LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
Department "Institut für Informatik"
Lehr- und Forschungseinheit Programmier- und Modellierungssprachen
Prof. Dr. François Bry

Projektarbeit

RESTful Backstage

Johann Kratzer
johann.kratzer@campus.lmu.de

Betreuer: Alexander Pohl
Verantw. Hochschullehrer: Prof. Dr. François Bry

Zusammenfassung

Die Verfügbarkeit einer breiten Palette an verschiedenen Interaktionsmöglichkeiten erscheint für moderne Anwendungen, vor allem jene die im Kommunikationssektor ansässig sind, nahezu unverzichtbar. Durch die Unterstützung vielfältigster Eingabegeräte wird nicht nur eine maximale Zugänglichkeit gewährleistet, sondern ebenso die Akzeptanz solch einer Software gesteigert. Um diese Verfügbarkeit auch technisch sauber und modular umsetzen zu können, sollte bereits in einer frühen Phase des Entwicklungsprozesses die grundlegende Designentscheidung festgelegt werden, welche speziellen Rahmenbedingungen für die gesamte Architektur gelten sollen. Einer der gängigsten Softwarearchitekturstile für Web-Applikationen, welcher eben diese Konzeptionen vorgibt ist der Representation State Transfer (kurz: REST)
Die vorliegende Arbeit beschreibt sowohl in konzeptioneller, als auch technischer Hinsicht, die Erweiterung eines bestehenden Software Prototypen mit dem Namen “Backstage” um eine REST Schnittstelle. Backstage offeriert großen Hörerschaften einen, auf mirco-blog Technologie basierenden, digitalen Kommunikationskanal, welcher dem Dozenten auf vielfältige Weise zu direkten Rückmeldung verhilft.
Zieht man die große Vielfalt zwischen verschiedenen Benutzern in Betracht und mit welchen technischen Möglichkeiten sie heute mit Webanwendungen kommunizieren können, so erscheint ein vereinheitlichtes und nahezu Endgerät unabhängiges Konzept wie REST am besten geeignet, diesen Anfordernissen gerecht zu werden.
Zusätzlich behandelt diese Arbeit im zweiten Teil eine weitere praktische Anwendung des REST Konzepts welcher vor allem für die Entwickler von Backstage von speziellem Interesse ist. So wird es durch die Erweiterung um REST zum ersten Mal möglich, unter Echtzeitbedingungen Belastungstest durchzuführen, welche wertvolle Informationen über die Performanz des Systems und dessen potenzielle Engpässe unter Volllast liefert.

Abstract

Offering a wide range of different interaction modes seems nearly indispensable to modern, especially communication-based, applications. The support of multiple input devices not only provides a maximum of accessibility, but also often enhances the acceptance of this kind of software. To realize these demands in a modular and technically clean manner, certain basic design concepts that determine general conditions of the whole architecture already have to be conceived in (an early stage of) the development process. One of the most common styles of software architecture for web applications providing the desired standard guidelines is the Representational State Transfer (abbr.: REST).
The present work describes the extension of an existing software prototype called "Backstage" by a REST interface in both a conceptual and technical manner. "Backstage" provides digital backchannel communication for a large audience based on mircoblogging technology that is able to support the lecturer in terms of getting instant feedback in manifold ways.
Considering the huge diversity among people in regard of their technical possibilities to interact with web-applications today, a concept like REST, which offers unified and almost non terminal dependent communication, seems best suited to meet these requirements.
Additionally, this work will cover the benefit for the developers of "Backstage" gained by accessing the software via REST interface for evaluation purpose. In particular, stress tests, which can be performed under real usage conditions for the first time, become available and can deliver valuable information on the capabilities, the performance and potential bottlenecks of the system under heavy workload.

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig angefertigt, alle Zitate als solche kenntlich gemacht, sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

München, 24. Januar 2012

. . . .

Acknowledgements

First and foremost, I want to express my gratitude to Prof. Dr. François Bry who supervised this thesis and, in doing so, established the basis for me to get insight to one of the most powerful and formative concepts in the field of web technologies. I also want to explicitly thank Alexander Pohl, who supported me to the best of his knowledge and provided his creative suggestions and beneficial ideas in numerous discussions throughout the whole project. May my work on this project be a profitable and useful tessera for his doctoral thesis. A special gratitude of a quite different nature is dedicated to J.H., the love of my life. She is the source of creative inspiration, the best mental support and the most sincere backup I could ever wish for. Finally, I want to dedicate this whole work to my recently deceased mother to whom I owe my deepest gratitude. You will always be in my heart of hearts.

Conventions

This work includes several code samples to illustrate the presented facts. Since some of these examples consist of omissions, abbreviated syntax and/or pseudocode, a few conventions have to be introduced to ensure a proper comprehension. Even though these examples are nearly self-explanatory, the following section summarizes the used conventions in detail.

- Squared brackets including three dots (i.e. "[...]") indicate that one or more parts of code that are not relevant for the actual consideration have been omitted. They could further indicate space for optional items not listed in detail.
- Squared brackets including italic type indicate a placeholder. It stands for one individual item that belongs to a group of the type denoted in italics.
  For example, "[*someFruit*]" could be replaced by "apple", "banana" or "strawberry". This placeholder is often used if the concrete syntax or type of an element can not be further determined or is of subordinate importance in source code examples, as well as in graphs.
- Examples within the implementation chapter (i.e. Chapter 6) are written in the Groovy programming language.
- Examples describing queries include a protocol directive (e.g. "GET") followed by an identifier (e.g. "URI").
- The source code sample in Chapter 3 is written in XML syntax including above-mentioned conventions.
- The API related overview of URI composition in section 4.3 is presented in a tree structure. It distinguishes replaceable parameters (i.e. placeholders or wildcards) and fix parameters (i.e. expressions that are used within a URI exactly as denoted) of a URI by coloration. While a yellow node is a fix parameter, a green node represents a replaceable parameter. Note: No valid URIs can be extracted from this illustration, it just offers a quick overview to visualize the presented structures and relations of the API. To get concrete schemes of the URIs please refer to the annexed Appendix I.

Contents

1 Introducing REST

Before focusing on certain details of the respective project, it is a fundamental requirement to take a closer look at the components appearing, especially to provide a basic knowledge of the REpresentational State Transfer (abbr.: REST). Roy Thomas Fielding coined this expression in his dissertation named "Architectural Styles and the Design of Network-based Software Architectures" [Fielding 00]. An extensive explanation becomes even more important, regarding the fact that REST is not a kind of definite standard solution to solve a certain problem in computer science, but first and foremost an architectural style for designing web applications, almost even a way of thinking and finally a general approach to core functionalities of the World Wide Web itself. In fact, most facets of the World Wide Web we know and use today, fundamentally originate from the main concepts of REST [Erenkrantz 06].

1.1 Basic classification and context

To decrypt the meaning behind every component of the acronym "REST", it is necessary to understand the surroundings at first. Basically, REST is naturally settled in the context of client/server communication [Fielding 00]. Regarding a web service as the field of application, very generally speaking, it exists a requesting part (the client) demanding some information from a source or destination (the server) that contains the data. This data of whatever nature is called “resource”. According to Fielding, the resource is “the key abstraction of information” in REST [Fielding 00]. REST is further defined by a set of constraints - specified in the course of this chapter - that shape the architectural style and assign the environment of the resources. Any other specifications besides these constraints can be devised arbitrary and adapted to respective requirements of the concrete topic.

1.2 Constraint I: Uniform interface

The most considerable and central part of REST is the use of a uniform interface. Since this constraint consists of several important individual concepts, its explanation is interspersed throughout the whole section. By presenting different aspects in this non-sequential order, one can comprehend the coherence and elaborate the definition of REST gradually.

- Uniform identification scheme
  A first idea of the uniform interface is to identify the resources through one uniform scheme. So, every resource being present possesses its own identifier consisting of exceedingly diverse location descriptions, which can establish a direct access. The most well-established identifiers are the Uniform Resource Identifier (abbr.: URI) and the increasingly often used Internationalized Resource Identifier (abbr.: IRI)[1]. At this juncture, it has to be pointed out clearly, that an URI (or IRI) in general may not be mistaken as only being some kind of link to a homepage in the context of the World Wide Web (cf. term Uniform Resource Locator [abbr. URL], which is in fact a subfield of the URI term). The meaning is much more extensive, which is expressed by the ability, not only to refer to simple data in web service context, but to any kind of abstract or physical resource [Berners-Lee, et al. 05].

- Uniform set of directives
  A further component, building the uniform interface, is to establish manipulation of resources by a uniform set of actions. For instance, these commands can be treated just as an imperative form of predicates in clause grammar, where the resource would act as the object. An example from everyday life would be the instruction "BUY some COOKIES !".

[1] The only difference between these two types is the presence of the full Universal Character Set in case of IRI

BUY is obviously the instruction, *COOKIES* build the resource. To create a RESTful query, one of the directives has to be combined with one of the identifiers of the resource, instead of the resource itself. Trying to apply this to the above-mentioned, very simplified example the result would potentially be "BUY some COOKIES from the top shelf in corridor seven of the grocery store [...] etc.". In a human conversation this example truly sounds a bit odd, but a straight forward and quite precise location description seems rather conventional, keeping the incredible amount of resources in the World Wide Web in mind. With only a bit more concrete information on specific syntax of the directives and the available REST services, the client is already able to emit a request for information to the server, which is compliant to the uniform resource and consequently RESTful in this respect.

- Communication by representation
  In principle, the only thing the server has to provide is a logical unit, which understands the request, processes it and returns the corresponding data to the client in an appropriate format. Satisfying another uniform interface guideline, the information exchanged within the communication is never the resource itself, but a client adapted representation of it. Such a representation is holding the current (if the query is a retrieval) or future (if the query is an update) state of a resource and is intended for transport purpose. The selection of a client-appropriate format is only possible if multiple representations of the same resource exist, which, in general, is a common case (e.g. the representation contains the same content, but can be provided in several languages or different formats). Methods to decide which variant to deliver are the client's explicit postulation - for example by attaching the desired format of the answer to the requesting URI (cf. file extension) or to the request itself (cf. protocol header information) - and more dynamic alternatives, for example the so called content negotiation between client and server, where both parts agree upon a suitable representation [Holtman, et al. 98].

1.3 Differentiating REST

For now, further uniform interface constraints shall be put aside to examine the difference between the intention of REST and established web technologies.
The concepts mentioned so far do not seem to be very cutting-edge. Being a bit more familiar with the technical aspects of today's web standards like the Hypertext Transfer Protocol (abbr.: HTTP), the idea of REST being nothing else than this ordinary network standard suggests itself. A closer look at a typical HTTP Request can exemplify this:

```
GET http://de.example.org HTTP/1.1
```

Listing 1: Standard HTTP query

Regardless of the protocol version negotiation [i.e. HTTP/1.1], this request almost fulfills every condition stated so far. *GET* serves as an instruction, the URL behind seems to be giving some information about where the desired resource resides - just like an identifiers does.
Though a certain similarity cannot be denied, it is actually more about the famous chicken/egg problem, than comparing congruent approaches. First of all, it is important to remind that HTTP and REST take place on quite different levels. While REST is an architectural approach and more a code of practice defining certain constraints [Fielding 00], HTTP is a concrete protocol with fixed abilities [Fielding, et al. 99]. Just like every concept is realized by adapted practices, REST can also be carried out by the use of different protocols other than HTTP. So, some parts of HTTP are one possible way to implement REST, and apparently the most common one today. For example, the request methods of HTTP (i.e. "*GET*", "*POST*", "*PUT*" and "*DELETE*") are

for many users synonymous with the REST predicates (i.e. the uniform set of actions to manipulate resources). Given their huge prevalence in any REST implementation, they seem to be highly suitable [Richardson, et al. 07]. This reveals impressively the capability of HTTP concerning REST. However, it is not surprising that Roy Fielding is also one of the cofounders of HTTP [Fielding, et al. 99] and that REST was initially developed to re-establish structure in HTTP 1.1 [Erenkrantz 06].

Another reasonable conclusion at this point raises the question vice versa, if every web service using HTTP (which basically is one of the main cases in the World Wide Web) is a RESTful service? According to Fielding's intentions, this would be the perfection [Fielding 00], but the reality differs [Berners-Lee, et al. 05]. It becomes much clearer if one takes a look at another illustration of the uniform identifier guideline. The initial question is, how a uniform identifier has to be composed? For instance, there is a request from a client to a server that hosts school administration data. Trying to get some information about a specific pupil named John Doe could initially look like Listing 2 depicts. A corresponding answer is shown in Listing 4.

```
GET http://myschool.com/REST/pupils/JohnDoe
```

Listing 2: Simple RESTful query

```
GET http://myschool.com/REST/pupils/findpupil?name=johndoe
```

Listing 3: Alternative query

Another possibility to access John Doe is to use given functionalities on the server side (cf. Listing 3). The "*findpupil*" reference used in this alternative is exposing the access point of an internal algorithm or functionality located on server side. What already looks somehow like a method call in object-orientated programming, is designed as a Remote Procedure Call (abbr. RPC). As its name implies, an RPC in client / server context is the request for execution of a non local subroutine. An RPC is often accompanied by variables, which are passed to the processing instance by a query string (cf. "*name=johndoe*" in Listing 3). The computation on the server side is performed in a concealed way that keeps the user unaware of the detailed processing. During this activity, certain side effects may occur, too. Typically, in a final step an answer containing results of the execution is returned to the calling part [Bloomer 92]. This variant of information retrieval would not comply with the RESTful requirements. To differentiate REST from calling server side scripts, in a simplified, but yet appropriate way one could again reference clause grammar and determine that an identifier shall only be made out of nouns of whatever nature. Every part of information integrated in an identifier should be designed according to a resource, or a set of them, directly addressable, which in general is syntactically embodied by a noun. The only verbs (i.e. instructions) desired are the predefined set of directives in front of the identifier, like the "*GET*" used in both requests. Otherwise, the uniform interface constraint would be violated due to neglecting the "uniform set of actions"-part. Today, REST and RPC are the most common approaches designing a web service [Richardson, et al. 07]. In general, there is no standard solution. REST is admittedly efficient regarding large-grain hypermedia data transfer, but might not be optimal for any specific case, where RPC or a completely different style may be more suitable [Fielding 00].

```
[...]
<pupil [someLinkFormat] = "http://myschool.com/REST/pupils" >
  <name [someLinkFormat] = "http://myschool.com/REST/pupils/JohnDoe">
    JohnDoe
  </name>
  <grade [someLinkFormat] = "http://myschool.com/REST/grade/6">
    6
  </grade>
  [...]
</pupil>
[...]
```

Listing 4: Answer from RESTful server

1.4 Constraint II: Statelessness

Besides the uniform interface constraint every RESTful service has to carry out, one very important fragment is accompanying the characteristics of REST detailed so far:
A RESTful request should never contain any data that is referring to a state the client is currently in, while sending the request. Every detail needed from the server to answer the request properly has to be included in the request, replacing any persistent client state. Hence, REST does not support features like dropping cookies on client side or more complex concepts like sessions to store information on the client beyond a single request. However, the server on his part is holding a state manifested by the representation of a resource. On one hand, this property burdens the client with a lot of responsibility to create a successful request, but on the other, the interaction with the server is finalized by returning an answer (from a server-side point of view), without the need to store additional, overarching data. The big advantage of this concept is a better overall performance by saving bandwidth and minimizing server-side application state, and the intense enhancement regarding the scalability.

1.5 Defining REST - State transfer and hypermedia

With this information in mind, it is now possible to decipher the abbreviation REST completely: An extension of the above mentioned example can finally help to clarify the term REST.
Maybe, one already queried John Doe, got the answer posted above (cf. Listing 3). This answer represents the current state of the communication. Within, the presence of further representations to related resources is observable. This additional supply marks the last part of the uniform interface constraint, which is the “hypermedia as the engine of application state” [Fielding 00]. If one, for example, is further interested in a list of John Doe’s class mates, one could easily follow the link contained in the answer above (cf. “*http://myschool.com/REST/grade/6*”) by dropping another query. Doing so, the current state gets changed by following the new identifier, switching to a new resource and finally getting a representation containing information on the new state. To make a long story short, one just performed a REpresentational State Transfer.
Though being a very powerful and comprehensive concept, REST still appears as a very theoretical and abstract. To draw a more practical picture one could think of a “forest” data structure in a semantic way as the application level of REST (note: the forest itself is not equal to REST). Imaging every tree within this forest represents a server containing information (and thus in terms of REST can deliver representations of resources) REST defines the overall structure to interact with this forest. It determines how a user should access specific trees, it demands servers or content providers in the broader sense (i.e. the creators of new trees) to offer independent representations as well as hypermedia to create a semantic network of related information (note: REST does, of

course, not imply the logic of the referenced hypermedia, which means that REST is not interconnecting trees by its own). Finally it offers practices that characterize the precepts of the interaction (cf. statelessness and the upcoming performance constraints). The World Wide Web we know today would be an example of such a, admittedly very immense, forest.

1.6 Performance constraints

While the uniform interface constraint as well as the statelessness constraint specify and shape a RESTful interface, there are a few other constraints that mainly rely on an effective overall performance. Their listing and brief explanation should complete the insight into the defining concept of REST [2].

1.6.1 Constraint III: Cache label

The response to a RESTful query has to be implicitly or explicitly labeled as cacheable or non-cacheable. A cacheable response concedes the right to reuse this response for later, equivalent requests. The benefits and deficits of using cache have already been adequately defined in numerous cases of application and thus be not further specified here.

1.6.2 Constraint IV: Layered system

Each processing component between the end points of communication (i.e. client and server) should not be aware of every other component along the way. Only components residing on the same logical layer should have mutual knowledge. A client, for instance, should not be cognizant of every transit station across the network. The only thing he should know is the end to end connection to a server. This abstraction helps to simplify the components by moving infrequently used functionality to shared intermediaries. Also other features like load balancing can be performed covertly along the way. By following the self descriptive and independent design paradigm of the uniform interface for interchanged messages, each component, no matter on which logical layer, has enough information to process the data properly.

1.6.3 Constraint V: Code on demand

The last constraint is not a necessary feature of REST, but an optional one. The Code-on-demand functionality can be used if a client has access to a set of resources, but not the know-how to process them. In this case, a request to the server for the code representing the know-how is sent. The server returns that code (in general JavaScript) and finally the client executes it locally. Due to the reduced visibility, Fielding declared Code-on-demand as facultative. A noteworthy citation in this context is a comment of Fielding, relating this constraint to a modern concept for web programming: "[...] I still think it is amusing that, six years later, people started referring to AJAX as a new thing" [Fielding 08].

Summing, it can be constituted, that each of these constraints improves the scalability and the allocation of responsibilities at the expense of administrative overhead.

1.7 Why use REST?

Considering the significant number of constraints and principles to comply with if designing an application the RESTful way, the legitimate question arises, what the profit of this liability is? Especially if an already designed substructure is given and REST should only be applied to form an additional interface (like in the present case), it seems to be much easier to convert existing

[2] If not stated otherwise, the following itemization is completely based on [Fielding 00]

implementation logic straightforward and relinquish to drudgingly squeeze it into a new format. Though an additional effort is undisputed, there are still various reasons to retain REST.

- Longevity
  By REST's inherent convention that the same request always ends in the same result, a certain level of reliability is created. Further, the use of different representations, notifications upon failure or success of a request (cf. status codes in case of using HTTP) and the included hypermedia intended to traverse related resources can increase the probability to fit future demands. Including the following aspects, a RESTful service's persistence is clearly promoted.

- Control of side effects
  From a client's perspective, practically no side effects occur if a RESTful query gets executed. In contrast to an RPC, the user is always in full control of his current state and the performed tasks arising from a query. The concept of statelessness and the postulate of clear semantics of the uniform interface directives can be accounted for this. It has to be mentioned in this context that application intern side effects on server-side that are not related to the RESTful communication may still emerge.

- Simplicity
  By applying the uniform interface constraint, the communication gets considerably simplified. On one hand, the small set of directives defines clear instructions, which offer few tolerance in terms of misinterpretation or even incomprehension. On the other hand, the self descriptive style of the identifiers mostly reveals the intention of the query immediately. By the increase of diverse and multifaceted data amount, simplicity gains more and more in importance and is a basic element of functionality.

- Scalability
  The better the scalability of a system, the better is its ability to support very large amount of concurrent requests as well as a sporadic number of them in comparatively equal amount of average time per request. There are two principles, which are commonly determining the degree of scalability: On the one hand the execution of a request in one go, which means without the need to interrupt or to pause (potentially caused by an additional information retrieval), is conducive. According to that, an a priori completed request like the present, self-contained RESTful query is beneficial for this purpose. On the other hand, the efficient use to distribute and split tasks to use the system's capabilities to its full capacity at any time improves scalability. Disregarding technical requirements to cope with heavy load, REST also enhances scalability on a conceptual basis by applying "cache label"-, "layered system"- and "code on demand"-constraints (cf. section 1.5). A high level of scalability is virtually indispensable concerning the almost non-predictable extent of resources in long lasting web services.

2 Inventory taking of Backstage

In order to create a useful and accurate representation of the application's functionalities made available by applying REST, it is important to initially identify and categorize the existing features to obtain a certain structure and/or model, which can be transformed into a new interface. Having an abstract pattern in mind, while mapping the features, results in a much clearer and comprehensible reproduction than a simple depiction of every single application function. Therefore a quick overview of the prototype's abilities - primarily disregarding concrete technical implementations - is necessary to be able to describe the design of the new interface in a more detailed manner. In general Backstage is composed of three semantic core components:

1. The messaging system
   As mentioned at the beginning, Backstage's main communication part is a microblog-like messaging system, which provides electronic exchange of views among the users. The obtained messages are shown in a constantly updated list in the main window. There is a possibility of setting and using anonymous and private messages.

2. The feedback system
   The central idea of giving feedback to a lecturer is realized by this set of functionalities. Feedback can be given as well for any of the messages, as in form of annotation for the ongoing lecture. Often these feedbacks can be attributed with predefined values and are further explicated by additional textual content, concreting the annotation. For example, an annotation type could be "*not clear enough*" and the annexed message would provide further information on an existing problem, concerning a topic presented by the lecturer. Another noteworthy part of the feedback system, besides post rating and lecture annotation, is the quiz component. It enables an "on-the-fly" reflection of lecture contents, which means the ability for a lecturer to initiate a prepared survey covering recently recited topics while a lecture is running. The quiz itself is being presented as multiple-choice question - though other kinds of quizzes are to be implemented - which is inserted into the user interface and offers dual feedback. On the one hand, the possibility to test one's own comprehension of the subjects taught is given by returning correct answers and results of every quiz to the listener [Kay, et al. 09]. On the other hand, important information on the absorption of the topic, supporting the lecturer, can be collected at the same time.
   The presented systems are logically surrounded by a third component:

3. The lecture system
   Due to the obvious fact that some listeners not only participate in one single lecture, a basic scaffold to separate, manage and administer different lectures for the two sides - audience as well as lecturer - is provided. For each existing lecture a single instance including the two, above mentioned, core components is given in Backstage. However, only one instance can be active and used at the same time, resulting in a specific lecture session.

These three systems, each with its respective specification depending on the user group, include every component to interact with the application. That way these systems are presumably the ones to be supported by the interface with their complete lineup of methods. Before investigating the process of mapping the systems to the interface, a brief overview of the REST implementations of famous microblog applications should sketch current approaches and give a short discussion of their interfaces.

3 RESTful equivalents

If one compares the degree of RESTfulness of existing microblog solutions, like the famous Twitter or Google's counterpart Jaiku - open sourced by the Google company in 2009 [Google, Inc. 09] - at first sight not everything of REST's actual intention remains. Both applications are offering a REST interface using HTTP features as underlying technology, but their functional range and market position is clearly influencing the extent of them: While Twitter is featuring almost every potential action, providing 87 different identifiers [Twitter, Inc. 11] - disregarding their miscellaneous characteristics using different request methods - Jaiku only offers 7 URIs [Google, Inc. 11], which can be accessed both as REST, as well as remote procedure call. Obviously, Twitter can be regarded as a kind of reference product and thus should be investigated to further details:
Despite the huge amount of Twitter's URIs, the nomenclature of most of them is kept quite simple and has huge affordance[3], but due to the extensive use of verbs within in most instances, these identifiers are more representing remote procedure calls than addressing any resources (cf. Listing 5).

```
http://api.twitter.com/1/friendships/create/id.format
```

Listing 5: Exemplary Twitter URI

To pass more information, but keep the referencing short, a lot of optional settings can be added to these URIs using parameters. These additional variables are arranged within a URI by attaching the query string "*?paramName = paramValue* ") (cf. Listing 6). In consideration of REST's constraints there seems to be no violation of these terms at first sight. Though, a total compliance remains doubtful for another reason: To fulfill the uniform interface constraint an action on a resource has to be carried out the exact same way at any times. Having alterations in form of options, the same resource and request method can potentially cause different results. One might think that most of the methods, applied in a different context, result in ambiguous outcome (cf. POST command applied to a resource), but in all of these cases the difference is based on the "what" (which is the dissimilarity of resources) whereas the options are influencing the "how" (which is the request method). If another resource is used (which means the "what" changes) the final result surely can be different, but if a directive is influenced by the additional passed option instead of using another one with other well defined semantics (e.g. POST instead of PUT), a uniform interface can't be warranted.
This consideration is only partially correct for Listing 6 (i.e. the section marked by the "*follow=true*" statement), because the "*screenname*" variable in fact seems to address a fix resource. In this case the options are determining both, "what" and "how". This deviation is caused by the general absence of a resource denotation in the basic URI, which, as a matter of principle, is not desired in REST. However, a conform URI using options does never profit in terms of clarity and accordingly should not contain any of them, though it is not an explicit proscription of REST.

```
http://api.twitter.com/1/friendships/create.xml?screenname=101010&follow=true
```

Listing 6: Exemplary Twitter URI containing added options

A tradeoff between featuring most of the available actions that the application logic has to offer and keeping the documentation simple and usable at the same time, is obviously a challenging task, but if in the final accomplishment the REST principles partially disappear and REST seems to be more like a cloak by name, the use of such a portentous term seems questionable. On the

[3] In this context "affordance" pertains to a concept in Human-Computer Interaction and describes the situation where an object's (sensory) characteristics intuitively imply its functionality and use[Foraker Labs 11]

other hand concerning Fielding's meaning of the importance of the different guidelines, it is probably better to provide overburden and grubby identifiers, than losing the statelessness [Fielding 00]. By supporting not only a few of Twitter's basic properties, but a full range of quite dynamic tasks, a complete statelessness of the implementation apparently can't be warranted, though most standard cases are totally compliant in this point [Makice, et al. 09]. Other demands, like the "uniform set of actions"- constraint, are being used quite properly by the equivalent, if they are used solitary. Still, this could be owed more to the standard usage of HTTP's request methods, which are completely fulfilling this requisition anyway. Though, additional directives within the URIs (cf. "*create*" in Listing 5/6) rebut and superimpose this concept repeatedly.
Though not completely detailed at this point, in summary it turned out that the reviewed REST interfaces of current microblogging services at most follow the basic principles of REST, but barely one of them is totally compliant. The resulting interfaces leave the impression of a compound solution. Such being the case, the more accurate definition for contemporary implementations would be the "REST-RPC hybrid" approach [Richardson, et al. 07].

4 Creating RESTful Backstage

The following section provides detailed information on the developed REST Application Programming Interface (abbr.: API) for Backstage (cf. Appendix I), not only focusing on functionality and explanations of the respective design steps, but also expounding the thinking process behind to reason them. For this purpose, this chapter has been divided into three subcategories. At first, the technical fundament of Backstage, wherein the interface has to be incorporated, is reviewed. The following passage covers different aspects influencing the design of the interface, before the last section details the resulting API.

4.1 Technical remarks

The first object of research delivers insight into the technical concerns. By taking a brief look at the underlying technology REST is making use of, the complex interactions of REST and RPC concerning the integration process of the interface, and finally giving some thoughts to the updating mechanism for the client side, three practical topics covering the most relevant technical matters are being discussed in this section.

4.1.1 Used technologies

The widespread HTTP as one of the most common and proven technologies in the World Wide Web today is applied by Backstage's REST interface, too. Especially the most basic subset of HTTP's request methods (GET, PUT, POST and DELETE) is best suited to be used as uniform interface (cf. Section 1.3). The lack of potential alternatives and the fact that Fielding adapted REST in combination with HTTP as prime solution [Fielding 00], rather derives the usage to a notable fact than to a grave deliberation on options. Further beneficial arguments for this choice come into play taking a closer look at security concerns later on.

4.1.2 Combining RPC and REST

So far, the two architectural styles RPC and REST have been introduced as counterparts (cf. Section 1.3). Now, another technical angle, where these two concepts have to work together, shall be presented at first, before examining the structural look of the interface. However, it has to be clearly pointed out that this combination is not performed on the API level - this would be a REST-RPC hybrid style (cf. Chapter 3) - but on the intersection point of the RESTful API and the RPC-containing main application. Assuming that there is a completely designed REST API, a next step is to discuss how to integrate this part into the existing Backstage system. Because of potential influences to the look arousing from the decisions defined at this point (cf. Section 4.3), it seems to be appropriate to operate in reversed order.

If the application logic offers simple methods like getters to retrieve certain data snippets, the mapping to the accordant REST methods is an easy task and would not need further explanation, but the structural complexity, caused by a very dynamic application like the present one, is deeply aggravating this procedure. Backstage, in its current appearance, is written in the web framework "Grails", which is based on the "Groovy" programming language. Grails' fundamental application architecture is the Spring Web MVC framework. In a rough outline it is based on the well-established Model-View-Controller design pattern, more precisely a web application specific modification of the MVC, the so called "JavaServer Pages Model 2" [Oracle Corporation 10] is the underlying architecture (cf. Figure 1). A further detailing of this architecture's concrete specifications will be disregarded at this point for the sake of simplicity. For the upcoming explanations the knowledge of MVC's fundamental idea is totally sufficient.

The main concept of the MVC pattern is to completely separate application logic (i.e. the "model" or "service class" in this context) from its presentation (i.e. the "view") by the use of intermediate

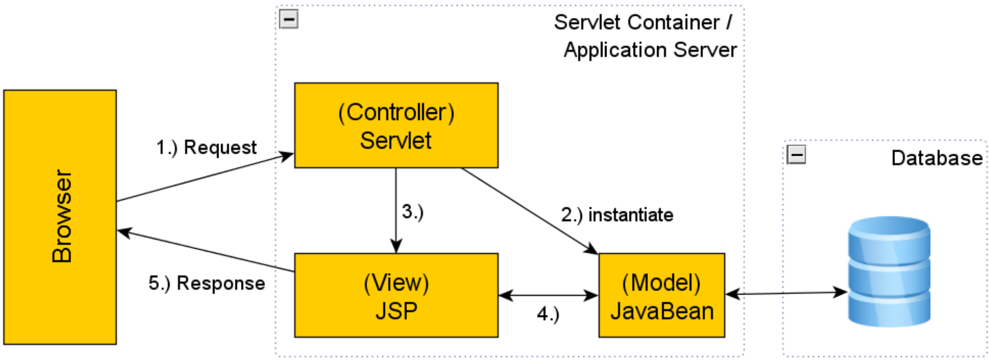


Figure 1: MVC-based JSP Model 2 design pattern

control structures (i.e. the "controller") [Reenskaug 03]. Accordingly, a request to get a certain service should always be intended to a controller that handles the further processing by accessing the adapt service class. Directly querying the model is not intended using MVC.
So far, there seems to be no problem in adapting this to the REST interface. By creating a REST controller, which can interpret the REST queries and redirect them correctly, every demand seems to be fulfilled [Rocher, et al. 09], but unfortunately there are further obstructions concerning the subsequent processing.
According to the instructions of the controller, the control flow is passed on to certain service classes that compute the requested data (i.e. apply business logic) and return the computed results to the controller, but this process is totally tailored to the particular needs of the web application and, for example, many service methods do not comply with the present data of a RESTful request. The possible options to encounter these problems would either be the implementation of own REST service classes, or the adaption of the existing ones to work ambivalently. Because of the almost congruent effects of the web application's logic and the REST handling logic, the option to double source code by operating with a separated logic for each of them seems superfluous. Hence, to avoid overhead in many ways, the adaption is the better solution here. Again, the effort performing adjustments appears limited.
A much bigger issue results from the additional use of Direct Web Remoting (abbr. DWR). DWR is a RPC library, which, among other things, enables an application server to call JavaScript functions on the client dynamically. Additional Java Script Code is running in the client's browser to send requests and perform dynamical updates of the web page. This whole feature is also known as "reverse-AJAX" or "Comet". The big advantage of this technology is that every web browser, if it is able to handle JavaScript, can use it innately, which means without the need for further plug-ins [DWR 11]. The problem in this case is that the logic handling DWR initiated requests is also intimately connected with the application logic in the service classes.
To have a short recap: The really crucial point here is, how to bring application logic - designed for the use of RPC and the handling of dynamic web context - and the REST interface together and still keep the number of redundant tasks small, or in other words, how to merge a stateful RPC and a stateless REST approach.
An analysis of the workflow starting with a client's query can help explaining this: Based on the REST principles, the use of states beyond the ones realized by representations of a resource is undesirable and therefore it has to be assured, that every request being made, contains every data to fulfill the request (cf. Section 1.4). This can be achieved by providing a qualified API (cf. Section 4.3). After the sending of the request, the REST controller on the server side takes over control and, for now, delegates the request to the application logic. At this point, the REST request will fail. The reason for this is that the existing application logic needs information concerning

the state of the user in form of a HTTP session. If there is no data about the current state, the request cannot be handled further simply because certain parameters the application logic needs to compute the results are not present. Fortunately, in the present case the information extracted from a session consists of information, which is integrated in the requesting URI and accordingly can be mocked up easily (e.g. user name, intended lecture,...). Thus, it is possible to circumvent this problem by inserting an adapter, which can assemble volatile information that matches a session's data, from the given URI data. This task can, for example, be carried out in the REST controller. If a controller needs any session information, the information will be present. Even if the session concept seems to be abused a little bit at this point, it is still a practical solution, which can be achieved without much effort.
The last part of the workflow, except for the reshipment of the answer to the client, is the content negotiation part of the respective service method, which is basically the only modification, where additional code has to be inserted. The reason for this is the multifarious kind of formats of the service method's return values. With it, marshaling of the data to be output for REST is so different, that none of the existing code can be reused for this purpose. To reduce the modification effort and keep the source code as modular as possible, it would make sense to put the control flow back to the scope of the REST controller, but the diversity of output types could impede the complexity notably. Furthermore, the DWR affiliated features within the service classes have to be excluded from the REST compliant workflow to complete the adaption. However, bypassing the DWR context in all its aspects can also originate tricky issues, which often are not perceptible at first sight. For example, if a user posts a message using the REST interface, this message not only has to be processed properly by the application, but also the notification of the other "subscribers" of the lecture using reverse-AJAX, has to be performed. The application itself uses another advantage of DWR to determine the subset of affected users - the ones who are currently following the lecture and who are also intended to receive this message - and to notify them accordingly. At least this DWR task has to be included somehow in the REST processing. Otherwise no other user would get notice of the changes a REST user performed. However, only the clients using the web application have to be taken into consideration regarding this notification. The REST clients on the other hand can be disregarded, because their information passing mechanism is working differently (cf. following section). A detailed technical description of the applied solution for the remarked problems is provided in the upcoming implementation chapter (cf. Chapter 6).

4.1.3 Updating the client

Dropping the whole state afflicted scenario, it is plausible that an updating routine loses its trait. It is not possible to exactly determine the relevant changes that took place since the last retrieval. Yet to keep the client up to date, the REST interface is executing a simple kind of periodic polling, where any information matching the request is transmitted, regardless of already being sent to the client earlier. Though this process is obviously not high-performance, it is still sufficient.
While brainstorming, the idea of using "HTTP long polling" as an expedient alternative emerged. Long polling constitutes a technique at which a client's HTTP request is not answered immediately, but the connection is held open by the server as long as no change of the requested data, in comparison to the client side's version, is detected. Once there is an update, a response including the latest set of data is sent back. Usually the client side reopens the connection to the server right after receiving the response in order to not miss an update [Loreto, et al. 99].
At first, this concept does not really seem to fit REST, thinking of its apparent concern of states. This way, it is not surprising that even DWR, among other options, carries out long polling [DWR 11]. However, the very idea of this technique, which is reducing the amount of data exchanged between server and client, is profitable for REST. But it has to be mentioned here, that long polling is definitely not belonging to REST's scope of operation. The deliberation on this topic has to be understood as a concept, which is intended for the development of a client ap-

plication and still beneficial for REST at the same time. According to this, long polling can be integrated as an option to a RESTful request[4]. In contrast to a normal REST query, the server will not answer until an update of the requested resource is taking place. Compared to the way DWR is performing long polling, no dynamic scripting and for this reason no state, representing the current client information, is present here. After a change of the resource, its representation is completely re-transmitted to the client. Realizing an up-to-date client application, there is still a huge gain by only transmitting updates and not constantly polling potential changes. Thus, the load is displaced from the clients to the server, which has to manage persisting open connections to the clients now. By having working client software in the end, users probably will not notice big differences between simple and long polling, but concerning the limitations of mobile devices pertaining to process data, any option to avoid communication overhead on the client side seems to be worth a try. Especially those parts, which are rarely expected to be subject to changes - like the quiz component - can highly benefit from this alternative.

[4]This feature is not yet part of the API documentation

4.2 Remarks about design

One might think, that the design process, in contrast to the technical aspects mentioned above, reveals much more creative freedom than being tied to prerequisites of an existing solution. Nevertheless, underlying technical concepts and their demands often limit the creational process equally. In the present case a well-formed naming, self explanatory identifier content and stateless resource addressing are factors to cope with. A few structural aspects regarding the design not in particular, but as a whole, are hereinafter presented.

4.2.1 Creating thorough URIs

While the existing Backstage system in its web application mode, in contrast to the RESTful mode, offers a workflow that starts with a dialog for selection and/or management of different lectures, followed by activating a specific one to start the main program and from this point operating within this lecture, this type of state-afflicted procedure is not ideal to fit the demands of REST. It should be reminded here, that a RESTful service is characterized by independent, self-contained queries. Consequently, there is no sequence to emulate the application's workflow. Whether a RESTful query belongs to a certain lecture must be stated explicitly. An adjustment to this problem is to carry out every detail for lecture identification in every REST request. Otherwise, the application logic would not be able to determine, which lecture the performed action relies on. This causes at least an prefix phrase[5] (as shown in Listing 7) as a part of the URI, in which the lectureID is a dynamic wild-card character, respectively replaced by valid lecture identification.

```
[...] /lecture/[someLectureID]/ [...]
```

Listing 7: URI prefix to lecture related content

Almost all REST-demanded data, except for the lecture settings and general informations (cf. “admin” and “info” URIs), always refer to a specific lecture. Consequently, the identifying term has to be included in all of these requesting URIs. Most appropriately, lecture concerning representation should be put in front of the desired URI, because of classification reasons and the important character of the lecture system as a main construct. Framing a concept that follows the idea of having a lecture identification prefix as the beginning of each URI, a hierarchical (i.e. tree-like) building of the complete URIs suggests itself. Accordingly, the URIs can be built the same way as file system paths, mapping the respective functionalities to different nodes (cf. Section 4.3 and Figure 2).

Imaging the resulting API, that specifies the REST interface in particular, every URI contained starting with the same specification runs the risk of being unclear and can potentially result in additional overhead interacting with the server side, too (cf. decomposing the whole URI followed by information extraction and processing). On the other hand, if a client's query should be completely independent (i.e. not rely on a client state), the whole information has to be passed to the server this way to remain RESTful.

4.2.2 Support of selected features only

It is, of course, discussible to which extent the interface should cover the functions of the core application. Is it necessary to completely support everything, or is a small number of the most commonly used tasks enough to offer? Cogitating this, one has to keep potential use cases as well as logical deductions in mind. It often can be a quite extensive task to determine different

[5] In fact this would be an infix phrase because the server identification (or similar) is the very first part of any URI, but the denotation here refers to the actual context being accessed for the sake of simplicity

needs of future users, but the field of application can be narrowed down without any difficulties here. Disregarding special cases, the fact that Backstage is a web application, which can also be accessed directly by its own user interface, shortens the focused user group mainly to a couple of people using the application on, mostly portable, terminal equipment. A closer look to the two big user groups of Backstage, lecturers and listeners, gives further clues. Obviously, the number of listeners is a lot bigger than the number of lecturers, which leads to the conclusion that the support of this major group is preferential. If one further analyzes the interaction and common tasks of a lecturer, which basically are composed of lecture administration while lecturing and a few setups, preparations and maybe analysis otherwise, one could conclude that operating the system without a laptop, but using a mobile terminal won't take place very often. Thus, the majority of the lecturer functionalities can be treated more subordinate and does not need to be covered by a REST interface. To force the use and prevalence of the application, it is surely no drawback to offer more than one way to the listeners to interact. Regarding the use of backstage during a lecture, there is only a manageable amount of interactions available to the listeners, too. These facts call for a complete implementation of listener oriented features, which means at least the lecture accompanied ones. For the sake of completeness, it has to be pointed out, that of course not every implemented URI can be called in combination with every request method conforming to the "uniform set of actions"- constraint interface, simply because not every semantic result is desired or actually exists.

4.2.3 Logical coupling and separation

With these preliminary considerations in mind, the REST interface has been designed the following way: There are three subcategories, each of them corresponding to one of the earlier identified semantic systems: feedback, message and admin (cf. Chapter 2) and one small subsection called info providing additional information on core constructs such as users and lectures. As a fifth part the quiz model has been excluded from the feedback system and has been made accessible by its own identifier, to the following reason: Feedback identifiers summarize methods to initiate either self regulatory mechanisms of the system (e.g. exclusion/marking of spam posts) or sentiments and questions on the lecture directed towards the lecturer. They are available at any time and always initiated by the audience. On the other hand, the quizzes are an optional feature, initiated by the respective lecturer and offer also a possibility to monitor a listener's comprehension. As this difference is also present in the technical realization of Backstage, this decision is reasonable.
The exact opposite of following the technical implementation of the application is used adapting messaging and annotation constituents. The messaging system on its part builds the base for the annotation system in a certain way: Whenever some kind of annotation is sent by a user, from a technical point of view only a special kind of tagged message is transmitted using standard messaging methods (cf. Chapter 2). The decision not to subsume these two features, even if they are quite coupled in a technical way, was finally based upon getting a clean separation of the available functionalities.

4.3 Concerns on the API

As a final result, each URI subcategory is described in a brief summing-up of its abilities with regard to additional background information affecting the respective look.
Figure 2 provides a general schematic overview of the URI composition. A detailed graphical breakdown and the corresponding API of each subcategory presented in the following section is provided in Appendix I.

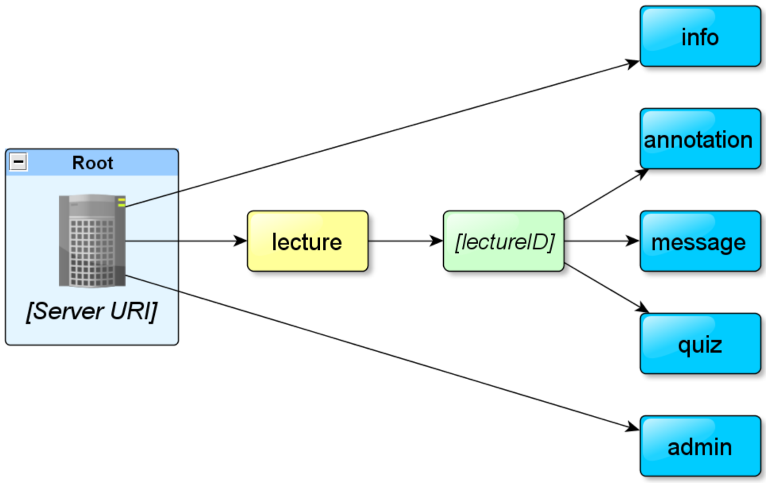


Figure 2: Schematic overview of Backstage URI composition

4.3.1 Admin

Basically, the admin-tagged URIs represent the possibilities of Backstage's initial start screen. Simple administrative options like adding, deleting and getting information (which match the HTTP request parameters pretty well) are provided and aimed to manage one's own set of lectures. The only method restricted to a lecturer's use is the creation of a new lecture. At this point, it has to be remarked, that every query changing the constitution of a resource (i.e. every URI call prefixed with PUT, POST or DELETE parameter), which is not further detailed in this section, carries its information in the body of the respective HTTP request. The determination on the arrangement of all of these methods can be extracted from the examples given in the API documentation (cf. Appendix I).
The two methods regarding the URI "admin/lecture/*[someLectureID]*" deserve special attention. Even though they are addressing a well defined resource (i.e. an existing lecture) and, as per this aspect, are REST compliant, their effect, which is to start a new lecture session, smacks more of a remote procedure call.
Nonetheless, this divergence is reasonable. Backstage is handling all of its lectures in different sessions, one assigned to each of them while "running", or more precisely after being activated by a lecturer. A use beyond this regulatory part, even if this would be the most RESTful method, would cause an additional refactoring of the existing implementation to the point of completely bypassing and therefore doubling it and is consequently not practicable. However, another optional possibility, which is in fact using sessions but masking this concept for the users, should be sketched here: It would be possible to automatically start a new session every time the number of users for a certain lecture changes from zero to one and likewise stop the session if the last user logs of or gets dropped due to a timeout. Unfortunately, this procedure would either take away the

control of the own lecture from the lecture, or, if restricted to the lecturer, who created the lecture, always demand for his incessant presence. Thinking of a sluggish solution like that, there is no real need to change the existing one, which offers an option solely in black and white terms:

Either relinquish the support of initiating and terminating a lecture via REST, or deviate from the strict REST demands. In respect of the lecture administration as the primarily part to initiate the application's functionality, the focus was rather on completeness than compliance, which means, that in this section of the API no omissions of the core application's actions were performed and an RPC-similar style has been deployed.

4.3.2 Message

The message related functions include every possible way to communicate in the application excluding communication that is intended to give feedback. Accordingly, URIs for receiving posted messages and on the other hand posting them are provided. Special attributes, like sending private messages or performing any posting anonymously, were also taken into consideration. The realization of these options includes some noteworthy alternatives.

To be able to follow the deliberation, again a look at the technical way Backstage is handling different types of messages is advisable. In fact, there is no difference in mapping the diverse messages to the operating program logic. They are calling one and the same service method, which is able to distinguish the type of message by parsing directives included in the message. Thus, a user has to insert a known tag (e.g. "@>" followed by a username to send a private message) occurring anywhere in the text to instruct the method properly.

Using Backstage via web interface, this procedure gets automated by offering checkboxes (for anonymous posting) and click options (for private posting) to select the type of message, which inserts the proper tag on its own. Keeping this in mind, again two potential REST transformations result:

First, it is conceivable to assign only one URI handling the messaging and so to stay implementation-based, which then calls for the passing of tag information in the body of the query. This option needs to be documented very well in the API and could easily cause some problems if being put in a user's hand. For example, if a user decides to send private message, which contains personal information and is not intended to be published, and commits a typing error by mistake affecting the privacy tag, a private concern could not be identified and the result, a post visible to the public, would be quite deficient.

Accordingly, the alternative, as a more secure possibility to avoid this kind of error, would be to adapt the user interface's approach and insert the more cryptic special tags automatically. Here, the server side communication part of the REST interface has an effect by handling and processing the query. It can easily transform the message sent into an application's logic compliant text, which gets prefixed correctly without the user's help.

Though there is another option to disburden the end user completely by holding the developer of the client application responsible for an appropriate realization, a more abstract solution supplied by the server-side can increase simplicity while masking internal structures at the same time. Nevertheless, the uncommented existence of a prefixed solution as a "shortcut" for experienced users or a client-side developer can help to increase the usability [Shneiderman, et al. 04].

It is further questionable, how to pass the desired messaging mode. Being totally RESTful, it seems the best to assign an independent URI to every combination imaginable. The possibility to send public as well as private messages anonymously would define the number of messaging URIs to four, a private and a public messaging mode, each of them optionally published anonymously.

Taking the assumption that anonymous and private posting is rarely used in comparison to standard messaging anyway, another concern would be to outsource these options to additional HTTP header information, which would actually help not to overburden the API.

Additionally, the total omission of these features would be conceivable, too (it has to be remarked

at this point, that the omission of anonymity would not increase the safety regarding spam avoidance as one might think, because every user sending anonymously is only unrecognizable to other users, but never to the application itself).
The solution applied tries to keep both messaging modes alive in order to feature the complete range of options while being RESTful to the greatest possible extent. Thus, the omission and the HTTP header options (to maintain a clear uniform interface) are discarded.
To prevent indefinitely long URIs, the different modes as well as the intended receivers are not integrated in the URIs, but made a part of the request's body alongside the text to be posted (cf. Appendix I). As a result, only one specification is needed, which keeps the API simple, and avoids to influence the HTTP request methods by enforcing options leading to ambiguous interpretations that would partially insult the uniform interface constraint.
To check on availability of private and anonymous messages, getter-like URIs are present querying the status. Likewise, presenter-restricted setters are prepared to enable these special messaging features.
In general, a URI used in combination with POST as request method is always intended to update a resource. This way, the HTTP body should contain new data, but as this resource can only be updated to one other context (comparable to a light switch that can either be set on or off) the body is empty here, even though it is used a bit sloppy in terms of REST. One might also argue that the use of setters eliminates the stateless approach of REST at this point, but the permission to post anonymous or private is not a client side state. While this type of state information should be avoided, the server is still allowed to retain state (cf. Chapter 1.4). In fact, the request is completed after its sending, without the need of further information retrieval between different queries and this way compliant.
Another useful functionality is the applied pagination. Whenever a set of posts is requested, only a predestinated amount is returned to the user unless the label "*all*" is explicitly stated. Portable terminals, which are often limited in their abilities to display huge amounts of information at once may profit from this feature notably. Traversing different pages is provided by equipping each page being returned with hyperlinks to adjacent postings. These state changing hypermedia references perfectly comply with the postulations of REST (cf. Chapter 1). Deliberations on setter options for a predefined number of posts per page have been discarded, because such a user individual-state stored on server side definitely has an impact on the statelessness. As the result the design decision to use a fixed number of ten posts per page, which seemed to be an adequate amount at all events, has been applied.
The last feature in this category is an optional method of post listing. It is designed to filter posts matching different criteria, for example, to show the top-ten rated posts. While "GET [...]/message/category" returns a list of applicable filters or rather categories, the insertion of a concrete category into the request returns a corresponding list of posts matching the specified filter. Hereby, the first named result of the answer, placed on top, is the most relevant one in the demanded category. Up to now, the filtering feature is not yet present in the prototype, but it emerged from a brainstorming where it has been concerned to be a useful option for analyzing purpose.

4.3.3 Info

One of the smallest subcategories concerning its scope of operation and complexity is the info section. These URIs are expediting the hypermedia contents of the REST interface by offering additional information about certain models. For example, properties of a specific lecture, like the amount of active users or the possibility of posting anonymously, can be obtained. This type of basic information module can be referenced at any occurrence of the specified model within other responses. A prospective extension promoting the density of linkage between the representations is desirable and recommended.

4.3.4 Annotation

By the way of comparison, the annotation section is a simple one-to-one transfer of the already mentioned properties. Post- as well as lecture-related annotations can be emitted, but only annotations to a post can be uncategorized. A few getter URIs help to retrieve annotation categories for posts and valuation types for the lecture. While the rating of a single post can be queried easily, a lecturer cannot retrieve lecture feedback this way, since Backstage internally handles these valuations as messages. For this purpose the "*GET /message*" task is assigned. Options to modify valuation types are not supported by the REST interface and only part of the main application's extent.

4.3.5 Quiz

The elements of the last category are also converted in a straightforward manner. From the creation and initiation of a quiz to its cessation, the whole properties of a its life circle are made available to a lecturer. The listeners on their part are able to query an ongoing survey and also to submit an answer accordingly. Though the look and type of the answers can diversify exceedingly, at least a uniform answering scheme (cf. Appendix I) has to be retained to facilitate its processing on server-side. Fortunately, the surveys are composed of multiple choice questions and do not contain free text forms, which allows to work with indexing, for example. The profit on server-side, concerning the costs of parsing and interpreting free text forms in comparison to process indices, is evident.

5 Security Concerns

A factor that has been completely neglected so far is the security of the application's RESTful enhancement. What is the profit of the most useful add-on, if it is affected by inherent security holes? Especially applications that are offering ubiquitous access for varied devices have to arrange precautionary measures due to their much bigger attack-surface. The following section will analyze different threats to the developed REST interface and describe various prevention attempts and techniques.

5.1 Identification of potential threats

A vantage point for evaluating the endangerment of a system's abuse is to put oneself in the position of an attacker or a person trying somehow to cause damage to the system. In doing so, the following points are listing some perils, possibly impending:

- Spamming the system
  (arisen from the misuse of the messaging functionalities to disturb the proper operating of the system)
- Enforcing a denial of service attack
- Intentional adulteration of annotations or quizzes to provoke distortions of evaluation
- Intentional downgrading of useful posts
- Spoofing the identity of another user to discredit him by submitting improper content
- Accessing methods, which are restricted to another type of user
  (e.g. stopping a lecture session though only holding normal user rights)

Having identified these threats and following the proceeding of security engineering, a next step would be to ascribe a probability of occurrence to each of these threats and further to perform a corresponding risk prioritization by also balancing the prospective costs for each event. According to this ranking, the required countermeasures can be initiated [ITU-T 91]. Due to the prototypic character of the application and the absence of acute danger of being a primary target, this approach seems to be too exaggerated concerning Backstage. Rather, a basic security substructure, which is able to prevent, or at least impede, most of the specified concerns, should be created.

5.2 Counteracting by limitation

The implication to curb most of the perils is limitation. If, for example, the number of posts is limited, the danger of suffering from Spam drops. Also, if an attacker is not able to palm off massive manipulating feedback or vote discretionary, the effect of single, malicious outliers can be neutralized by a robust rating algorithm present in the main application. A big issue in this context is, how to impute the limitations to the users? If one allows only a regulated number of REST queries at the same time, on one hand a system overload could indeed be averted, but on the other it can still be the same source inflicting corrupt data. The solution for this purpose is a simple adaption of the main application: Accounting.

This way, every action performed can be assigned to a user and is therefore selectively limitable. Of course, the spoofing of another user's identity to circumvent certain user-related constraints is still possible, but an elementary protection, as desired, can be ensured at least. Consequently, the user on his part has to authenticate himself. As the authentication in REST context is not a standardized element, the following section will present the most commonly used integrations and give reasons for an eventual selection.

5.3 Modes of authentication

Like the selection of a lecture, also a RESTful authentication process has to differ from the one that is part of the web application's workflow. The authentication, using the web interface, is performed and saved initially by a simple login mechanism. The reason to not adopt this principle is the same as in the aforementioned lecture context, which is to keep statelessness at any time. Yet again, the implication is to include the authentication data in the request, but fortunately, there are alternatives to enlarge the URIs in this case. The subsequent paragraphs exemplarily illustrate three completely different possibilities to employ authentication for the RESTful Backstage interface.

5.3.1 HTTP Basic Auth

Potentially the plainest option to operate with is HTTP Basic Auth. The Hyper Text Transfer Protocol offers a special header field, named "Authorization" for entering an encrypted user / password combination. This tuple is stored concatenated and ciphered in the designated HTTP header segment. As encryption scheme Base64, which is based upon the transformation of the plain text to ASCII code, is applied. A resulting bit pattern of this ASCII representation gets split into eight bit chunks and retransformed into character format to obtain the final cipher. By using eight bit, the cipher consists of a total number of 64 different characters at the maximum. This dictionary is composed of character entries that can be interpreted by all participants of the communication. Thus, it is ensured that also complex alphanumeric combinations can be transported via simple transit systems without defects. After transmission the cipher is disassembled in reverse order and the access to the desired resource is granted, if the given data can be verified [Franks, et al. 99].

5.3.2 O-Auth

In actual fact the operating mode of O-Auth is much more based on the principle of authorization, which is the assignment of access rights, than authentication. The most common specification is the "three-legged" O-Auth variant. This is a three-component model including the following parts [6]:

1. The “user”, who wants to share private data with another application or system,
2. The “service provider”, which is the system the private data is currently located on,
3. The “costumer”, which is a third-party application or system intended to get access to the private data

To achieve this, the user, working on customer site, gets redirected to a login interface of the service provider, if he wants to grant the customer to access data from the service provider. After having authenticated to the service provider, a request for permission - which allows the customer to work with the data being stored on the service provider - is presented to the user. At this point he explicitly has to allow the sharing. By doing so, the customer gets legitimated. The entire mechanism, from a technical perspective, is based on the use of two types of tokens, a Request Token and an Access Token. The Request Token is submitted by the costumer and gets authorized by the service provider, as soon as the user authenticates. Its only purpose is to obtain a user's approval. Afterwards the authorized Request Token is replaced by the Access Token, which is handling the access of the protected data [Hammer-Lahav 10].
A huge advantage of this procedure is that a third party (in this case the customer) can be authorized without ever coming into contact with security relevant things, like a password. Though being simplified, an example from everyday life - the payment by cash card in a store - can illustrate the issue once more: The store (as a customer) needs to get access to your private goods,

[6] The O-Auth corresponding terminology for each component is quoted

which is your money to pay the bill. By using your cash card and your pin (login mechanism), you are able to authenticate your bank (the service provider), which holds your money, to authenticate the store. After a successful authentication, the store can access and transfer the correct amount of money, but it never got knowledge of your private pin.
Besides the "three-legged" type, there is another model, called "two-legged" O-Auth (also known as "signed fetch" or "phone home"). This far less used model is also intended to control the access of data between two systems (the customer as requesting part and the service provider as offering part), but without the presence of an explicit accreditation by a user. Accordingly, there is only an Access Token, but no Request Token. An example of use would be the communication between two applications developed by the same software engineer. One of them, for example a mobile phone gadget, could retrieve data from the other, for example a content holding database, to get detailed information on the user. Although this variant is being used quite rarely, it is often associated with REST. But in most cases the explicit definition that these requests are initiated by one of the systems itself and not by a user is missing. On top of that, the two parts involved already have to trust each other [Dewanto 09].

5.3.3 API Key

The authentication by API Key is quite similar to HTTP Basic Auth. There is also a shared secret provided by the server side. However, in contrast to HTTP Basic Auth a standardized way of assembling the key does not exist. The service administrating this way of authentication itself has to take care of this procedure. Furthermore, the key is not transmitted as a part of the Hypertext Transfer Protocol, but in general integrated within the requesting identifier. Though being only an alternative to an already present scheme, the idea of using an API Key is widespread among existing web services (cf. Google Maps [Google, Inc. 11], Flickr [Yahoo!, Inc. 11], WordPress [WordPress.com 11] etc.). Presumably just the larger design flexibility might be one of the reasons for this.
In combination with a different authorization practice, the API Key method can still reveal some new aspects worth mentioning:
Up to now, the presented authentication methods are similar in their scope of authorization, which especially means their management of access rights. The rules, whether or not to access security relevant data, in the above examples are completely based on the successful recognition of a legitimate right holder. This process (i.e. the authentication) is able to allocate certain rights to each user identity unequivocally and is called "Discretionary Access Control" (abbr. DAC). However, there are some other types of rights management, which are not referring to a single instance but a classified group. One of them is the Role Based Access Control (abbr. RBAC). It subdivides the users into different groups, each of them representing a role. In most cases, this role describes the relevance of a user for the system and can be applied to every member of the group. Only depending on this belonging, the rights are attributed. Conversely, this attribution implicates the authentication of a group member only. The definite identity, which member of a group is accountable for a performed action, is not ascertainable in this case [Ferraiolo, et al. 92].
With the RBAC model in mind, the API Key can be designed to contain simple and direct access rights, even if the user is in possession of multiple roles (A concrete example in the context of Backstage is the attendance of a single user at various lectures). Therefore, the server side logic has to provide keys that are able to represent every combination of existing roles. The most efficient implementation seems to be a combined key, consisting of the appropriate role-permissions. By disassembling the key, if a new query reaches the server, the permission is directly identifiable. In contrast to the individual user authentication of HTTP Basic Auth, there is no need to query the database to obtain and monitor the access rights. Despite the fact that the authentication procedure is not necessarily part of the REST specification, such an API-Key design keeps up the idea of using independent, self-contained URIs, which again accords with intentions of REST.

Nevertheless, this method entails some dangers. Once the secret composition of the API-Key can be deciphered, an attacker is able to gain any access right without being detectable, because of the not user-, but role- related authentication. The subsequent effort to revise this problem is also much more expensive, given that the security modalities for a whole group have to be changed.

5.4 Suitability criteria

To finally determine which authentication scheme suits the present demands best, a look at the intended use is helpful. Backstage as prototyped version is currently in an early stage of development, which rather requires a simple and functional than an exaggerative and complex solution. It also affects the costs of implementation. Keeping that in mind, the setup of a completely new structure, like the environment of OAuth, is not preferable. The future use of this concept in a final version of Backstage is thoroughly conceivable and expedient, the more so, as most of the famous REST APIs (cf. Twitter and Google) are using OAuth nowadays [Twitter, Inc. 11]. It should be pointed out here that the two-legged variant of OAuth most likely has to be discarded, because the majority of queries will be user initiated. While the devised role-based API-key method would also require the presence of an administrative environment to manage the API Keys, the simple API-key variant and the HTTP Basic Auth both could revert to the authentication parameters of the present core application. With the access to the database entries of Backstage, the only requirement of the REST authentication unit would be to decipher the encrypted login credentials and to deny or grant the access depending on the comparison with the information stored in the database. The modified API-key scheme is probably suitable for a use in the final version, but keeping the potential security concerns and OAuth, as a capable alternative, in mind, it finally applies more to web services with either low-level security, or a huge affinity to an exaggerative REST compliance. Concluding, the almost congruent types, HTTP Basic Auth and the simple API-Key solution, remain. Due to the already very substantial design of the URIs (cf. Section 4.2.1), it seems useful to avoid additive extensions to keep the identifiers perspicuous, which makes HTTP Basic auth the most favorable option.

5.5 Conclusion and outlook

By using HTTP basic Auth, at least a minimum level of security can be assured at moderate implementation costs. This solution appears sufficient for a prototyped version of Backstage, but should be reassessed and enlarged for a final release. Because a lot of unanticipated security leaks arise in operating state (i.e. real-time conditions), the technical expertise, gained from prototype phase, is another important component that can be assessed and incorporated later on. It is also possible, that these findings disprove some parts of the decisions made during development process and cause sizable changes. Unlike the explanations illustrated in this chapter so far, there are many diverse security concepts that are not taking place on the application layer of the OSI (Open Systems Interconnection) reference model. Sophisticated techniques belonging to this category (e.g. "Secure Sockets Layer" [abbr. SSL] or "HyperText Transfer Protocol Secure" [abbr. HTTPS]) can support the application security and should be considered besides, though not being further detailed here. But nevertheless it has to be reminded that additional security concepts are always a potential threat to loose statelessness and with it RESTful compliance [Rocher, et al. 09].

6 Implementation

After a basic understanding of the required conceptional and technical principles has been created, the following abstract will illustrate the concrete implementation. Since evidently not every line of source code can be scrutinized, only the most noteworthy characteristics and intricacies are being reviewed. In doing so, the actual extent of the implementation is outlined first, followed by a workflow oriented detailing of the realization.

6.1 Scope of implementation

First and foremost, it hast to be pointed out that the API has not been implemented to its full extent - as it is noted in Appendix I. There are a few aspects to reason this decision: The intention of a project thesis is rather to elaborate new considerations on the presented topic than repeatedly perform similar tasks. Additionally regarding the limited time frame, a target-oriented approach that abandons redundant features is preferable. In more specific terms, only one of the five main subcategories - the message functionality - has been implemented[7]. Nevertheless, every step to set a proper fundament has already been taken, which means in particular the setting of a basic scaffold and the creation of a commented dummy method. Until now, this method contains a description of further tasks to complete and is the only point where the corresponding logic has to be inserted. Every detail to route the workflow (cf. URL mappings and security filter) is present and even applies to currently incomplete methods, which expedites a future process of completion evidently. In regard to the following testing, one might think that a complete lineup of methods is necessary to be able to investigate the performance properly, but in fact the different messaging functionalities are totally sufficient to get an approximate performance rating, because the potentially most frequently used components - message and feedback related functionalities - are all based on almost the same methods of the core application's logic. All of the other features are expected to be invoked significantly less and therefore do not seem to considerably alter the testing results.

6.2 URL mapping

One of the most challenging tasks when realizing the API is, of course, to get a clean internal separation of the complex URI structure. Clarity as well as the unproblematic access to further modifications depend on this arrangement. Fortunately, Grails provides a predefined configuration file called "URL Mappings", which works as a simple entry point for the programmer. Within, it is not only possible to define the valid combination of acceptable queries, but also to redirect an arbitrary query to a specific method of a controller that is called "action" in this context. Another resourceful possibility is to define global parameters within the constraint definitions, which allow to map the queries dynamically. A system-wide availability of these elements is assured by Grails, which is very useful regarding the further processing of the queries in the respective controller. Listing 8 shows a detailed example of a dynamic mapping. The third line illustrates the regular expression-like pattern used to diversify the requests. Parts prefixed with a dollar sign define the parameters. These parameters can be accessed as shown in line six. The post-fixed question mark determines the optional character of a parameter and the asterisks allow an arbitrary depth of the "specification" variable. As one can see, every request matching this pattern is forwarded to the controller named "rest". At this, the action is determined dynamically by returning the parameter "category". The respective method of the controller has to be named the same way of course, to map the request properly. Though non existing combinations automatically result in the return of an error code, the definition of constraints, as shown in line ten, is desirable to ensure non

[7]Since filtering messages is not yet available in Backstage (cf. Chapter 4.3.2), the corresponding methods detailed in Appendix I were not implemented

erroneous behavior. In consideration of REST, Grails offers a special feature to subclassify the invoked URL regarding the appendant request method. This scheme can additionally be used defining the respective action, but it has been designed to only feature the most commonly used HTTP request GET, PUT, POST and DELETE. Although one single point to define the mapping would be preferable, at the present case a two step solution had to be applied. Because of the fact that Grails currently does not support the combination of dynamically diversifying both, URI parameters and request methods, at the same time using the URL mapping, it is only carrying out the proper disassembling and forwarding of the URI. The request method is being distinguished by a switch statement within the respective action. Another possible solution would have been to completely abandon a dynamic approach and resolve the URIs and request methods statically, but if one thinks of the multitude of possible combinations this seems to be inapplicable.

```
static mappings = {

  "/rest/lecture/$lectureID/$category/$subsection?/$specification**?"{
    controller = 'rest'
    action = {
        return params.category
    }

    constraints {
      category inList:['message','annotation','quiz']
    }
    [...]
}
```

Listing 8: Exemplary dynamic URL Mapping

6.3 Authentication by filtering

The concept of filters in Grails is a very useful option to outsource perseverative administrative tasks that are not coupled with a specific method. This might be the counting of performed tasks, certain filter operations - as the name already implies - or the procedure of authentication. The basic working principle of a filter is to interrupt the workflow of the application at certain points. In contrast to the concept of an event listener, which is able to interrupt the workflow arbitrarily if a specific event occurs, a filter can only be executed on three fix occasions: 1) before the filtered action is performed, 2) after the action has been performed, 3) after the rendering of the view (cf. Listing 9). Consequentially, the authentication has to take place before the action gets executed. The scope of the filter can be defined separately in the head of each filter definition. As all of the RESTful methods should be restricted to registered users, every action of the REST Controller is filtered (cf. Listing 9). If an error occurs within the “before” filter, for example caused by an invalid authentication, the return of a boolean “false” statement is sufficient to prevent the execution of the filtered action. After detailing the general functionality of a filter, the concrete steps of the authentication can be sketched:

1. The readout of the authentication data from the specific field in the HTTP header, called “Authorization”
2. The decryption of the Base64 enciphered credentials
3. The comparison of the credentials with the deposited database entries of the user

If the submitted credentials accord, the user can be marked as authenticated and the filter can hand over the control to the designated action.

```
def filters = {
    basicAuth(controller:'rest', action:'*') {
        before = {
          [...]
        }

        after = {
          [...]
        }

        afterView = {
          [...]
        }
    }
}
```

Listing 9: Simple filter definition scaffold performing basic authentication

6.4 Session adaption

As already indicated earlier (cf. Chapter 4.1.2), the interplay between the statelessness of the client side and the wrapping of information to use already existing server-side functionalities emerged as a tricky task. Given that no information coming from the client should be stored longer than for the time a single query takes to complete, again Grails' filters appear as a practical solution to scaffold the logical components. Every time a query arrives and the previously described security mechanism succeeds, the necessary information for the server to operate properly is extracted from the present query and further accumulated in a dummy HTTP session.

Since in normal operating state of the web application this information can be obtained from the current web session, which unfortunately is not present using REST, this step appears indispensable trying to reuse the original source code. In general, the insertion of the current user name and the targeted lecture identification are sufficient. To avoid persistence of this data but still provide the information throughout the entire query, the information is registered right after the authentication within the "before" filter and accordingly invalidated after the processing within the "afterView" filter. This way, the demanded availability constraint can be ensured and a query can be completed via REST using the standard source code of the web application. Accordingly, the earlier developed idea of handling the session adaption within the REST controller had to be dropped due to the perfectly matching automatic execution of filters before and after a controller's execution.

Unfortunately most of the offered tasks entail aftermath that does not affect the proper execution of the task, but still influences the further application flow severely. The depiction of a standard task, the posting of a simple message via REST, can help to exemplify this problem:

Once a proper query has been sent from the client, the application flow traverses all of the already illustrated features up to the use of the originally web application oriented source code. At this point, the application logic is not only responsible for an orderly processing of the current query but it is also carrying dynamic tasks out. These, basically DWR-related, tasks ensure the on-the-fly spreading of recently incoming information to keep a consistent and up-to-date state for all of the respective users, which means in this case to display the transmitted message right after it has been processed. Otherwise, if this task fails, an intended receiver using the web application would have to initiate a completely new database query and therefore would have to reconnect to the system to get the latest updates, which is surely not desirable. Accordingly, this feature has to be kept to assure a useful interaction between REST and web application users.

To comprehend why this mechanism is not working for the adapted REST variant one has to trace

the DWR-related data flow. In doing so, it turns out that DWR is using so called "script sessions" to keep track of the participating users (note: not to be mistaken with the adapted HTTP session discussed earlier). Every time a user is following a certain lecture via web interface a script session derived from the current DWR "web context" class is registered. The web context is an administrative class containing the current path context of the web application. If one wants to get all of the currently active listeners of a lecture, for example to notify them that a new message has been posted, one has to retrieve all of the registered script sessions related to the actual lecture. DWR is offering various methods to retrieve these sessions, but unfortunately they have to be invoked on one's own valid DWR web context. Since a REST user has a different login process, a DWR web context is not present. Accordingly, the method cannot be executed because the web context is a "null" object in this case. Thus, the necessary script sessions cannot be determined.
Being aware of the failure's source, the solution to also adapt the DWR session (like the HTTP session within the filter) suggests itself, but the rather cryptic assembly of this class entirely eliminates this option. Given that this problem could turn out to be a knock-out criterion for the use of the whole REST interface, still a working solution has to be applied (cf. Listing 10)
The trick to achieve this is an escalation of the DWR web context variable. Every time this variable is used to invoke script session querying a conditional statement gets applied first. If the variable is null, which means the operating user is connected via REST and the request would normally fail, the correspondent method is called using by a class called "servletContext". This class is located one step above the logical level of the web context and is also present interaction with the application by REST. However, this first modification only allows for the correct invocation of the method, but the parameter value still cannot be determined dynamically because the "getCurrentPage()" method used normally is not available for the "servletContext" class. Fortunately, this variable can be filled in statically because it always equals the same (cf. "/Backstage/dashboard-/index" in Listing 10).

```
if (webContext == null){
    sessionCollection = serverContext.getScriptSessionsByPage("/Backstage/
        dashboard/index")
  }
else{
    sessionCollection = webContext.getScriptSessionsByPage(webContext.
        getCurrentPage())
  }
[...]
```

Listing 10: DWR modification to enable dynamic updates of web application users

Although this modification seems to be an unorthodox kludge, on one hand it serves its purpose and on the other hand it is apparently by far the most modest solution regarding the extent of changes.

6.5 Content negotiation

Besides the actual depiction of necessarily implemented problem solving, the last example will illustrate the realization of an optional modification. In more specific terms, this means the adding of an alternative representation concerning a requested resource. In addition to the widely spread XML standard the application should be able to deliver the, in web context also very common, JavaScript Object Notation (abbr. JSON). Disregarding its publicity, the decision to use JSON is mainly based on two reasons. On the one hand JSON is a well-established pendant to XML whenever technical capabilities are limited, which might be the case using REST. On the other hand the Grails framework natively offers builders for XML as well as JSON that enable a user to produce output in the corresponding format by using very simplified syntax. Thus, a complex assembling

should be avoided.
Given that a real content negotiation between client and server can be a laborious task concerning the administrative overhead, a client initiated solution seems to be far more efficient. Whenever a client wants to get a non standard representation, he has to specify this within his request. If the selected format is not supported by the server, he automatically receives a standard representation, otherwise his specification is enforced. From a technical point of view, the specification is either transmitted using the desired file extension that is postfixing the URI (e.g. by ".json" to get a JSON representation) or via a special "accept" request-header field of the HTTP. In the present case the "file extension"-option has been chosen because it seemed to be a more apparent and uniform approach than a non-transparent HTTP header field.
To obtain information on the desired format, Grails is offering a predefined "request" object. It is an instance of the "HTTPServletRequest" class that contains any information related to the respective HTTP request. Accordingly, the "request.format" statement provides the current file extension (note: the above-mentioned separation of HTTP request methods can be determined the same way, using the "request.method" statement). Hence, if both, the XML- and the JSON-builder syntax, would be the same, a simple conditional statement defining the render method would be sufficient to produce adequate output. Regrettably, it turned out that the JSON builder is natively not able to support nested structures. Additionally, the limitation that the JSON builder apparently does not process elements named the same as any elements (i.e. methods or attributes) of a superordinate class, makes the intention quite intricate. As a result, only one simple method has been implemented to provide the proof of concept and also a reference point for further implementations relating to this topic.

7 Load and performance testing

With the development of the RESTful interface not only a capable enhancement of the application but also an optimal basis for performing various testing has been created. For the first time, the developers are able to investigate the overall performance of the application concerning desired load scenarios. Due to the GUI-related workflow of Backstage as core web application, without REST, only non-automated testing - like user studies - could provide comparable results. Though, there is a certain risk performing user studies by having no earthly idea what the limitations and capabilities of a system are. From the lack of gradually approaching the load limit to the difficulties of an ad-hoc fixing of appearing bugs and the deficient uniformity of user behavior over time, a user study does not seem to be a suitable primary approach, in particular with regard to load testing. By mapping the complex functionalities of the main application to plain and simple HTTP requests, REST allows to utilize various well-established testing tools providing a perfect environment for fully automated testing.
This example of use shows a whole new capability of REST and emphasizes its importance concerning the development of Backstage from another point of view. The following chapter documents the whole testing procedure, starting with initial deliberation on the intended purpose and a short introduction of the applied soft- and hardware. Thereupon the different tests with their respective settings and related results are presented and summarized under various stages of development. The final conclusion gives an interpretation of the obtained results and offers advice to further improve the overall performance of Backstage.

7.1 Purpose of testing

Since prior to this testing absolutely no information on the overall performance of the system was present, first and foremost reliable information on the minimum number of users that Backstage is able to support in a nonrestrictive way (cf. Section 7.3) had to be gained. Depending on this factor the need and extent of further arrangements could be assessed. If, for example, the number of supported, concurrently operating users exceeds a few thousand without any improvements - which essentially cannot be assumed - further adjustments would involve more inefficient hardware to determine the minimal needs to supply a sufficient level. On the other hand, if a more likely result confirms very few users, it would be beneficial to identify and document existing bottlenecks - whether on hard- or software side - and give potential solutions to avoid them. Initially, the trouble-free support of a lecture's average listenership (i.e. approximately fifty to two hundred people) without any limitations would be desirable.

7.2 Test environment

To eliminate as many interference factors as possible and consequently getting clean and significant results, a fixed testing environment with a recoverable initial condition is indispensable. The subsequent hard- and software components built a constant composite during the whole testing. Every test run described in the course of this chapter based on these settings and calibrations unless stated otherwise in the respective scenario.

7.2.1 Hardware components and setup

Initially, the used hardware components were not designed to be high performance servers that were highly stressable simply due to the absence in the everyday situation of a lecture. An up-to-date desktop PC with the proper hardware lineup was considered as the most probable setting for the prospective operation purpose. This setup also facilitated the whole testing due to the more ordinary availability and handling of standard components compared to the one of high performance machines. Further, if the components showed incapability in most instances, a later escalation to a more powerful setup still would have been possible. Hence, the following hardware setup was applied:

- CPU:
  Intel Pentium 64-bit Dual-core processor with 3 GHz (approx. 1.9 GHz per core)
- Memory:
  8 GB DDR-3 RAM
- Operating system:
  32-bit kernel Linux (using physical address extension technology to make full use of the memory)

To achieve a separation of concerns on the hardware level but keep potential sources of error constant, three absolutely identical PCs were used. One working as database server, one containing the application and the third one representing the query server that contained the load testing tools. Otherwise, if, for example, database and application were set up on the same machine, additional monitoring tools and server jobs could exceed the available total power of the system and accordingly cause adulterated results.
The interconnection between the PCs has been established within the regular Ethernet network of the Ludwig-Maximilians Universität. This network is administrated by the Leibnitz Rechenzentrum (abbr. “LRZ”) and provides an infrastructure capacious enough to support the upcoming traffic. A completely local ethernet configuration has been abandoned due to a potentially erroneous extenuation of the results.

7.2.2 Software components and setup

The main attention concerning the used software was paid to get by on non proprietary solutions. A comparatively small test designed to just get first impressions on the general behavior of the system did not seem to legitimate the costs of proprietary solutions. Fortunately, a couple of powerful and comprehensive alternatives - introduced separately in the upcoming section - are freely available to satisfy the requirements.

1. Application Server - “Glassfish v3”
   Though not directly comparable with monitoring or testing software, a server-side program hosting the application was the key requirement setting up the test environment. Regarding the huge number of potential, freely available application servers, the objective to find a solution causing as less administrative overhead as possible has been pursued preferentially. Offering a very quick and easy setup and application deployment, the “Glassfish v3.1” software has been chosen for this purpose. It is an established, Java-based application server published under GNU General Public License that gets constantly refined by the developer team of Sun Microsystems since 2005. Staying quite simple in first instance, there are still enough advanced on-demand settings present to counteract contingent bottlenecks (e.g. the need to allocate more memory) that the application may cause during the testing.

2. Database monitoring - “mongostat”
   To monitor the currently not very widespread MongoDB, the rudimentary console output of the “mongostat” utility, which is already included in the database, proved to be totally sufficient. Figure 3 shows an exemplary console output (note: for lack of space the graph, which is in fact showing four associated lines of console output, has been split into subgraphs a and b). After being started mongostat outputs one new line every second containing the refreshed specifics of the currently running mondodb instance. In particular, the following parameters provided valuable information. Any other data has been regarded secondary and accordingly was not further used in the process of evaluation:

- insert - number of inserts per second
- query - number of queries per second
- update - number of updates per second
- delete - number of deletes per second
- flushes - number of fsync flushes per second
- faults - number of pages faults/sec (linux only)
- locked - percent of time in global write lock
- qr | qw - queue lengths for clients waiting (read|write)
- ar | aw - active clients (read|write)
- netIn - network traffic in - bits
- netOut - network traffic out - bits
- conn - number of open connections

```
connected to: 127.0.0.1
insert  query update delete getmore command flushes mapped  vsize    res
     0      0      0      0       0       1       0     0m  2.36g     4m
     0      0      0      0       0       1       0     0m  2.36g     4m
     0      0      0      0       0       1       0     0m  2.36g     4m
     0      0      0      0       0       1       0     0m  2.36g     4m
```

(a) Console output

```
locked % idx miss %     qr|qw   ar|aw  netIn netOut  conn       time
       0          0       0|0     0|0    62b     1k     1   17:25:35
       0          0       0|0     0|0    62b     1k     1   17:25:36
       0          0       0|0     0|0    62b     1k     1   17:25:37
       0          0       0|0     0|0    62b     1k     1   17:25:38
```

(b) Console output (cont.)

Figure 3: “mongostat” console output

3. Server monitoring - “VisualVM”
Another significant indication for a thorough monitoring is the surveillance of the system state on server-side, first and foremost because early signals of overload are to be expected here. Avoiding commercial solutions, the monitoring tool VisualVM - contained in the well-known Java Development Kit (abbr. JDK) since version 6 (revision 23) [VisualVM 11] - has been used. Like most of the tools belonging to Oracle’s Java family, VisualVm has been published under GNU General Public License and hence can be used for free. It enables to keep a check on the respective load of CPU and memory, the number of current threads and loaded classes. These parameters are graphically visualized over time by respective charts. Additionally, the activity of the garbage collector and a meticulous break down of the thread allocation can be traced. For a detailed analysis in retrospect, VisualVM provides a so called snapshot functionality that creates an, in every aspect identical, image of the ongoing monitoring to be reviewed later on (cf. Figure 4).

4. User simulation - “JMeter”
The most complex component of the test environment is the software to simulate the users. While the other tools comparatively assume a simple function, this software has to offer a lot of setting options to create various testing scenarios in order to gain as much information as possible. In general it can be stated: the more data one can collect, the more significant

Figure 4: Exemplary "Visual VM application snapshot"

the results are. Hence the possibility to perform a quick and easy substitution or change of individual constituent testing parts (e.g. modeling a different frequency of queries) was preferable, too. The modular design principle of JMeter fits these requirements perfectly. JMeter's main scaffold is called "Test Plan". A set of concurrent users (each of them labeled as "Thread" in JMeter parlance) performing various tasks on the target application is located within. Every set of users is managed by a "Thread Group". It contains numerous (nested) basic modules defining the respective tasks this group of users should perform. Each "Test Plan", "Thread Group" and module has its own settings and can be adjusted to meet the required demands. After having started a test scenario every module within a "Thread Group" is executed sequentially, while all of the "Thread Groups" are running simultaneously (ie. parallel).

Figure 5 shows an exemplary setup used in the test scenario. This "Test Plan" consists of two different user type groups (ie. "Thread Groups") - one representing users posting a message via REST, the other one querying for recent messages. The respective highlighted module (in this case the messaging "Thread Group") is detailed in the right frame of the application, where the adjustments can be set. Each "Thread Group" is composed in the same way: The first two modules control different pre-adjustments like inserting the proper authorization data into the query's header or default connection setups to address the server. The "Restful PUT" is a HTTP request module containing the logic of the request (e.g. message to send, resource, URI) and has a nested timer module attached. This timer ensures that the throughput frequency of the sent requests is throttled specifically.

The last two modules "Graph" and "Results Tree" represent another important feature of JMeter. So, it is not only possible to create a testing scenario, but also to evaluate the performance of these tests at the same time. In the present case the "Graph" module generates an individual graph mainly showing performance related parameters (note: detailed examples are presented in the course of this chapter). "Results Tree" offers a precise breakdown of every request being made - and also the related answer received from the server - including statistics as well.

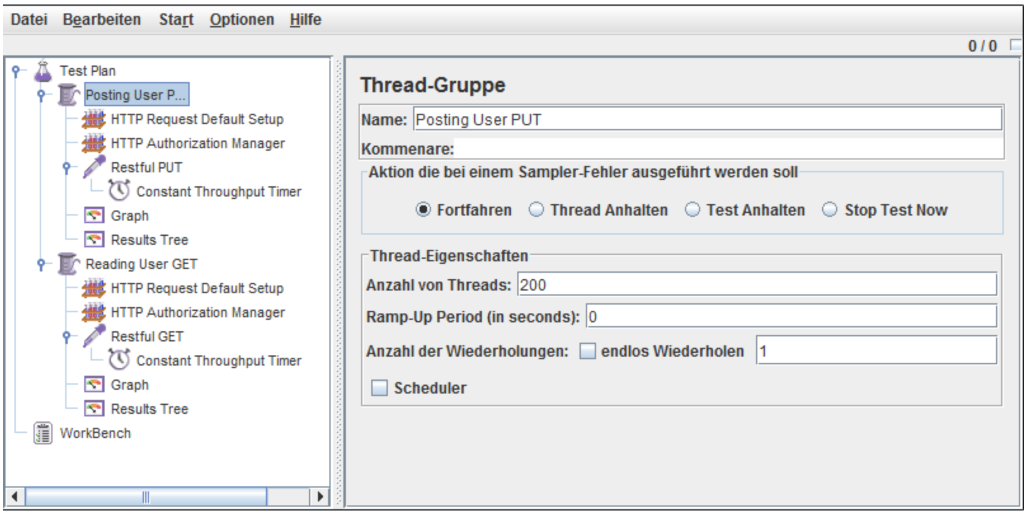


Figure 5: Exemplary "JMeter Test Plan"

Changes between two test runs were accomplished by simple parameter changes (e.g. set a higher number of threads - which means concurrent users) or the enabling/disabling of a module or a whole Thread Group. Combined with the fact that JMeter can be purchased for free (developed within the Jakarta Project by Apache Software Foundation) and the possibility to extend the functionality by several available custom modules (designed and published by the user community), JMeter is a highly suitable solution to simulate any desired scenario.

7.3 Preliminary considerations

Supported by this plenitude of analyzing tools, the last preparation step before testing was to define the structure of the respective components. Due to the absence of experienced data regarding an improvement of the monitoring parts, mongostat, VisualVM and the Glassfish v3 server were initially used with their default settings. If it turned out that customized settings were indispensable, either to obtain more detailed results on specific aspects using a monitoring component or to increase the performance of heavily loaded system components (e.g. by enlarging caches of the application server), one still has been able to perform step-by-step adjustments and document them individually.
A very important rule in this context (i.e. the modification of the settings), which has been applied throughout the whole testing, refers to the frequency of adjustments: Only one single setting should be changed per test run. The reason for this is to keep transparency, traceability and informative value of the obtained results. One only imagine the resulting chaos trying to identify the decisive factor of better or worse performance if multiple changes were applied between two tests.
A last consideration affected the initial setup of the primary JMeter test. The first test has been designed to reflect a scenario pretty close to reality to receive an impression of Backstage's current capabilities regarding the later field of application. It has to be pointed out here again that, prior to the testing, not a single definite reference point concretizing the performance existed. Having the potential increase of complexity in mind, the first test had to be kept simple and manageable at the same time. Once a certain basic knowledge on the behavior of the application under load evolved, the tests still could be revised and modified to fit a more complex but potentially more realistic scenario. The tradeoff between these requirements constituted the first Jmeter test as follows:

- The present main functionalities[8] of the application - querying for a list of messages and posting new messages - were integrated and modeled under realistic conditions (cf. upcoming calculation).
- In addition to monitoring the data with mongostat and VisualVM, the querying and posting of messages was analyzed respectively by using the “Results Tree” as well as a modified “Graph” module, developed by Rubén Laguna [Laguna 09]. This revised visualization is reduced to only show throughput and response time of the current queries over time, replacing the overburden and confusing standard figure (cf. Figure 6[9]).
- The modeling of DWR-related features has been abstracted away from an initial configuration and was integrated in later test runs (cf. Chapter 7.6).
- To simulate the most exhausting case, the maximum allowed size of 160 characters per message was used[10]. By contrast, the content of the messages could remain the same because each message traverses the same parsing process and does not get cached within the application. Consequentially there was no need to create dynamic messages.
- The process of authentication preceded every single request due to the independent and self-contained character of RESTful queries (cf. Chapter 2).

Based on the fact that an authentic use of Backstage includes simultaneous posting and receiving messages, both functionalities were mapped to two synchronous thread groups. Each thread group included the same course of events - Authentication, HTTP request containing the logic of getting or posting messages, Results Tree and Graph (cf. Figure 5). The final option to select was similarly the most important one: The intensity or, more technically spoken, the frequency of the queries. On this, the above-mentioned throughput timer offered the possibility to apply a constant level of load - at least as long as the total capacity of the networking infrastructure was not exceeded. The value to set is given in samples per minute. To determine a proper factor one first needs to define a desired number of concurrent users. A value of 200 users, resulting from the average estimation of a considerably frequented lecture, has been chosen here. This number (i.e. “Threads” in JMeter context) additionally had to be set in the respective Thread Group properties. If one further assumes that a very high-frequency writer is physically limited to write an expedient message[11] every five to six seconds (note: this number could be used as threshold value identifying potential spamming users), a value for the throughput timer can be determined using the formula

$$f_{[N_{queries}/min]} = N_{users} * \left(\frac{60}{\triangle t_{posts}} \right) \tag{1}$$

in which $\triangle t_{posts}$ is the temporal difference between the sending of two messages regarding one user (five to six in the present case). This results in:

$$f_{[N_{queries}/min]} = 200 * \left(\frac{60}{[5,6]} \right) \quad \Rightarrow \quad f_{[N_{queries}/min]} = [2000, 2400] \tag{2}$$

Accordingly, a total throughput demand of 2000 to 2400 queries per minute was set for the timer. One might argue that these values are exaggerated, especially because not every single user is using the application to its full capacity at the same time. Though, they are intentionally targeted at the higher end of an imaginary load scale simply because this test was designed to simulate an extreme case.
Having constituted and configured the complete testing environment a first test run can finally be performed.

[8] In terms of “identified as the most commonly used ones”
[9] These figures were created by Rubén Laguna [Laguna 09]
[10] Generated by the “Lorem ipsum generator”[Jones 07]
[11] Users intentionally posting spam messages have been disregarded here, provided that a reliable protection is present

(a) JMeter's original "Graph" module

(b) New "Graph" module

Figure 6: Comparison of "Graph" modules

7.4 First results and adjustments

Probably the most wishfully expected moment for a software developer is to see how the long development phase of an application ends in the very first use under real time conditions. At this, a certain prospect of the performance is of course the most desired result, especially because a general functionality often can be assured from small, preceding functional checks during the development.

Consequently, the first test run has been observed with greatest interest, but the obtained facts were, contrary to the expectations, quite disappointing in the end.

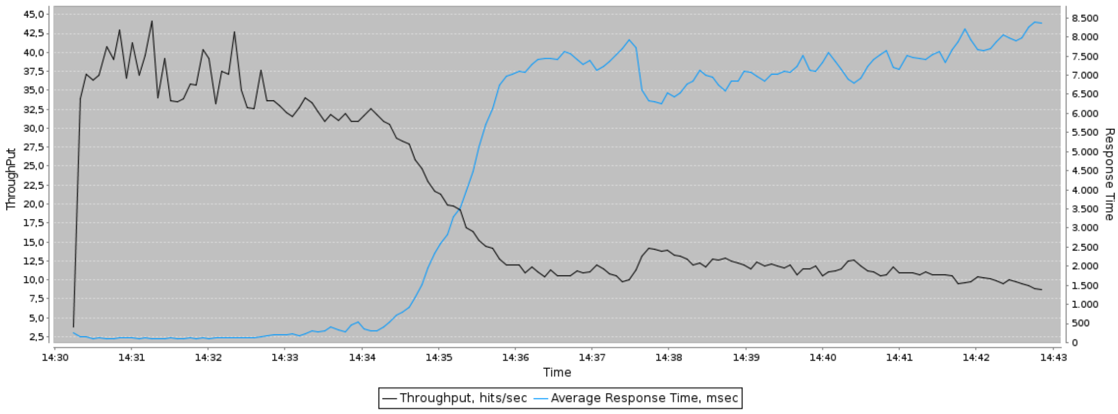


Figure 7: Exemplary GET result of the first test run

At first, everything went the intended way: The posting and receiving of messages was performed properly and - as the Results Tree module showed - without any failure. Postings were persisted as entries in the database, the sever-monitoring VisualVM displayed fair amounts of load of CPU and memory and also the response times output by JMeter, in the end the most important reference point, were within a satisfying range of about 70-80 milliseconds.
But running this test over time (an initial period of ten to fifteen minutes was intended) the result changed considerably. As Figure 7 exemplary shows for the GET queries[12], after approximately four minutes the response times of both sections, PUT as well as GET, began to increase dramatically (cf. the blue graph indicating the average response time in milliseconds, accordingly the black graph shows the average throughput in hits per second). Within two more minutes a linear increase to the maximum response time of over seven seconds per request, which consequently is far beyond any useful limit for a proper operating with the application, was reached. Thereafter the increase flattened, but still grew steadily until the abortion of the test after thirteen minutes, reaching a total response time of eight seconds per request. To exclude the possibility of any temporary coincident event (e.g. memory allocation problems), the identical test was repeated twice, showing the exact same result in both runs. Even if one could argue that the response times worsen if they are measured within the same tool that is performing the load test[13], an increase by a factor of one hundred had to have a more serious cause than an average parasitic error.
Accordingly, a next step was to find the potential source of this impairment. For this purpose, a closer look at the recorded results seemed consequential. A notable first evidence was the decrease of load of the application server's CPU negatively related to the increase of the average response times. This behavior indicated that the CPU was able to handle the initial amount of queries. It also revealed that deficient CPU power could not be the source of the deceleration. Otherwise, if the CPU was responsible for the trip, its load would rather increase or permanently stay at one hundred percent than being reduced as it was the case here.

Still accounting the application server as most probable starting point for the problem, a few more aspects were taken into consideration. VisualVM showed a constant, brimful filled memory section called PermGen - an abbreviation for "Permanent Generation". This is a specially reserved range of the memory pool containing data needed by the Java Virtual Machine (abbr. JVM) like objects describing classes and methods of the application [Sun Microsystems, Inc. 03]. A possible explanation would be that its size was initially chosen much too small and the overhead to reload

[12] Every graph included in this section is intended for a general overview purpose. Due to the size and the limited space on the page one might has to refer to high resolution pictures included in the annexed CD for details (cf. table of content on page 46)

[13] Simply originating from the simultaneous workload of measuring and posting combined with response times within millisecond range

uncached, necessary data (and thus further processing interrupted requests) caused the problem, which accumulated and built up over the time running.
Another approach affected the number of threads used by the application server to divide the load and decrease the individual response time if facing a huge amount of queries. If the maximum number of threads to create has been reached over time, the individual response time could eventually have been enlarged more and more. Investigating these aspects, two separate tests have been performed. The first one used a bigger PermGen memory allocation of 512 megabytes - instead of 192 megabytes as it was the default setting. The second test only increased the maximum allowed number of threads created from 50 to 200. Factors this high (in each case multiple times the standard setting) have been chosen to definitely be on the safe side regarding the observation of any potential effects.
Regrettably, the changes were noneffective and the results showed the exact same characteristics as earlier tests. Further research proved that the size of both factors was set dynamically by the application server with regard to the respective needs. In fact the borderline VisualVM displayed was not the maximum capacity but a dynamic hysteresis. Revealing no further peculiarities, the application server had to be excluded from further investigation and a different explanation had to be found.
To abdicate the user simulating part's responsibility for this case, the server executing JMeter was next to check on. Because JMeter showed no specific abnormalities (remember: every request was marked within the Results Tree module as successful so far) - only a quick check on the most revealing system properties has been performed with the result that CPU (8-10% load), memory (27% load) and network (approx. 150 kilobytes per second) loads were far below each upper load limit. Though, one more test scenario has been set up to diminish any remaining doubts: The original test scenario was split into two different instances of JMeter, seperated by the respective type of user simulation. The first one simulating the message posting users remained on the initially intended server, the other one realizing the users querying for messages was relocated to the database server. This way, the potential effect of a mutual interference of the different Thread Groups should be assessed. As almost presumed, again both graphs revealed the well-known, abruptly increasing response time pattern.
Since every modification up to this point ended in the same result and further reasons were gradually falling off, also the possibility that the failure originated from one of the testing components narrowed down. In this context it has to be reminded that the whole testing procedure took place within the network of the LRZ. Hence, the present subnet could definitely be subject to specific security policies like a restriction of the queries per node over a defined time period to avoid congestion. This perception could also explain the obtained graph: At first everything is working properly until a certain limit of queries has been reached. At this point the throughput is throttled down drastically (cf. the exponential increase of the average response time in Figure 7) to encounter congestion or intended flooding of the network. Once a predefined level is reached, a softer slow down approach (cf. the linear progress after the exponential increase of the average response time in Figure 7) could have been applied to still guarantee a completion of the request. Being so, one would at least have a functional test of the network policies. So much for the theory. In practice, however, one still had to prove the speculation. Given that an entire local configuration of the testing environment would have required a completely new adjustment of the servers and testing parameters, which again could have created new sources of error, at first a consultation with a person in charge (e.g. a network administrator of the LRZ) seemed the more practical solution. The obtained statement finally refuted the theory by stating that in fact there were certain restrictions to avoid congestion, but they were dimensioned for much bigger rates compared to the ones resulting from testing Backstage and accordingly were out of the question. After having discarded this explanation as well, a first summary of the characteristics of the problematic behavior gained so far could be concluded:

1. The first few minutes of every test ran properly. Consequently the application was working to a certain degree.

2. Since almost the exact same failure pattern resulted from every test and a network related issue had to be ignored, an external failure could practically be excluded.

3. Most probably some sort of accumulation issue had to be responsible for or at least related to the effect. Otherwise, a performance deficiency would be observable from the beginning.

4. The reason could still be originating from the application itself[14].

5. Of the three main test components only the database was left for inspection up to this point.

7.5 The “count()” incident

Since a reliable and trouble-free persisting of data has been taking place throughout all tests, MongoDB seemed to be the least likely cause of failure. Drawing the above conclusions, a closer examination of the database was still inevitable. The most obvious approach regarding accumulation as a decisive factor seemed to focus on the saved posts. Accordingly, the next test included a pre-filled database approximately consisting of sixty thousand posts, which resulted from the last run.

Finally, a first hint arose. A huge delay of the response times accompanying a very low throughput, similar to previous high load phenomena, appeared immediately after the start throughout the whole test. Also the CPU load of the database server amounted to nearly 90 percent, which outnumbered the expected maximum load by far. To verify these results and clearly identify the database as the source of the problem, the subsequent test was designed to combine the earlier obtained results with the latest findings. Hence, this test has again been performed with an initially empty database, but additionally included an on-the-fly modification: As soon as the response times reached a critical level, a manual deletion of all posts within the database was triggered. If the response times decreased notably after the erase instruction took place, the message quantity would prove as decisive factor. Figure 8 shows the recorded behavior that exactly visualizes the anticipated results. As one can see, the manual deletion was performed twice (cf. time points 12:36 and 12:44) and each time entailed an abrupt decrease of the response times (and so an improvement of the throughput) until the database was filled again. Further, the minimal number of messages causing a critical state was identified at approximately eleven thousand. Imaging a real time setup using a central database server, this number was highly insufficient.

As consequent further development an accurate breakdown of this problem had to be accomplished to create at least a basis for any future use[15]. For this purpose the separation of GET and PUT requests was the aim of the following tests in order to further narrow down the problem. Since the amount of messages was somehow related to the source of the problem, the primary investigation of the generating instance (i.e. PUT queries) was self-evident. Thus, the next test included only PUT requests at a rate of two thousand messages per minute. Surprisingly, the response times were unexpectedly low (about 12.5 ms) over the whole testing period performing at the desired rate (the test was aborted after the sending of approximately 65 thousand messages). To confirm the results and eliminate PUT as a potential cause, a follow-up test at a rate of 12000 messages per minute, which would correspond to 1200 users posting every 6 seconds or 200 users posting every single second (cf. Formula 1), was performed. Figure 9 illustrates the exciting effortlessness of the system handling these intentionally exaggerated rates[16].

[14] So far, Backstage as a prototype has never been subject to any optimizations

[15] At this point a general impracticalness of the whole application was conceivable

[16] The increase of the response times at the very end of the test resulted from an unintended modification and can be neglected

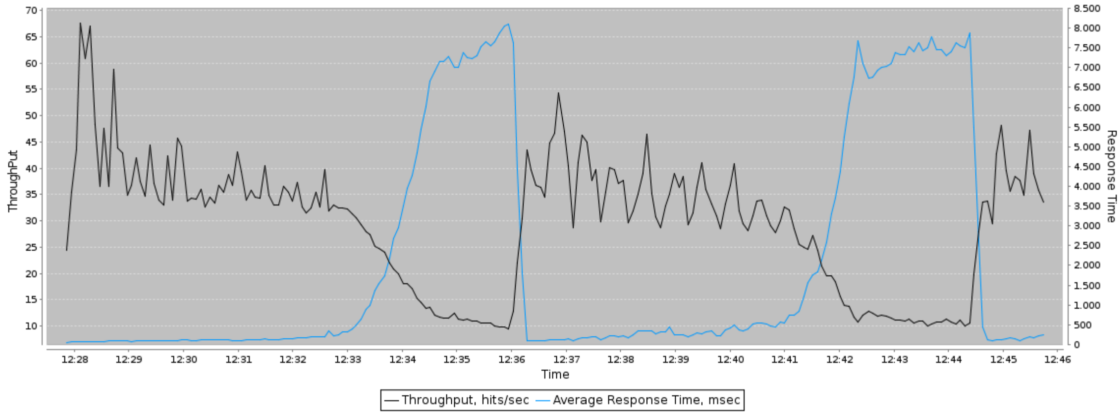


Figure 8: Confirmation of MongoDB as the source of failure

Considering these results, the focus was now on the GET component. Its accountability for the negative impact on the latency could be verified by performing a GET-only variant of the test running at a rate of 2000 queries per minute. Of course the database had at least to be filled with the earlier identified, critical amount of messages to cause notable effects, which was achieved by keeping the 65000 messages of the preceding PUT-only test in the database. An instant, intense increase of the response times from the beginning provided evidence of an existing GET-related problem. To substantiate the deficit, an experimental diminution of the GET frequency has been evaluated. The result was disappointing: Not until a minimal number of 150 GET requests per minute was reached, the response times were within a reasonable range of 110-130 ms. This fact clearly emphasized the shattering effect of the problem: Having only 65000 posts stored in the database, the system was barely able to support 150 users requesting evenly spread once every minute without adding new posts. As these numbers mark the upper limit of capability even in terms of load peaks, this problem could definitely be seen as a knock-out criterion for the whole project.

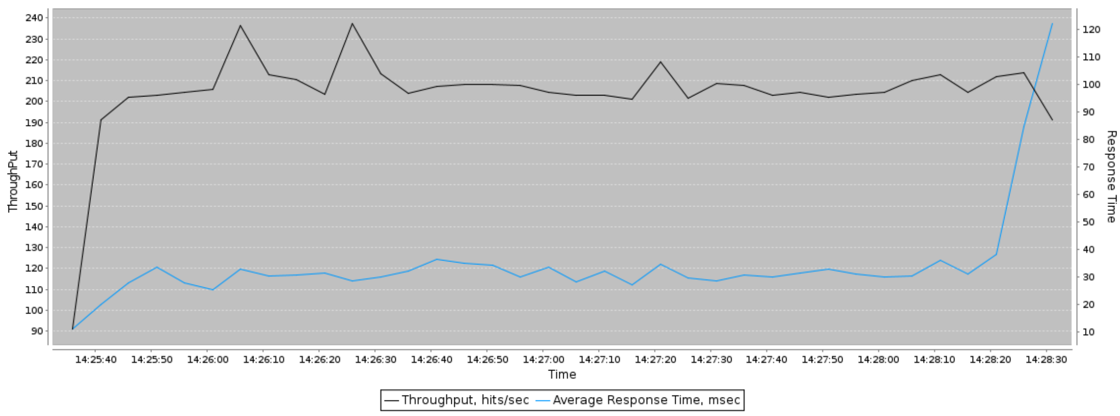


Figure 9: PUT performance test at the rate of 12000 messages per minute

Having scrutinized the concern this far, an exceptional examination of the source code was a consequential, next step, especially because there was no further modification of the test environment that could have altered the results. The main method handling the the GET request basically consists of two parts: on the one hand the fetch of the requested posts from the database and on the other the rendering of these messages. These two mechanisms are surrounded by the pagination mechanism. The rendering mechanism could be excluded from consideration for two reasons. First of all, this operation is performed independently by the application and thus in no way re-

lated to the database that has been identified as a responsible part. Second, the effort to perform the rendering is the same for every instance (due to the predefined number of messages per rendered page) and thus does not show any potential of accumulation. On the contrary, the fetch of the requested posts is querying the database.
Further tracking the process of the message retrieval from the database, a highly suspicious entry in the domain model of the posts could be found: The “dateCreated” field, responsible for a proper sorting of the messages by time, was not provided with a functional index. This index is designed to expedite the search in the database for certain entries and as a result it can shorten the sorting notably. Being absent, the costs for a search of the desired content multiply to a considerable degree if the number of posts increases. If the additional expenses get too big, it could entail the observed accumulation and accordingly be the trigger for whole problem. After having added the index to the entry in question, a new GET-only test run was performed to inspect the achieved effects. Yet again an improvement of the results failed to appear.
After there was no definite finding examining the source code, almost any option for a revealing analysis had been put to the test. Consequently, a more precise analyzing method, in particular applying to the database had to be found. Instead of introducing yet another complex tool, at first an additional, quite simple feature of MongoDB, the so-called “database profiler”, as last alternative was given a try. This profiler has in principle the same effect as a filter mechanism upon the processing time of the operations performed by the database. If one, for example, adjusts the profiler to a specific threshold value, only operations lasting longer than the selected time are being displayed. Additionally, the granularity of the filtered operations can be determined in a two-option selection including a coarse and a fine mode. While the coarse mode performs a pre-filtering by only regarding slow operations (classified by the database), the fine mode includes every operation performed. As one can see, the database profiler is a very simple tool to discover single bottlenecks within the database.
To receive every appearing non-standard delay, a fine-mode examination with a respective threshold value of 50 milliseconds had been selected for the upcoming analysis, which still focused on the GET-only layout of the previous tests. The resulting findings were extremely astonishing: The only operation consistently showing abnormal delays between 60 and 230 milliseconds, with a mean average of 160 milliseconds, was a simple *count()* method. Within the current application testing this native database operation was executed only once in the context of the pagination mechanism to determine the maximum number of available pages and denominate them accordingly. However, count() as the definite source of the delay perfectly matched any characteristics obtained so far:

- It is located within the database component.
- It is not a deficit of the hardware components or the testing environment and their respective settings.
- It is only present in the GET-case scenarios.
- And finally, it is directly related to a parameter that accumulates over time (i.e. the amount of posts stored in the database)

Even though this count() incident caused severe impacts on the overall performance of Backstage, its correction was comparatively simple. The most obvious and direct solution was the removal of the count() method which would cause a cessation of the complete pagination mechanism. Since in general an already existing and working feature is only dismissed reluctantly and regarding the fact that many other operations throughout the whole source code, beneath the ones currently put to the test, were utilizing the same count() method, a more sophisticated solution had to be applied.

Evidently a count() method simply returns the current amount of a certain type of data, which changes over time. In the present case a variable counting the current amount of persisted messages was returned whenever count() gets called. Having a corrupt count() potentially caused by a failure of the internal logic of the MongoDB, an efficient workaround would be to emulate its behavior and replace it by an externally administered atomic increment. This construct describes a single variable that is also stored in the database holding the current number of posts. Since every lecture stored in the database obviously includes a different number of messages, one variable for every lecture administered by Backstage has to be created. In contrast to the count() method, this variable is not updated by the database, but by the attached logic of the application. For example, whenever a post is received via GET, it is not only Backstage's responsibility to assign the storage of this post in the database, but also to increment the variable holding the number of posts. The same thing applies to the deletion of posts in the form of a decrement.
Additionally, this variable has to be declared "atomic", which ensures the exclusive access to the counting variable (i.e. one at a time) to avoid an inconstant state that could cause a wrong counting. One might argue that such an extensive procedure is not in relation to an ordinary database method, but the doubtless and obvious faultiness of MongoDB in this matter legitimates the insertion of such a workaround. This kind of adaption also altered the devision of work. An atomic increment always keeps the current value at hand whereas count() does not start a counting until the actual method was called. Accordingly, the atomic increment needs additional processing every time a message arrives or gets deleted to keep the variable up-to-date. Though, regarding the actual extracting of the current count, the maximum complexity using an atomic increment is only determined by the number of the different lectures needed to be scanned. Assuming that the single update procedure of the count variable can be managed without losing much performance - a potential effect would be the increase of lock percentage since the count variable is defined atomic - the pure count retrieval procedure changes its complexity dependency from the number of posts to the number of lectures. Given the fact that the original count() performed exceedingly slow on a certain level, such a context switch and the distribution of load should certainly be beneficial for the performance.
After having performed the described changes, the only way to finally verify the correctness of the made assumptions and the functional efficiency of the workaround was another testing. The pre-adjustments included a filling of the database with 100000 posts to provoke a count()-crucial scenario from the beginning. The GET and PUT were both set up for running on the same server again. The frequency of GET was set to 3000 queries per minute, the PUT frequency to 2000 queries per minute and both were performed simultaneously again (corresponding to the very first tests)
This time the obtained result were highly satisfying (cf. Figure 10). Compared to every other test conducted so far, for first time the perceived outcome was highly congruent to the intended expectations. The key-facts in the following listing can substantiate the success of this test run:

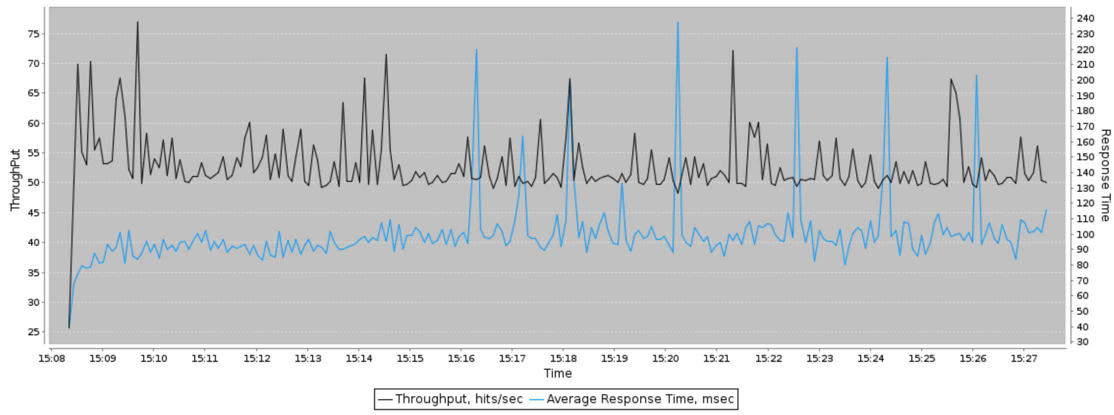
- The MongoDB database profiler showed no more operations lasting longer than 50 milliseconds

- The amount of read/write locks in the database increased from almost zero to approximately 4-5 percent, which matched a realistic access restriction for the applied settings.

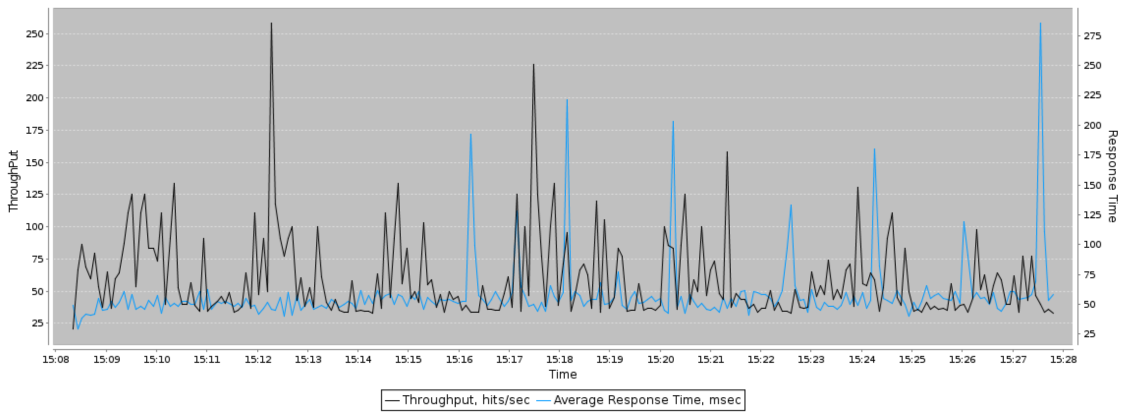
- The CPU load of the database server dropped from over 90 percent to around 10 percent in every core, which clearly indicated a huge performance gain and thus a first hint of the true potential of MongoDB.

- In contrast, the CPU load in the application server increased notably to around 80 percent. Now, for the first time not the database server but the application server was the current delimitating factor

- While fulfilling the demanded throughput rates, almost perfect results for a smooth supply of the listeners under heavy load conditions have been achieved having an average PUT response time of about 50 milliseconds and an average GET response time of about 100 milliseconds.
- The average response times did not increase over a period of 20 minutes with a final database count of about 150000 entries and showed constant values. Accordingly, no more accumulation issues were observable.



(a) GET graph



(b) PUT graph

Figure 10: Test results of a count()-free implementation

7.6 Integrating DWR

Coming to the conclusion that the current, yet not optimized, Backstage implementation is already able to handle a considerable amount of users under heavy load conditions, a promising condition for the future continuity of the whole project can be stated. However, the results obtained up to this point were not very representing concerning an actual scenario because of their static nature. It is assumed that most of the users following a Backstage conducted lecture mainly use the web interface of the application. In doing so the highly dynamic DWR-related features are playing a decisive role regarding the computational complexity of Backstage. One might argue that these additional expenses can simply be simulated by performing static tests - like the ones performed by now - at a higher rate, but a correct interpretation of the results seems nearly impossible regarding the following facts: On the one hand the relation between the costs to perform a simple static

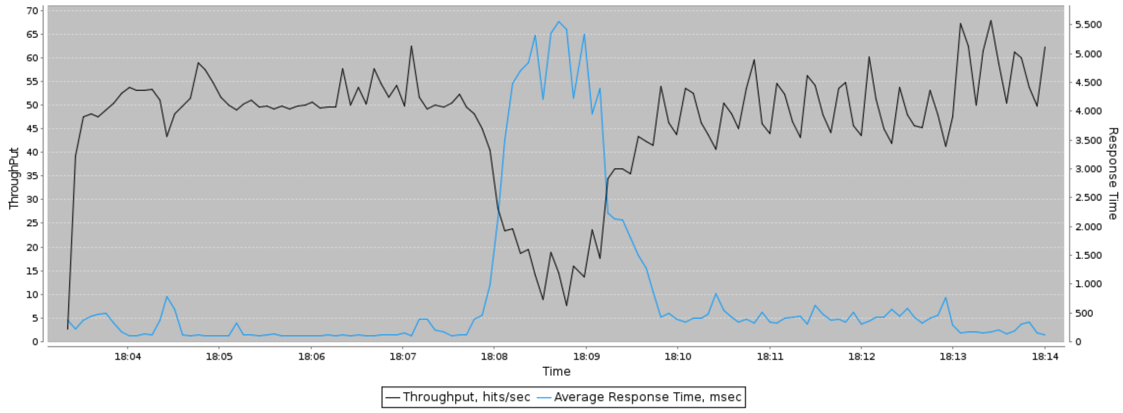
call and a dynamic variant is not known, on the other hand a precise and faultless effectiveness of the whole dynamic section cannot be assured by just using an extrapolation of the static one. Since it is also absolutely conceivable that DWR causes the major part of the computational effort, a proper integration of it is indispensable to reproduce exact and significant testing.
To integrate DWR to its full extent two basic parts had to be replicated. The first thing supposed to create additional complexity was the traversing of the registered script sessions (representing the users participating via the standard web interface) on the arrival of a new event (e.g. a message) to determine its relevance for the individual user (cf. private messages, which only affect a subgroup of registered listeners). Fortunately, this traversing is performed by the DWR engine itself and one only has to provide an appropriate number of registered script sessions. The creation of these faked script sessions, however, required additional expenses described later on.
The second requirement was to simulate the periodic polling of the DWR-registered users. As stated earlier (cf. updating the client Section) every client connected via web interface (more precisely every DWR-instance attached to a listener using the web interface) is polling the server, in particular the DWR-engine, in periodic intervals automatically to check on relevant, recent updates. This additional load had to be considered as well to create actual quantity and distribution of load to the greatest possible extent.
However, both of these processes required a closer examination at first in order to develop a proper realization. To detect the necessary steps of a scriptsession registration process and examine a client emitted DWR-engine polling, another analyzing tool monitoring the web traffic had to be introduced. Since the well-known network traffic observation tool “Wireshark” seemed to be too excessive identifying potentially simple information exchange on the application layer, the more elementary analysis add-on “Firebug”, extending the installed “Mozilla Firefox” browser on the server, was chosen. “Firebug” is a free[17], cross-platform website monitoring tool in particular relevant for testing DWR-related features because of its Java-Script based console that is able to monitor AJAX activities. Accordingly, a detailed description of the required steps to register a scriptsession and perform the polling could be extracted without any problem (cf. annexed data medium). The outcome of this monitoring was quite conductive. It showed that a valid and totally DWR-accredited scriptsession could be created by the use of two simple HTTP requests.
The first request was directly addressed to the DWR-engine - the main module of the DWR component. Querying this engine using a standard GET request triggered the creation of a new scriptsession by the engine on server side. The answer to this request contained the whole information regarding the new scriptsession. To actually cause an effect, this scriptsession had to be linked in a second step with the currently tested lecture. To achieve this, at first two relevant identification parameters “httpSessionID” and the “scriptsessionID” had to be extracted from the answer of the preceding GET request using a standard JMeter “Regular Expression Extractor” module. By emitting a second GET request that included these parameters - among others indicating the intended lecture and custom settings - the newly created scriptsession got attached to the lecture. In contrast to the first GET request the present one was not directed towards the DWR engine, but towards the standard address for plain DWR calls (note: A detailed review of the whole procedure’s setup can be reviewed in the “scriptsession faker.jmx” file stored on the attached data medium). By repeating this process an arbitrary amount of fully functional scriptsession dummies could be created.
As almost expected, the polling of the application server for potential updates showed a similar format and was also accessible by a plain HTTP GET request. Again, this request was directed to the standard DWR calling address using different parameters.
The huge benefit of this finding for the DWR testing process was that both functionalities, registering the script session as well as the periodic polling, could be completely emulated by JMeter. Thus, it was not necessary to integrate further testing tools that could additionally impair and complicate the evaluation. The synchronous monitoring of ordinary DWR-related and RESTful load

[17]Distributed under the New BSD license

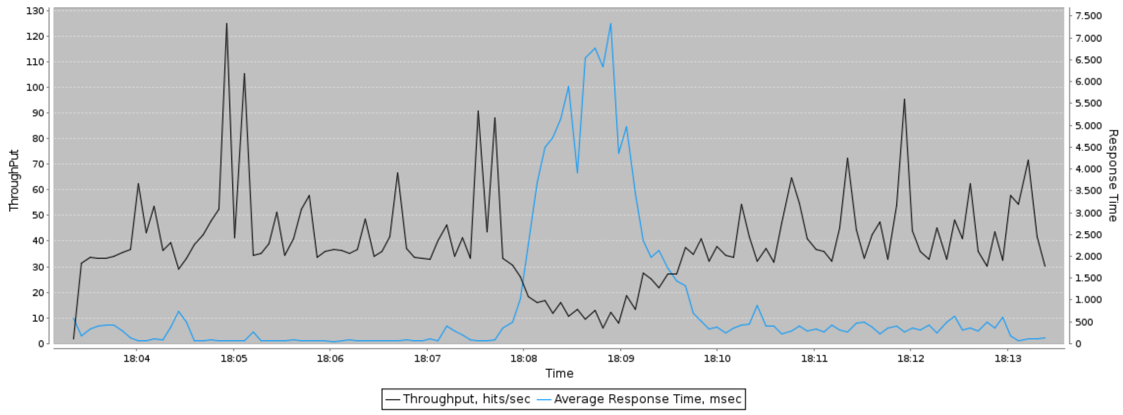
scenarios was also facilitated by retaining only one load simulating part.
While the concrete integration of the new tests with JMeter was very straightforward and thus is not further described at this point, the layout of the tests, however, changed notably to a two step model. Before the execution of the actual test - which now consisted of the three components GET, PUT (known from previous tests) and the periodic DWR-related polling - the creation of the faked script sessions had to be enforced. In a final step two important configuration modifications had to be performed within the application to ensure thorough testing: On the one hand the limitation of the maximum idle time of a DWR-registered listener had to be removed or at least prolonged. Otherwise, the faked script sessions would have been terminated by the DWR logic after a preset time of 5 minutes and a complete reinitialization would have been necessary. On the other hand, the direct access to the DWR-engine - the target of the POST queries to fake the script sessions and the periodic polling - had to be allowed explicitly within the DWR configuration. In general, this instance is locked for obvious safety reasons and should only be made available for the time of testing.
By having implemented these modifications, everything had been prepared for one last test operating as close as possible to real, heavy-load conditions and using the following settings:

- Initially 200 dummy scriptessions were registered to simulate a proper DWR load.
- The database has been filled with an average amount of 50000 posts.
- PUT requests have been performed at a frequency of 2000 queries per minute.
- GET requests have been performed at a frequency of 3000 queries per minute.
- Periodic DWR-polling requests have been performed at a frequency of 3000 queries per minute.

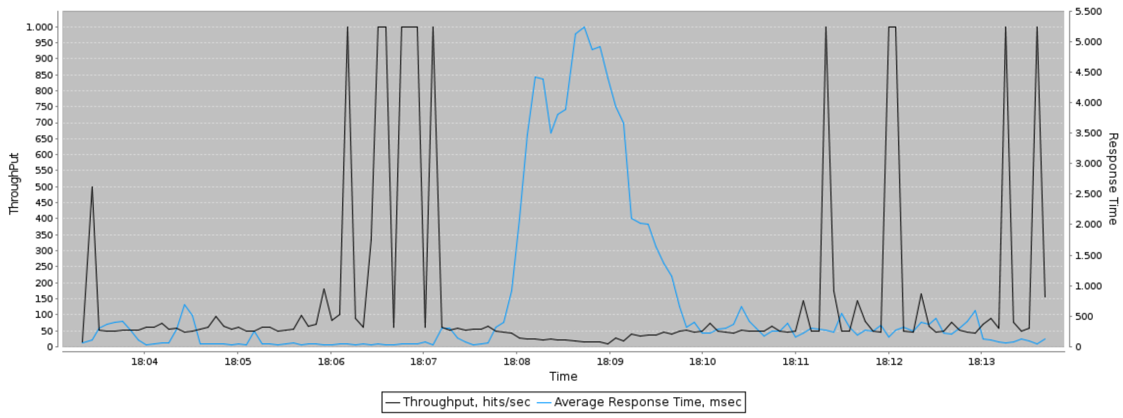
With the help of Figure 11 the final results can be summarized as follows:
Most noticeable is the huge exponential increase of the average response time (cf. blue plots) and the simultaneous decrease of the average throughput (cf. black plots) over the time of more than a minute in the middle of the test. What looks like the result caused by the faulty count operation at first sight is in fact the confirmation of an issue that occurred while running this particular test for the first time. There was an inexplicable increase of the response times for unknown reasons, too. After researching it turned out that a web browser's view showing a logged in user's web interface of Backstage was still running in background on the application server. Before the actual testing began this view was used for setup purposes. Being still enabled, the browser tried, of course, to constantly update its view. Having in mind that approximately 33 new messages per second had to be graphically displayed and at the same time every occurring request of the load test had to be processed, an abrupt decrease of the performance becomes an inevitable consequence. In the current test, these results were inspected by activating a standard user's web view on the application server for an arbitrary time during the running period. Hence, the peak in the Figure 11 visualizes the verification of this phenomenon. After having terminated the active window the response times began to drop rapidly to expected values.
Consequently, for an assessment of the performance accomplishments only the remaining parts of the graph - disregarding the peak - were taken into consideration. It can be stated that over a period of ten minutes every single component of the test (i.e. GET, PUT and POLLING) showed an equivalent performance pattern. Backstage was able to answer every request directed towards the server within a maximum mean delay of about 0.3 seconds while satisfying the intended throughput of each component. Without observing any effect of accumulation, these results are highly sufficient regarding the current prototype stage of this project.



(a) GET graph



(b) PUT graph



(c) POLLING graph

Figure 11: Test results of the DWR-integration

As a final conclusion of the testing process a short comparison between the obtained values and the equivalent number of users with their respective activity can be drawn to be able interpret the presented results of this last test and understand their relevance correctly:
The above-mentioned numbers model an ongoing lecture activity attended by 200 simultaneously active listeners that are connected via the web interface (thus using DWR). “Active” means that they are using Backstage at a high frequency. More precisely, every single listener emits a public message every six seconds. Every four seconds the DWR context of each user additionally polls the server to check on potential updates (i.e. new messages). Since at a PUT rate of 2000

messages per minute approximately 33 new messages per second reach the server, there is always an update to perform. Consequently every four seconds each user requests the latest updates and the server returns a list of the ten most recent messages (note: the real amount of messages provided by Backstage for a listener using the web interface might differ. This number is owed to the use of the REST interface as described in Appendix I). Every single request for every single user can be fulfilled by the server within a total latency of about 350 milliseconds. Having in mind that only major failures impeding the overall functionality have been fixed and no detailed performance improvement has been applied so far, this performance is quite impressive and gives a first indication of the inherent capabilities of Backstage in a standard server environment.

8 Conclusion

Summing up, a brief recapitulation of the achieved innovations in the context of Backstage as well as a listing of related, future improvements and resulting research approaches are presented.
Up to this point it succeeded to extend Backstage by a new interface following the architectural web style REST. For this purpose a mapping of Backstage's main functionalities to self descriptive URIs has been performed. Further, the bridge between the highly dynamic content of the initial application and the stateless and static approach of REST has been built. At the same time it was able to refactor most of the present source code to avoid duplication to the greatest possible extent. An estimation on the attendant security issues was given as well. In a second step a practical example of use in form of a load testing has been applied to Backstage. Now, for the first time one was able to actually measure the capabilities of Backstage. After having detected and fixed a major bug in a standard routine of the database MongoDB, a thorough applicability for the intended use cases could have been proven.
Based on this primary conceptional elaboration further research could cover the following topics:

- The complete implementation of the remaining features/branches.
- Systematic content negotiation via Backstage's REST.
- Enhancement of the security by the introduction of API key or OAuth.
- Targeted search for source code optimization to improve performance.
- Long-term testing of the application in a real environment.

List of Figures

Content of the enclosed data medium

```
RestfulBackstage/
├── 01 - Thesis
│   ├── 01 - RESTful Backstage
│   │   ├── 01 - PDF
│   │   └── 02 - LaTeX
│   └── 02 - Appendix I REST API
│       ├── 01 - PDF
│       └── 02 - LaTeX
├── 02 - Testing files
│   ├── 01 - Software
│   │   ├── 01 - Apache JMeter 2.4
│   │   ├── 02 - Visual VM 1.33
│   │   └── 03 - REST Client 2.3
│   ├── 02 - Configuration files
│   │   └── 01 - Apache JMeter Config
│   └── 03 - Results
│       ├── 01 - JMeter graphs
│       ├── 02 - Visual VM application snapshots
│       └── 03 - Informal test notes
├── 03 - Data sources
│   ├── 01 - Cited book sources
│   └── 02 - Cited web sources (digital copy)
└── 04 - Source code
    └── [...]
```

References

[Berners-Lee, et al. 05] T. Berners-Lee, R. T. Fielding, L. Masinter: Uniform Resource Identifier (URI): Generic Syntax. Network Working Group, Request for Comments: 3986, Status: Standard, January 2005.

[Bloomer 92] J. Bloomer: Power Programming with RPC O'Reilly Media Inc., California, September 1992.

[Erenkrantz 06] J. R. Erenkrantz: Architectural Styles of Extensible REST-based Applications. ISR Technical Report UCI-ISR-06-12,Institute for Software Research, University of California, Irvine , August 2006

[Ferraiolo, et al. 92] D. F. Ferraiolo, R. Kuhn: Role-Based Access Controls. In 15th National Computer Security Conference, pp. 554-563, Baltimore, 1992

[Fielding, et al. 99] R. T. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee: Hypertext Transfer Protocol – HTTP/1.1. Network Working Group, Request for Comments: 2616, Status: Standards Track, June 1999.

[Fielding 00] R. T. Fielding: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, Information and Computer Science, University of California, Irvine, 2000.

[Hammer-Lahav 10] E. Hammer-Lahav: The OAuth 1.0 Protocol. Internet Engineering Task Force (IETF), Request for Comments: 5849, accessed March 31, 2011.

[Franks, et al. 99] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, L. Stewart: HTTP Authentication: Basic and Digest Access Authentication. Network Working Group, Request for Comments: 2617, Status: Standards Track, June 1999.

[Holtman, et al. 98] K. Holtman, A. Mutz: Transparent Content Negotiation in HTTP. Network Working Group, Request for Comments: 2295, March 1998.

[ITU-T 91] ITU-T study group 17: Data communication networks: Open systems interconnection (OSI); security, structure and application. CCITT (The international telegraph and telephone consultative commitee), Recommendation X.800, Geneva, March 1991.

[Kay, et al. 09] R. H. Kay, A. LaSage: Examining the benefits and challenges of using audience response systems: A review of the literature Computers and education no. 53, p. 819-827, Ontario, May 2009.

[Loreto, et al. 99] S. Loreto, P. Saint-Andre, S. Salsano, G. Wilkins: Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP. Internet Engineering Task Force, Request for Comments: 6202, ISSN: 2070-1721, April 2011.

[Makice, et al. 09] K. Makice, R. Romano: Twitter API - Up and running O'Reilly Media Inc., California, March 2009.

[Reenskaug 03] T. Reenskaug: The Model-View-Controller (MVC) - Its past and present University of Oslo, 2003.

[Richardson, et al. 07] L. Richardson, S. Ruby: RESTful Web Services O'Reilly Media Inc., California, May 2007.

[Rocher, et al. 09] G. Rocher, J. Brown The definitfe guide to Grails - Second edition Springer-Verlag, New York, 2009

[Shneiderman, et al. 04] B. Shneiderman, C. Plaisant Designing the user interface: strategies for effective Human-Computer Interaction Addison-Wesley Longman, 5th revised edition, Amsterdam, March 2009

Online references

[Dewanto 09] L. Dewanto: “Autorisierungsdienste mit OAuth” - heise developer blog article http://www.heise.de/developer/artikel/Szenarien-und-Spezifikationen-845431.html, accessed January 11, 2012.

[DWR 11] Unkown author: “DWR: Easy AJAX for JAVA” http://directwebremoting.org/dwr/introduction/index.html, accessed January 11, 2012.

[Fielding 08] R. T. Fielding: “BitWorking” Blog by J. Gregorio - “Code on Demand, REST, and Cloud Computing” - article comments http://bitworking.org/news/355/code-on-demand-rest-and-cloud-computing, accessed January 11, 2012.

[Foraker Labs 11] Unknown author: “Usability glossary” - Foraker Labs, UsabilityFirst http://www.usabilityfirst.com/glossary/affordance, accessed January 11, 2012.

[Google, Inc. 09] Unknown author: Google Inc., Jaiku - Jaiku blog http://jaiku.jaiku.com/presence/108e94b3953b48f5bb01bb4b6239396f, accessed January 11, 2012.

[Google, Inc. 11] Unknown author: Google Inc., Jaiku - API/´Documentation http://www.jaiku.com/api/docs, accessed January 11, 2012.

[Google, Inc. 11] Unknown author: Google Inc., Google Maps FAQ - API key http://code.google.com/apis/maps/faq.html#keysystem, accessed January 11, 2012.

[Jones 07] C. Jones Lorem ipsum http://www.loremipsum.de, accessed January 11, 2012.

[Laguna 09] Rubén Laguna: Better JMeter Graphs http://www.rubenlaguna.com/wp/better-jmeter-graphs/#comment-65735931, accessed January 11, 2012.

[Oracle Corporation 10] Oracle Corporation Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html, accessed January 11, 2012.

[Sun Microsystems, Inc. 03] Sun Microsystems Inc. jGuru: JavaServer Pages Fundamentals, Short Course Contents http://java.sun.com/developer/onlineTraining/JSPIntro/contents.html, accessed January 11, 2012.

[Twitter, Inc. 11] Unknown author: Twitter Inc., Twitter API Documentation https://dev.twitter.com/docs/api, accessed January 11, 2012.

[VisualVM 11] Unknown author: VisualVM 1.3.2 - Features http://visualvm.java.net/, accessed January 11, 2012.

[WordPress.com 11] Unknown author: WordPress.com, API Keys http://en.wordpress.com/api-keys/, accessed January 11, 2012.

[Yahoo!, Inc. 11] Unknown author: flickr - Der App Garden, API-Schlüssel http://www.flickr.com/services/api/misc.api_keys.html, accessed January 11, 2012.