to familiarize oneself with the different terms and objects and the basic transaction which build the basic idea around messaging.

[1] Java Messaging Service (JMS), `http://www.oracle.com/technetwork/java/jms/index.html`
[2] Apache ActiveMQ, `http://activemq.apache.org/`

The terms and names used in this part are learnt from the Java Enterprise Edition API[3] and may vary on other messaging platforms.

JMS provider In this case ActiveMQ is the actual provider of the JMS functionality and handles all connections and information.

Message A message consists of three parts: the head containing all information necessary for the JMS provider to process the message. A set of properties which can be used to filter messages through the application. The body of the message contains the actual payload: all the data the recipient needs to process a message successfully and potentially additional information on what shall be done with the data.

MessageProducers and MessageConsumers As the names may indicate MessageProducers create and messages while MessageConsumers receive and process them.
While the producer can send messages at any given time, the consumer must be permanently connected to the JMS provider in order to receive messages.
Yet the roles do not exclude each other. This way a consumer can also send messages as a producer during reacting on a message.

Destinations: Queues and Topics Messages are not directly addressed to a specific consumer but either to a queue or a topic. The difference between these two is the way how messages are preserved and forwarded. The messages in a topic are retrieved by all endpoints registered to the topic and will be discarded once all endpoints accepted the message. Those in a queue will only be delivered once to a single consumer and be discarded afterwards.
Both are gathered under the generalization of a Destination which describes the address a producer sends its messages to.

Connections and Sessions These objects are necessary to connect to the JMS provider and to establish a communication channel. While a connection simply establishes the communication path between messaging service and the messaging user, the session registers the user at the service and is necessary for the actual communication.
While they are inevitable in the use of JMS they will be neglected in following graphs visualizing the JMS structure.

5.2.1.4 Benefits of using JMS

Correctly used JMS brings along some advantages which might be useful for Backstage.

[3] Package: javax.jms, `http://docs.oracle.com/javaee/6/api/`

Asynchronicity The messaging process is designed to run asynchronously and improves the handling of long-lasting requests: Once a task has been committed to a JMS queue there is no further control about the message for the sender. Thus the request terminates here and can be replied.
The processing of the message is invoked by the JMS service which ensures that it will be still handled.

Decoupling While all components within Backstage reference each other to some extent, JMS requires to drop most of these references on the parts which shall be invoked via messaging.
Because accessing a component that listens to a queue should not be accessed directly in order to avoid malfunction, all references to this component should be either entirely dropped or modeled via JMS.
Furthermore request and response are not directly connected anymore. Originally invoking a function on another object (request) waits its completion and is likely to use the returned value (response). Messaging takes away this direct possibility. With messaging there is no intended way to wait for the response. Instead the response must have enough information within itself to ensure a correct further processing.

5.2.1.5 Message Driven Components

Decoupling leads to a state in which the components controlled by the messaging service are able to perform with at best no external dependencies.
Additionally they are stateless: You can determine the result in advance and it does not change over time. Therefore all information that are necessary are contained within an incoming message and the component only relies onto these.
This state is often referred to as message driven. Java EE also contains a standard for *Message Driven Beans*[4]. Yet this goes beyond the targeted use as it would require to load additional Java EE libraries which we want to avoid at this stage in order to keep Backstage slight.
If you find the term message driven bean in this work then it is rather related to the basic principles then to the Java standard.

5.2.2 Integrating JMS in Backstage

ActiveMQ which is chosen as the JMS provider for this implementation brings along various libraries to simplify the integration. However we decided to keep the actual implementation above the connection level as close to the JMS standard as possible. This way Backstage remains independent from ActiveMQ and can easily adopt any other JMS provider.
More precisely ActiveMQ's `PooledConnectionFactory` takes over the handling of connections and optimizes connections to the ActiveMQ service by reusing con-

[4] What Is a Message-DrivenBean ?,
`http://docs.oracle.com/javaee/5/tutorial/doc/bnbmd.html`

nections instead of permanently opening and closing connections. All other functionality is done by directly inheriting and implementing standardized functions.
In the context of this work messaging has been integrated in the illustrated processes of users creating new posts and interacting with quizzes as well as with the conversion of slides into graphics. Due to the large similarities in the actual implementation the implementation is explained on the example of the MicroblogBean which has the job of accepting incoming posts from users, saving them into the database and prepare them for the broadcast to the users.
The decision for integrating messaging at these components in the first place is based on their importance for a seamless running Backstage and their high load and the resulting risk of being a bottleneck as already explained.

5.2.2.1 Preparation: Extracting the Entry Points

The server-client communication with DWR should remain unchanged and the implementation should focus on the application core of Backstage. Therefore the work starts after the DWR has passed the request to one of the exposed entry points (3.2.5.2).
These entry points used to be direct functions integrated into the logic of the single process. Ahead of integrating JMS those directly accessed functions were moved to particular objects only dealing with incoming DWR calls and then forwarding them to the actual processing logic.
Through this step the beans which contain the actual logic for processing data become stateless as well. Previously DWR which itself relied on the connections between server and client determined the state of the whole bean.
Overall establishing this new layer allows us to maintain a better overview over the whole application as all entry points are collected in one location. Potentially this would also allow us to replace DWR while leaving the actual functionality of Backstage unchanged.

5.2.2.2 Preparation: Uncoupling the MicroblogBean

As the DWR dependent part of the previously overarching MicroblogBean is now extracted, the new MicroblogBean contains only the domain logic for processing single posts.
Still the MicroblogBean contains a huge set of external references (Figure 5.2). Uncoupling these references reduces additional lock times, for example the lecture which has been resolved through another bean and was therefore dependent from the availability of that bean. Generally this is a step getting single beans closer to provide their whole functionality by fully relying on the incoming messages.

5.2.2.3 Preparation: Defining a Message Format and a Queue

All data within a message must be represented as a plain key-value pairs. While the keys must be represented as strings the values can contain different data types. It is recommended to limit the values to basic datatypes, like integers and strings,

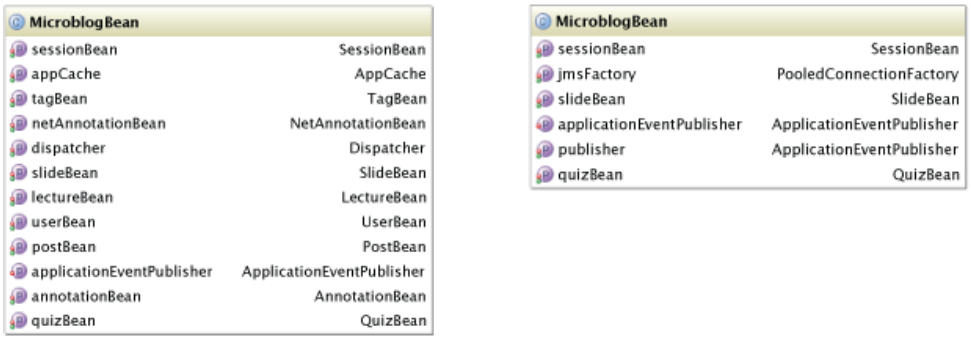


Figure 5.2: References of the MicroblogBean before (left) and after (right) the uncoupling

Identifier	Value	Content
`post-text`	String	Raw text content of the incoming post
`post-label-category`	String	Category of the label (e.g. question, answer)
`post-label-value`	String	Potential value of the label
`post-location-x`	Integer	X-Coordinate of the post
`post-location-y`	Integer	Y-Coordinate of the post
`post-location-slide`	Integer	ID of the slide on which the post is positioned
`refersToId`	String	ID of a potential parent post
`user`	String	ID of the user
`lectureKey`	String	Key of the current lecture
`sessionID`	String	ID of the current lecture-session
`type`	String	Internal variable for further extension

Table 5.1: Fields of a message containing all informations for processing incoming posts.

in order to keep the datavolume low. But also lists can be directly passed through JMS as long as they only consist of plain values on their behalf.

The message format for representing an incoming post is defined with the fields and values in Table 5.1.

Queue-naming In the larger context of the web application messaging is currently meant to be replace parts of the internal communication. The general naming pattern for all queues is `incoming-<COMPONENT>`, where `<COMPONENT>` stands for the internal component meant to execute the job represented as message, e.g. `incoming-microblog`.

5.2.2.4 MicroblogDWRBean: Forwarding Messages to the Queue

The MicroblogDWRBean is invoked whenever there is new post coming in via DWR. This condition causes the MicroblogBean to be stateful by having this constant connection to the DWR component. Regarding messaging it's its job to transform these posts into messages according to the previously defined format.
To be able to communicate with the JMS service it connects itself to it on instantiation. To send a message it enriches the incoming post with the meta-information defined in 5.1 then submits the message to queue by creating a temporary `MessageProducer`.

5.2.2.5 MicroblogBean: Reacting on Messages from the Queue and Pushing Them to the Clients

The MicroblogBean implements the interface `MessageListener` which explicitly requires to implement the function `onMessage()`. This function is called by the JMS service whenever a) there is a message in the queue and b) the bean has processed the previous message and is idle.
The MicroblogBean has been previously made stateless and is therefore able to act on the information coming via the message alone. But also has to instantiate all object by their incoming identifiers again in order to create a valid `Post` object.
The second part of the process involves aggregating a list of all recipients eligible to receive the post. Both this list and the post are sent to the Dispatcher for delivering them to the clients. The implementation of the logic for sending it to the queue `incoming-dispatcher` equals the MicroblogDWRBean.

5.2.2.6 Dispatcher: Sending the Post to the Clients

The role of the dispatcher is comparable to the one of the MicroblogDWRBean. It is connected to the DWR component because it needs to able to access the communication channel.
It is a `MessageConsumer` like the MicroblogBean and receives messages via JMS which are then further processed and sent to the clients using DWR.

5.2.2.7 Configuration: Parallel Beans through Messaging

This flow equals the direct communication and has simply replaced the direct access with JMS. From this point a decrease in performance is quite likely given the added latency because of the longer communication path.
The mentioned scalability of stateless beans and the benefits of failure resistance can be achieved by adapting the configuration of the MicroblogBean. Originally all beans are instantiated and configured at the start of the web-application[5].
The `DefaultMessageListenerContainer`[6] allows to set an average number of concurrent instances as well as a limit. If the number of messages in the queue

[5] #14.3 Runtime Spring with the Beans DSL,`http://grails.org/doc/latest/guide/spring.html`
[6] Spring Framework API 2.0,
`http://static.springsource.org/spring/docs/2.0.x/api/org/springframework/jms/listener/DefaultMessageListenerContainer.html`

outweighs the number of current instances, new ones are automatically added. Additionally it allows to set a timeout limit after which beans are destroyed. This adds an important factor for the failure resistance within Backstage: given the situation the MicroblogBean can't fully process a new post and an error doesn't allow it to continue this mechanism also handles such pending instances.
For an example configuration of the MicroblogBean which connects to JMS via a central `jmsFactory` and listens to the queue `incoming-microblog` with three concurrent instances and a maximum of 10. Additionally we specify to create a new instance once two instances are idle by setting `idleConsumerLimit`.

Listing 5.1: Configuration of a MessageListenerContainer for the MicroblogBean

```
1 microblogListenerContainer(DefaultMessageListenerContainer){
2     connectionFactory = ref('jmsFactory')
3     messageListener = ref('microblogBean')
4     destination = ref('incoming-microblog')
5     concurrentConsumers = 3
6     maxConcurrentConsumers = 10
7     idleConsumerLimit = 2
8 }
```

5.3 Analyzing the modified Backstage

To sum the implementation process up: The internal architecture has been revised in different iterations.

Disentangling the logic By separating the components which directly communicate with the clients from the domain logic the possibilities to work with Backstage on different levels has been increased.

Reducing dependencies The grown dependencies in-between different objects and beans has caused illogical and duplicate links which not only have been resolved. Instead they were reduced ensuring full functionality and creating a better maintainable Backstage.

Introducing JMS With the introduction of messaging for dedicated parts of the application Backstage gained a more robust internal structure around important and intense operations. The additional communication component allows internal scaling and asynchronous operating.

5.3.1 A More Robust Backstage

The mentioned steps created a more robust version of Backstage. On the one hand this reduced complexity comforts the developers as it is easier to navigate through the code and to maintain an overview of it. On the other hand it helps Backstage itself: The responsibilities for specific functionalities are divided more clearly reducing the risk of interferences.

The asynchronous execution of tasks actively avoids lock times. Times during which the application cannot do anything because single parts are still occupied.

5.3.2 Better Extensibility

Extensibility has been introduced as a reason for changing the communication process. Messaging can be used as the foundation for further developments around Backstage: Additional features such as video-streaming can be connected via JMS as it exposes its interfaces externally.
So the actual realization of the streaming can be solved completely independent from Backstage. While it might be necessary to have a component for displaying the information within Backstage, all communication and control can be done over JMS [16]. This would save the work for creating an additional API.
The same pattern is valid for extended analytics: As statistical computation on large data sets can be considered resource-intensive, messaging actively simplifies the computation on a different machine and the therefore necessary communication with Backstage.

5.3.3 Potential Next Steps Regarding Messaging

JMS is used only for very specific tasks and components by now. Taking the idea of messaging further there are additional areas where its use would make sense or areas where the integration could be improved.

5.3.3.1 Status Reports on Tasks

Currently messages are handed over to next component but there is no response-channel to the message omitting instance. Feedback is not that critical for less important tasks such a processing posts where a lost post only affects the author of this post. Given experiences from the user he might recognize such failures after a certain timeframe.
Converting slides should be seen differently: On the one hand a potential failure affects a larger audience and it might interrupt the teaching. On the other hand this is a job that can take a longer time therefore feedback whether the task is still in process or there has been an error help to evaluate the situation.
For Backstage it might be practicable to create some sort of status center which watches the processes and which is notified on both error and status reports.

CHAPTER 6

Conclusion

At the close of this work the results will be evaluate and how they agree with the original goals. Furthermore a outlook will be given on potential further work emerging from this thesis.

Goal of this work has been to improve the failure resistance of Backstage on the way to increase the scalability of this digital backchannel system. Beyond analyzing potential solutions and the theory behind them it also covered the basic implementation and examination of those using the example of Backstage.

At this point the work has been successful in adapting Backstage towards a failure-resistant system by changing internal process operations and introducing messaging. Furthermore it showed the ability to run Backstage within a flexible and easily scalable environment as the cloud.

Based on the primary evaluation and what has been learnt during the implementation process allows further research on the following topics:

Integrating messaging across all components So far messaging is only used for a small subset of functionality. Onwards from this state it can evaluated which other parts are eligible to be changed towards a messaging based component.

Outsource resource-intensive operations on other machines While this thesis used messaging to outsource the communication the application itself still remains coherent and monolithic. Forthcoming steps could be to consider each functionality as a separate application and re-implement them using the most appropriate programming language and tools. These applications could be deployed to customized servers providing an also specialized environment.

Messaging can be then be used to wire these distributed parts together as described in this thesis.

Evaluation of JMS regarding security Introducing an external component for communication added an additional starting point for potential security breaches. Beyond integrating ActiveMQ into Backstage there were no further steps taken regarding the integrity of messages. Instead we assume only correct and valid messages will enter the messaging system.

Enhancements for Backstage in a cloud-based environment Backstage has been deployed to the cloud in this thesis. Yet it uses just very basic features the cloud, and more specifically Amazon Web Services, offer. Further integration steps can range from using automated load-balancers which dynamically scale Backstage horizontally to developing deployment processes which allow to automatically deploy new versions of Backstage into the cloud.

Bibliography

[1] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al., *A view of cloud computing*, Communications of the ACM **53** (2010), no. 4, 50–58.

[2] André B. Bondi, *Characteristics of scalability and their impact on performance*, Proceedings of the 2nd international workshop on Software and performance (New York, NY, USA), WOSP '00, ACM, 2000, pp. 195–203.

[3] M. Fayad and D.C. Schmidt, *Object-oriented application frameworks*, Communications of the ACM **40** (1997), no. 10, 32–38.

[4] M.E. Fayad and A. Altman, *Thinking objectively: an introduction to software stability*, Communications of the ACM **44** (2001), no. 9, 95.

[5] I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke, *Grid services for distributed system integration*, Computer **35** (2002), no. 6, 37–46.

[6] I. Foster, Y. Zhao, I. Raicu, and S. Lu, *Cloud computing and grid computing 360-degree compared*, Grid Computing Environments Workshop, 2008. GCE'08, Ieee, 2008, pp. 1–10.

[7] Jeremy Geelan, *Twenty-one experts define cloud computing*, Cloud Computing Journal **2009**, 1–5.

[8] Athula Ginige and San Murugesan, *Guest editors' introduction: Web engineering&#151 an introduction*, IEEE MultiMedia **8** (2001), no. 1, 14–18.

[9] G. Hohpe and B. Woolf, *Enterprise integration patterns: designing, building, and deploying messaging solutions*, Addison-Wesley Professional, 2004.

[10] P. Krugman, *Scale economies, product differentiation, and the pattern of trade*, The American Economic Review **70** (1980), no. 5, 950–959.

[11] Kangchan Lee, Jonghong Jeon, Wonseok Lee, Seong-Ho Jeong, and Sang-Won Park, *Qos for web services: Requirements and possible approaches*, W3C Working Group Note **25** (2003), no. November, 1–9.

[12] Anbazhagan Mani and Arun Nagarajan, *Understanding quality of service for web services*, IBM developerWorks (2002), 1–8.

[13] D. Molnar and S. Schechter, *Self hosting vs˙cloud hosting: Accounting for the security impact of hosting in the cloud*, Workshop on the economics of information security, 2010.

[14] Jeff Offutt, *Quality attributes of web software applications*, IEEE Softw. **19** (2002), no. 2, 25–32.

[15] David Oppenheimer, Archana Ganapathi, and David A. Patterson, *Why do internet services fail, and what can be done about it?*, Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4 (Berkeley, CA, USA), USITS'03, USENIX Association, 2003, pp. 1–1.

[16] M.P. Papazoglou and W.J. Van Den Heuvel, *Service oriented architectures: approaches, technologies and research issues*, The VLDB journal **16** (2007), no. 3, 389–415.

[17] Soila Pertet and Priya Narasimhan, *Causes of failures in web applications*, Tech. report, 2005.

[18] Filippo Ricca and Paolo Tonella, *Detecting anomaly and failure in web applications*, IEEE MultiMedia **13** (2006), no. 2, 44–51.

[19] C. Strauch, U.L.S. Sites, and W. Kriha, *Nosql databases*, Lecture Notes, Stuttgart Media University (2011).

[20] L.M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, *A break in the clouds: towards a cloud definition*, ACM SIGCOMM Computer Communication Review **39** (2008), no. 1, 50–55.