



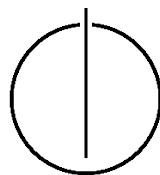
FAKULTÄT FÜR INFORMATIK

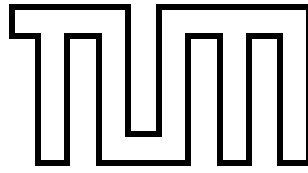
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

**Architectural Design and Implementation of a
Web Application for Adaptive Data Models**

Stefan Bleibinhaus





FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

Architectural Design and Implementation of a Web
Application for Adaptive Data Models

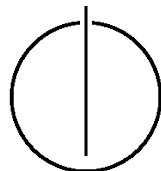
Architektur Design und Implementierung einer Web
Anwendung für adaptive Datenmodelle

Author: Stefan Bleibinhaus

Supervisor: Prof. Florian Matthes

Advisor: Matheus Hauder

Date: April 15, 2013



Ich versichere, dass ich diese Masterarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I assure the single handed composition of this master thesis only supported by declared resources.

München, den 15. April 2013

Stefan Bleibinhaus

Acknowledgments

I would like to express my very great appreciation to Prof. Florian Matthes for offering me to write my thesis on such a delightful topic and showing so much interest in my work. I am particularly grateful for the assistance given by Matheus Hauder and his will to support me in my research.

Abstract

This thesis discusses the architectural design and implementation of an Enterprise 2.0 collaboration web application. The designed web application uses the concept of hybrid wikis for enabling business users to capture easily content in structured form. A Hybrid wiki is a wiki, which empowers business users to incrementally structure and classify content objects without the struggle of being enforced to use strict information structures. The emergent information structure in a hybrid wiki evolves in daily use by the interaction with its users. Whenever a user wants to extend the content, the system guides them to automatically structure it by using user interface friendly methods like auto-completion and unobtrusive suggestions based on previous similar content.

An application of hybrid wikis is seen in the Wiki4EAM community, which aims to bring together practitioners and researchers to cooperatively pursue a different lightweight approach to Enterprise Architecture Management. In this community, architectural information, such as business processes, applications and organizational units is collected, structured, visualized and analyzed using the hybrid wiki Tricia.

The thesis starts with introducing the concept of hybrid wikis in more detail and presents Tricia, which is a hybrid wiki developed by the company infoAsset. Tricia's featureset is the guideline and motivation for an own prototype implementation called Trista, which is designed and implemented in this thesis. The chosen technologies used for its implementation are discussed. Among them is Scala, the multi-paradigm programming language built on top of the Java virtual machine (JVM), the play! web framework, the document-oriented database MongoDB and the HDFS (Hadoop Distributed File System) of Apache Hadoop. The goal of Trista is to be a lightweight hybrid wiki, which is horizontal scalable and provides a high useability. Trista provides an easy access to developers by its key usage of open source technology and is an highly extensible platform enabling rapid development. The thesis closes with an overview of Trista's implemented features and shows possibilities on how Trista can be extended.

Contents

Acknowledgements	vii
Abstract	ix
I. Overview	1
1. Introduction	3
2. Hybrid wikis	5
2.1. Motivation	5
2.2. Principles	6
2.3. Core concepts	8
2.3.1. Spaces	8
2.3.2. Content	9
2.3.3. Type	11
2.3.4. Attributes	11
2.3.5. Access control	12
2.3.6. Dashboard and individualization	13
2.4. Emergent information structures	14
2.4.1. Evolution of emergent information structures	14
2.4.2. Techniques facilitating information structuring	15
2.4.3. The soft constraint concept	16
2.5. Alternative interpretations of the hybrid wiki concept	16
3. The hybrid wiki Tricia	19
3.1. Introduction to Tricia	20
3.2. User interface of Tricia	21
3.3. Architecture of Tricia	23
3.3.1. Obstructions in development	23
3.3.2. Technological straits	24
3.3.3. Structural adversities	25
4. Trista requirements elicitation	29
4.1. Functional requirements	29
4.2. Non-functional requirements	33

II. Architectural Design	35
5. Scala	37
5.1. Introduction to Scala	37
5.1.1. Scala introductory examples	37
5.1.2. Scala features overview	40
5.2. Scala in comparison to Java	45
5.3. Scala as domain specific language for the web	48
6. Description of used Technologies and Frameworks	51
6.1. Architectural overview of Trista	51
6.2. The play! web framework	52
6.3. The MongoDB database	54
6.4. The Hadoop Distributed File System	55
III. Trista Implementation	57
7. Analysis	59
7.1. Hybrid wiki objects	59
7.2. Web view elements	60
8. System Design	63
9. Object Design and Implementation	65
9.1. Hybrid wiki object implementation	65
9.2. Database access	66
9.3. File handling	67
IV. Evaluation	69
10. Achievements in Trista	71
11. Conclusion	73
11.1. Summary	73
11.2. Outlook	74
Bibliography	77

Part I.

Overview

1. Introduction

The term software architecture was first used in the late 1960s. Since then, not just the definition but also the understanding of good software architecture changes and its quality has to be evaluated with regard to its requirements [Bass et al., 2013]. In the 1980s there was a breakthrough with structured analysis and design [Varhol, 1993], while in the 1990s object-oriented programming changed the landscape once again to face more increasingly complex software problems [Mandrioli and Meyer, 1992]. Beginning in 2000 with the increased popularity of the internet, solutions for new problems had to be found [Rosenfeld and Morville, 2002].

This thesis concentrates on the architectural design of a web application for collaboration using adaptive data models. A web application has its own architectural challenges [Schmidt et al., 2001]. It is supposed to run on a server with many different users accessing it at the same time. Therefore one of the major challenges of the architecture is to support responsiveness when confronted with a lot of requests at the same time. For example, the micro-blogging service Twitter has to be able to serve up to 6939 tweets per second [Twitter, 2013]. Therefore it is a crucial point for web services to be able to scale up. According to [Schmidt et al., 2001], scalability can also be one of the advantages of having a networked architecture. Other advantages include '*Collaboration and connectivity*', '*Enhanced performance, scalability, and fault tolerance*' and '*Cost effectiveness*'. For the web application designed in this thesis we pursue to take benefit of all of the mentioned advantages with a special interest in scalability. According to [Henderson, 2006] a scalable system has three characteristics: It can accommodate increased usage, it can accommodate an increased dataset and it is maintainable. The approach to reach scalability taken by us is horizontal scalability which differs essentially from vertical scalability. When the system provides vertical scalability, a replacement of the current hardware by more powerful hardware will make the system faster and more responsive. In horizontal scalability this is accomplished by adding more hardware and not necessarily replacing the old one. In horizontal scaling, the system is able to make use of more hardware by being able to distribute requests and/or calculations among the set up servers. The efficiency of scaling vertical is limited by the hardware capabilities of the current available technology while horizontal scalability is not limited [Henderson, 2006].

The designed web application of this thesis is called Trista. Trista is a hybrid wiki based on Tricia, which is a hybrid wiki developed by the company infoAsset¹. A Hybrid wiki is a wiki, which empowers business users to incrementally structure and classify content objects without the struggle of being enforced to use strict information structures. The emergent

¹<http://www.infoasset.de/>

information structure in a hybrid wiki evolves in daily use by the interaction with its users. Whenever a user wants to extend the content, the system guides them to automatically structure it by using user interface friendly methods like auto-completion and unobtrusive suggestions based on previous similar content. In practice this is achieved by adding a type and attributes to a page of a wiki. With the help of this user given information, the hybrid wiki is then able to structure the content as good as possible.

Tricia has grown over the years and lacks at the time of writing this thesis in scalability but also in usability. The user interface is outmoded and does not appeal anymore to users used to the modern web. The development team of Tricia also recognized this and is currently working on a redesign of the user interface for Tricia version 3.4. The architecture as well as the implementation of Tricia suffer under major flaws like the absence of strong third-party libraries, which could have helped the project for a better architecture and decreased maintenance effort.

We therefore decided to not migrate Tricia to a new software architecture, but to design a simple new architecture, which tackles this problems. Trista is scalable, performant and allows for faster development of future features by being up to date to current software engineering principles. Its simple design approach allows for additions and refactoring as suggested by agile methods [Beck, 2003] without requiring a lot of maintenance [Reussner and Hasselbring, 2006].

The technologies utilized to accomplish the aforementioned goals are the play! framework², Scala³, MongoDB⁴ and Hadoop⁵. The play! framework is an open-source web-framework for Java and Scala which focuses on high scalability and ease of development. It includes many convenient libraries among them Akka, an event-driven middleware which allows high concurrency with an actor-system. Scala is a multi-paradigm programming language running on the JVM designed as a "better Java". The language is functional, object-oriented and imperative. Its name derives from marketing itself as a scalable language, which is built with concurrency in mind. The database of Trista is powered by MongoDB, which is a scalable, high-performance, open-source NoSQL database. It is horizontal scalable through sharding, it allows high availability through replication and is a document-oriented storage. Hadoop contains the Hadoop Distributed File System (HDFS), which is a distributed, scalable and portable file system written in Java for the Hadoop framework. Trista utilizes HDFS for having a high-available and scalable file system, where files of users can be stored into.

²<http://www.playframework.com/>

³<http://www.scala-lang.org/>

⁴<http://www.mongodb.org/>

⁵<http://hadoop.apache.org/>

2. Hybrid wikis

This chapter describes the approach of hybrid wikis, which deals with the problems and situation described in *2.1 Motivation*. It follows up with the section *2.2 Principles*, which explains the three basic principles of hybrid wikis: Simplicity over expressivity, data over schema and evolution over rigidity. That section is followed by *2.3 Core concepts*, which explains the core concepts of hybrid wikis supporting emergent information structures in Enterprise 2.0 platforms. The next section *2.4 Emergent information structures* describes, how this concepts and models can be integrated in an Enterprise 2.0 platform. It also shows, how users can interact with these techniques. The last section *2.5 Alternative interpretations of the hybrid wiki concept* discusses the differences between the interpretation of the hybrid wiki concept taken in this thesis and the one in the dissertation by Christian Neubert [Neubert, 2013]. The aforementioned dissertation serves as the basis theory for hybrid wikis as described in this chapter.

2.1. Motivation

To avoid having their information distributed among emails, files and paper documents, companies started to use Enterprise 2.0 platforms. "Enterprise 2.0 is the use of emergent social-software platforms within companies, or between companies and their partners or customers" [Mcafee, 2006]. These platforms are being used now as lightweight shared content repositories which allows users to deal with information in a collaborative manner [McAfee, 2009]. This technology has according to [Mcafee, 2006] the power to let an Intranet for a company become what the Internet already is.

An important aspect of the information shared among these platform is its free-nature. Those platforms are often wikis in which the information is stored as plain textual, unstructured and user-generated content [O'Reilly, 2013]. While this form is fine for the user to add data, its unstructured form makes it also hard to search for content or evaluate big amounts of content in an analytical way. One common way to deal with this problem is it to use semantic technologies [Fensel, 2011] which have been integrated in Enterprise 2.0 applications [Passant, 2011]. They enable among other things the combination of text with structured data [Krötzsch et al., 2006]. The structured elements can then be queried and processed similar to the content of a database.

This approach however leads to some struggles applied in practice. IT experts have often to pre-configure and adapt structures in order to solve rather specific problems

[Ghidini et al., 2009], what often leads to business-IT communication problems [Boehm, 1991]. According to [Neubert, 2013] these often encountered obstacles are:

- Mismatch between demand and realization: The structures delivered do not fulfill the requirements.
- Latency between demand and release: The structures are not available when they are required.
- Cold start after schema migration: No data is provided when the changed structures are available.

Christian Neubert solves this problem by developing the concept of hybrid wikis, which is explained in more detail in the following sections.

2.2. Principles

This section presents the principles of hybrid wikis according to [Neubert, 2013]. The principles have to be understood as design guidelines developed to support the underlying goals of hybrid wikis. The main objective of hybrid wikis is it to facilitate emergent and adaptive information structures in Enterprise 2.0 platforms. In particular, hybrid wikis are built to encourage business users to provide structures and empower them to manage structures without the help of IT specialists.

Simplicity over expressivity

As the target group of hybrid wikis are business users, hybrid wikis aim to comfort them by a simple approach to adaptive information structures rather than confusing them with overly complicated processes and decisions. After all, hybrid wikis have to be seen as a tool, which aims to support its users and not as an artifact, which purpose of existence is solely justified by company policies. This implies that for the use of hybrid wikis, no special competencies of the user should be required. Especially the keyword-like annotations found in other approaches as semantic wikis have to be avoided.

The whole concept of hybrid wikis is built on the approach to start with a lightweight set of data and then let the system figure out ways to offer features that usually require a strict defined data model. According to [Neubert, 2013] this principle implies:

- A reduced set of structuring concepts instead of universal description languages.
- Default views to access structures instead of manual query-view configurations.

- Implicit management of information structures by means of views familiar to the users instead of formal and textual annotations.

Data over schema

The data over schema principle is devised on the assumption, that it is better to have invalid or inconsistent data than having no data at all. Which is especially the case when it comes down to user provided data. As hybrid wikis have their own approach in processing data, which is described in more detail in section 2.4 *Emergent information structures*, the system itself tries to build up temporary patterns on any kind of data given.

Furthermore, hybrid wikis functionality to suggest new attributes for certain types of content and auto-complete mechanisms, motivate the user to follow a discovered schema without disturbing them in their will to provide data to the system. Hybrid wikis may be set up, so that there can be given soft constraints to attributes. Soft constraints do not strictly enforce their given rule, but hint the user, when data given might not be applicable to the constraint. The concept of soft constraints is described in more detail in section 2.4.3 *The soft constraint concept*. According to [Neubert, 2013] this principle implies:

- Bottom-up, data-driven schema emergence instead of top-down meta modeling.
- Data first, schema second.
- Data gardening instead of data entry prevention.

Evolution over rigidity

As the previous principle, evolution over rigidity is also a principle affecting data, more precisely the user interaction with data in a hybrid wiki. Hybrid wikis focus on continuously evolving structure rather than a rigid defined data model. This way, a hybrid wiki engenders a vivid information model, which will change over time, when the user provides more data. This way, there is never need for migration of data to a new defined data model. Section 2.4 *Emergent information structures* describes in more detail, what is meant by the evolution of data. According to [Neubert, 2013] this principle implies:

- A vivid schema instead of rigid typing.
- Incremental structural adaption instead of large migration steps.
- Data migration on demand instead of data migration as obligation.

2.3. Core concepts

In this section, the core concepts of hybrid wikis are introduced. Those are derived from the principles described in the previous section 2.2 *Principles*. Its major elements are step-by-step defined and it is explained, how they work in relation to each other. The concepts described in this chapter will be the design template for the hybrid wiki developed in this thesis, which is named Trista. The core model of hybrid wikis with its major components is shown in figure 2.1.

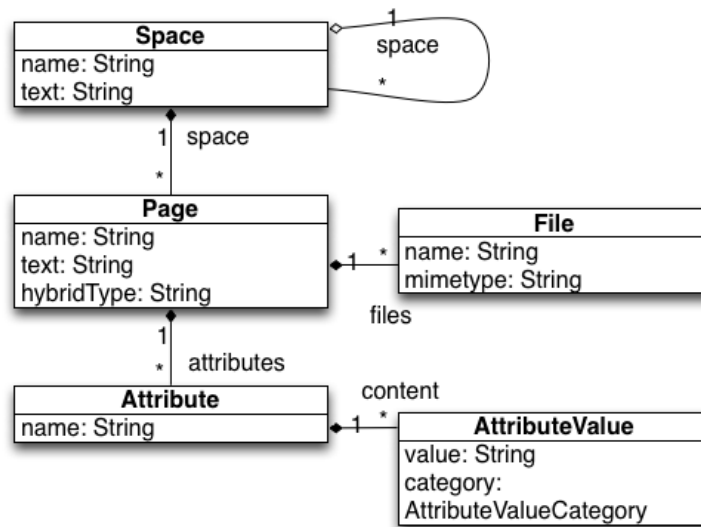


Figure 2.1.: An UML class diagram showing the model of hybrid wikis.

The main elements of the figure are spaces, content, types, attributes and attribute values. In a short summary, spaces serve as containers, which host other spaces or content. Content is either pages or files and are mainly described by their type and attributes. Those attributes have attribute values. Attribute values have a category to specify the value. Files are attached to pages. The following subsections describe those elements in more detail.

The theory on hybrid wikis of this chapter is based on the research done in the dissertation by Neubert [Neubert, 2013]. It differs in some aspects, which have been modified during the development of Trista and reflects feedback given by hybrid wiki users as well as some own considerations. The differences are shown and discussed in the section 2.5 *Alternative interpretations of the hybrid wiki concept*.

2.3.1. Spaces

Spaces in hybrid wikis serve as containers, which can host other spaces or content. They are identified by their name and the space, they themselves are contained in. The name is given

by the user, who creates the space. It can be changed later. Another important attribute of spaces are its text, which is editable and is shown, when a space is displayed. In Trista, that text is enriched by html-attributes, so that it can display media like photos or videos as part of the text. Figure 2.2 shows a screenshot of a space view in the hybrid wiki Trista.

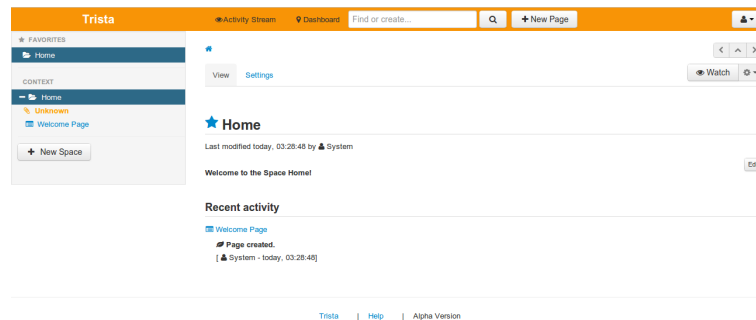


Figure 2.2.: A space displayed in Trista.

Spaces are used to order the tree-structured data. The underlying design pattern of the space concept in hybrid wikis is the composite pattern [Gamma et al., 2001], in which a composite component has an arbitrary amount of children, which are components on their own and even can be other composite components or leaf components. Leaf components have no children themselves. Applied to hybrid wikis, the composite component are spaces, whereas content are leaf components. Every hybrid wiki has at least one root space, in which all other content is put in. The root space is the only space in a hybrid wiki, which has no parent space. At the creation of a space, the parent space must be specified. A space can be moved into another space by changing its parent space. The root space can not be moved.

One purpose of spaces is, that they provide the main option to structure data. Similar data of the domain should be put in the same space. To support this usage, the web interface of Trista is built in a way, that the user can use spaces to navigate through the hybrid wiki. To maintain a neat structure inside the hybrid wiki, sub spaces might be used, when a space has too much content as direct children.

In Trista, spaces also show the recent changes of their direct children in the space view. This way, the user has an easy way to find out the newest development inside of this space.

2.3.2. Content

Content is another important component of hybrid wikis. Content items are a component of a space. A content item is identified by its name and the containing space. Content furthermore must have a hybrid type and may have attributes, which are explained in more detail in subsection 2.3.3 *Type* and 2.3.4 *Attributes*. In short summary, the hybrid type is used to identify, which kind of data is to be found inside of the content item. The hybrid type is beside spaces an additional way to structure data. The hybrid type is also used to identify,

2. Hybrid wikis

which attributes this content might have, as same types tend to have similar attributes (see 2.4 *Emergent information structures*). Content is used as the major container of data inside hybrid wikis.

In the hybrid wiki Trista, there are two types of content, pages and files, which are explained on their own in the next paragraphs. Additionally to them, there could be thought of extensions to Trista with more content types. Examples for possible additional content types are videos, audio files, surveys and diagrams.

Pages

Pages are content, which are mainly used to store associated text. Therefore pages have an attribute called text, which contains text associated with that page. The text is shown when displaying the page and can be modified by the user. In Trista, that text is enriched by html-attributes, so that it can display media like photos or videos as part of the text. Figure 2.3 shows the view of a page inside of Trista.

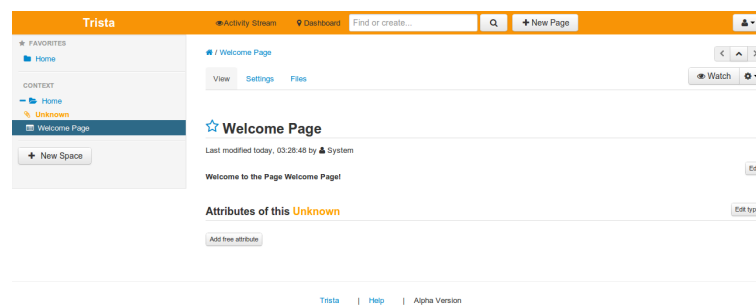


Figure 2.3.: A page displayed in Trista.

Files

Files are beside pages another type of content. Additionally, files are not only attached to spaces, but also to a hosting page. This has been designed this way, so that files are not floating inside of spaces, but have a context with the help of its hosting page. In Trista, files can be of any type or size and are accessible to the user on the associated page. Figure 2.4 shows the view of files associated with a page inside of Trista.

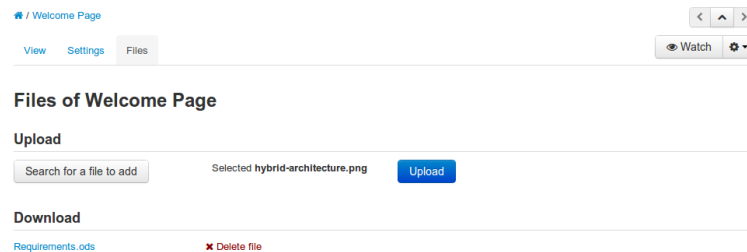


Figure 2.4.: The files tab of a page shown in Trista.

2.3.3. Type

Hybrid types are used to specify the category of content inside of hybrid wikis. Every content item of a hybrid wiki must have exactly one type. A type is usually given by the user as a string, when the content is created. The type can be changed later by the user. Beside the containing space of content, types are an additional option to structure the data in form of content inside the hybrid wiki. Content can also be found by searching for the type. The type plays a crucial role for the development of emergent information structures, which is in detail described in section 2.4 *Emergent information structures*.

As content is not just structured by its parent space, but also by its type, the existing types of a space are shown in Trista, when a space is opened. Figure 2.5 shows the listened types of an opened space.

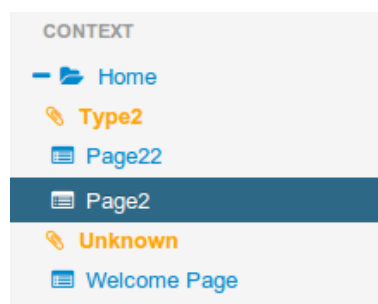


Figure 2.5.: Types of an opened space as shown in the context menu in Trista.

2.3.4. Attributes

Beside a hybrid type, content may also have attributes. Attributes are used to define the content apart from its name and type even more. A content item may have an indefinite amount of attributes. Each attribute has a key and a value. The key must be a non-empty string, whereas the value can consist of an indefinite amount of values. The values of an attribute may be strings, but may also be links to other spaces or content items within the hybrid wiki (internal links). The value of an attribute may have values of both categories.

2. Hybrid wikis

Figure 2.6 shows some attributes of a page in Trista. According to [Neubert, 2013], the attribute values inside of hybrid wikis may have the categories string, date, number, hypertext, record, external link, internal link or structured link, which are all supported by the hybrid wiki Tricia. At the current state of implementation, Trista supports strings and internal links as attribute values.

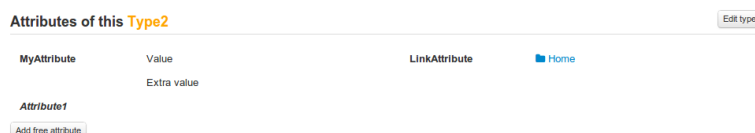


Figure 2.6.: Attributes of a page in Trista.

Attributes of content in hybrid wikis are similar in its feature to semantic wikis, where an underlying model of knowledge is used to describe the pages. However, in a hybrid wiki, the attributes are less strict than in semantic wikis. In a hybrid wiki, content may have attributes, but it is not forced to. The value of attributes in hybrid wikis may be checked according to rules given by the user, but values will not be refused, when they do not meet the rules specified. This concept is called soft constraints and is described in more detail in section 2.4.3 *The soft constraint concept*.

Attributes and their value may be used to search for content inside the hybrid wiki. They may also be used to order the search results according to the fact, if they have the attribute, or to the order of the attribute value. E.g. when searching for pages of the type *person*, these pages might have the attribute *age*. The found list of pages may now be sorted by the age of the persons in the list.

When attributes are used to link to other spaces and pages, a hybrid wiki shows the incoming links on the linked objects. Figure 2.7 shows how this looks on a page, which has an incoming reference by an attribute of another page. This feature is helpful, because relationships between content items might be bilateral in the given domain.

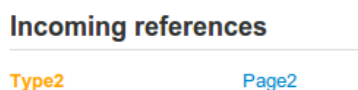


Figure 2.7.: Incoming references of a page in Trista.

2.3.5. Access control

Access control in hybrid wikis is not too much different from access control in standard wikis and follows the defaults of the underlying enterprise architecture. The access control is established by groups and users. Users and groups have names, which can be modified. Not logged in users are a special kind of users. All other users belong to the special group *users*, which is used to control access of all logged in users. Groups and users can be given

the rights read and write. New groups can be created and may contain an indefinite amount of users. Users can be added and deleted from a group. Groups can also be deleted. Groups must not contain other groups, but may contain all users of another group.

2.3.6. Dashboard and individualization

Hybrid wikis have features to individualize the experience for the user. Some users might have specific needs or might be interested in just a particular part of the data. As one of the goal of hybrid wikis is it to aggregate information and deliver the requested information as convenient as possible for the user, individualization features help to satisfy the needs of those users. It is a requirement for those features, that the user is logged in.

Favorites

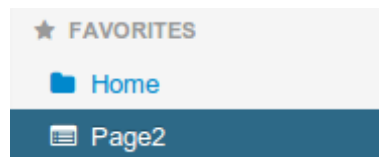


Figure 2.8.: The favorites menu of a user in Trista.

A user has favorites for quick access. Spaces and pages can be added to the favorites of an user by the user. Those favorites are quickly accessible in the navigation menu of the hybrid wiki. Figure 2.8 shows a list of favorites of an user in Trista. In Trista, spaces and pages can be added to the favorites by clicking the star next to the title of the item. On the same way, a previously added item can be removed from the list.

Watchlist

A user also has a watchlist for items, which changes are of special interest to them. A watchlist may contain an indefinite amount of spaces and pages. When showing the dashboard of the user, recent changes of items in the watchlist will show up. The list of recent changes is ordered by the time of the event beginning with the most recent change.

Dashboard

The dashboard is the personalized home of the user in the hybrid wiki. It shows information, which is of interest for the user. In Trista, it shows the watchlist of the user, which shows recent changes of items in the user's watchlist. Figure 2.9 shows a dashboard of a user in Trista with a filled watchlist and some recent changes.


Activity Stream

for  1


Recent activities in Trista


 Page2

 Page modified.


[ 1 - today, 03:37:04]


 Page22

 Page modified.


[ 1 - today, 03:34:33]

 Welcome Page

 Page modified.

[ 1 - today, 03:32:14]

 Space2

 Space created.


[ 1 - today, 03:38:53]

Figure 2.9.: A dashboard of a user in Trista.

2.4. Emergent information structures

This section describes how the emergent information structures inside the hybrid wiki evolve. The section gives insight into how hybrid wikis react to new user provided data and how this new data is processed with the knowledge of the already given data. It furthermore shows how users are guided to structure their input without being disturbed in their workflow. The main aspect here relies on the content data and its type as well as its attributes. The last subsection gives insight how soft constraint mechanisms may be implemented in a hybrid wiki.

2.4.1. Evolution of emergent information structures

In a hybrid wiki, data is more important than schema. Users of the hybrid wiki should not be bothered to care about or plan strict data models. The system of the hybrid wiki maintains a vivid data model which adapts itself with every new piece of data given. There is no need for migration of data inside of a hybrid wiki. The data of hybrid wikis is mainly found inside its content items, which contain the important attributes type and attributes. The text attribute, which can also be found in content items as well as spaces is plain text and therefore contains no kind of data model. The underlying concept of the vivid data model in hybrid wikis is based on the assumption that content with the same type is likely to share the same attribute keys and content items with the same type are likely to be found in the same space.

Based on this assumption, the hybrid wikis data model orientates on the the mathematical

intersection of attribute keys among all content of the same type. The mathematical intersection of two sets is the set of all elements which are members of all considered sets [Devlin, 1993]. Applied to the attribute keys of hybrid wikis and their hosting content items type, this means that when there exists a set of attribute keys, which are found in all content items of one specific type, this set of attribute keys is then considered to be part of the temporary schema for this content type. If there is no such set of attribute keys or some attribute key is included in most content items of that type but not in all of them, this set of attribute keys for the temporary schema might be created from a subset of content items with a specific type. In this case, content items which share this type might be considered as violating this temporary schema. In an actively used evolving hybrid wiki, it is likely that those content items might get the missing attribute keys within the next data updates. Because this case is so likely, the hybrid wiki suggests those attribute keys to the user in such cases. This process of suggestion is described in more detail in the next section.

2.4.2. Techniques facilitating information structuring

As one of the goals of hybrid wikis is usability, the system strives to assist the user in their workflow wherever it is possible to. Therefore the technique of suggestions and auto completion is used widely among hybrid wikis. The technique of auto completion is the ability of the web interface to suggest most likely input or input completion, whenever the user is about to enter data. In hybrid wikis, auto completion is not solely used for the purpose of better workflow but also for suggestions, which urge the user to provide structures without forcing them. Those suggestions for the user are done for the attribute keys as well as for their values. Additionally to that, types and data types are suggested to the user. According to [Neubert, 2013] hybrid wikis help this way to:

- Reuse existing information structures
- Discover new information structures
- Indicate information structures for similar business demands
- Unify terms and information structures

The suggestions for the attribute keys are based on the assumption mentioned in the last subsection, which states, that content with the same type is likely to share the same attribute keys. In Trista, when a content item like a page is viewed, the system checks for content with the same type and creates a list of attribute keys, which are part of that content items. This list of attribute keys is then sorted by their occurrence among the content items with the same type. The most occurred attribute keys, which are not part of the displayed content item yet are then suggested to the user at the content view. Figure 2.10 shows some suggestions for new attributes on the page view in Trista.

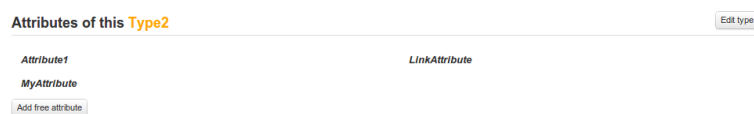


Figure 2.10.: Suggested attribute keys in cursive letters for a page in Trista.

Another suggestion includes the type of content. The goal of a hybrid wiki is to have no clones of types inside the system. A clone of a type is the same domain type but with a different name. An example might be *state* and *federal state*. When both types are identical inside the domain, but have different names, the described logic in this section is not applicable although it would be desirable. Therefore identical types with different names should not be part of a hybrid wiki. To avoid that situation, a hybrid wiki suggests types for content, when creating or modifying content.

The next form of suggestions are those for attribute values. As attribute values have an attribute category, the category for the attribute might be guessed according to the users input. An example is that the attribute category number is guessed, when a user types in an integer. Another suggestion of the hybrid wiki may be made, when the name of space or page is entered as attribute value, because this is most likely to be intended as an internal link to that item.

2.4.3. The soft constraint concept

As constraints are against the data over schema principle, which favors invalid or inconsistent data over preventing users from data entry in order to satisfy constraints, there are no strict constraints in a hybrid wiki. Nevertheless in some cases it is wanted to give the user additional guidance by providing directives for the type data which is expected for certain data fields. Therefore hybrid wikis have the concept of soft constraints, which allow users to set up constraints, which are not strictly enforced throughout the system.

In Trista the concept of soft constraints is not implemented at this stage. According to [Neubert, 2013] hybrid wikis may define soft constraints to specify how many values an attribute should have. Another class of constraints might define which category an attribute value should be for a specific attribute key.

2.5. Alternative interpretations of the hybrid wiki concept

In this section, the theoretical differences of the concepts for the implementation of hybrid wikis between the dissertation of Christian Neubert [Neubert, 2013] and this thesis are described. Figure 2.11 shows the concept of hybrid wikis as researched by Christian Neubert. The differences between this concept and the concept described in this thesis can

be discovered when comparing that figure with figure 2.1, which is shown at the beginning of this chapter.

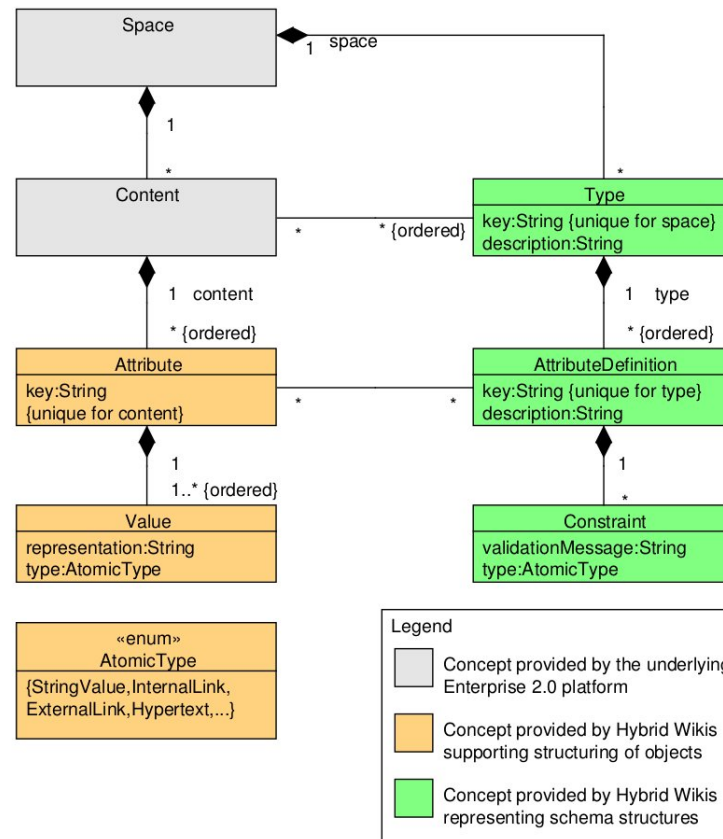


Figure 2.11.: An UML class diagram showing the concept of hybrid wikis as interpreted by Christian Neubert in his dissertation [Neubert, 2013]

The concept of Christian Neubert does not support subspaces. Subspaces are spaces which are contained in spaces. Therefore the arrangement of content is not as free as with subspaces. Because with subspaces, content can not just be sorted into one space, but in a tree-like structure which supports more information than the single parent space implementation.

Another major difference is, that in Christian Neubert's concept, types are bound to spaces and do not exist globally as in the concept presented in this thesis. The advantage of this approach is, that types might be used differently in each space. The disadvantage addresses the point described in 2.4.2 *Techniques facilitating information structuring*, in which the importance of avoiding duplicate types in a hybrid wiki is explained. Binding types to spaces requires the user to plan the spaces structure of the hybrid wiki more consciously to avoid the problem of duplicated hybrid types. This disadvantage is not given in the approach taken in this thesis.

2. Hybrid wikis

Content items in the concept of Christian Neubert are also allowed to have multiple types. The advantage of this approach is to give the user more control to distinguish between content items in the hybrid wiki. But on the other hand, it violates the principle of simplicity over expressivity by making the data structures in the hybrid wiki far more complex than they need to be. It encourages the user to add more hybrid types instead of assisting them to concentrate on the concrete range of types in the specific domain. It therefore adds complexity to the data model and requires the user to be more thoughtful with the input data.

In Christian Neubert's concept, attributes and their values are described more finer-grained with the help of attribute definitions and soft constraints. This concept is in more detail described in the 2.4.3 *The soft constraint concept* subsection of this thesis. The concept allows the user to bring more information into the hybrid wiki to guide other users to the currently desired data schema. The disadvantage is, that it complicates the system and makes it therefore less accessible to the business user.

3. The hybrid wiki Tricia

This chapter describes Tricia, which is a hybrid wiki developed by the company infoAsset¹. Tricia is the role model for Trista, the hybrid wiki developed in this thesis. The first section *3.1 Introduction to Tricia* gives an introduction to Tricia and its context of development. Then Tricia's drawbacks are discussed which are the main motivation for the development of Trista. The drawbacks of Tricia are divided into two sections. The first section is *3.2 User interface of Tricia* which describes obstacles found in the user interface of Tricia. The next section *3.3 Architecture of Tricia* then continues with the description of the impediments found in the codebase of Tricia. It also discusses general architectural design decisions which make Tricia not scalable and hinder faster development. In general the section about the user interface concentrates on flaws an end-user of Tricia might discover whereas the section about the architecture of Tricia describes which problems a developer in Tricia might face. It is important to note, that at the time of writing, Tricia 3.3 is the most recent and therefore discussed version in this thesis.

¹<http://www.infoasset.de/>

3.1. Introduction to Tricia

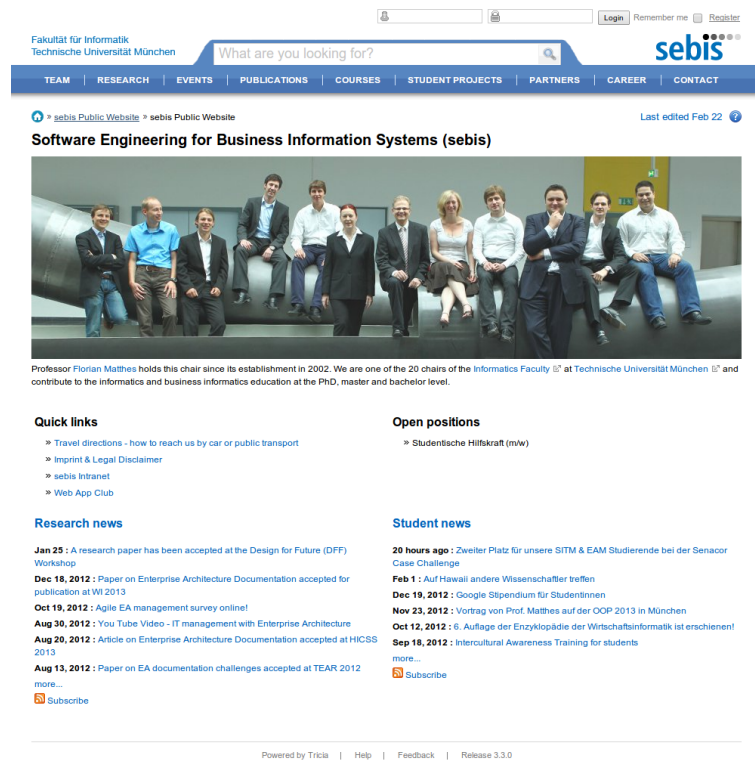


Figure 3.1.: Tricia in its current deployment as homepage of infoAsset.

This section presents the hybrid wiki Tricia, which is based on the research done by Christian Neubert in his dissertation [Neubert, 2013]. Tricia is a hybrid wiki following the three principles of hybrid wikis described in 2.2 *Principles*. Its core concepts are similar to those described in 2.3 *Core concepts* and are discussed in more detail in [Neubert, 2013]. The last section 2.5 *Alternative interpretations of the hybrid wiki concept* gave an overview of differences in its core concepts to the core concepts of this thesis. Figure 3.1 shows a screenshot of Tricia in its current deployment as homepage of infoAsset. Tricia is a feature rich hybrid wiki, which is daily used as a powerful tool by established companies². According to infoAsset³, the main advantages of the Tricia's system architecture are, that Tricia is:

- Little administrative overhead
- Robust
- Scalable and lightweight

²<http://www.infoasset.de/pages/11r1lagzrxo54/Partners>

³<http://www.infoasset.de/pages/11r0iafxcx9v6/Tricia-System-Architecture>

- Platform independent
- Secure
- Extensible and open

However, we identified a list of obstacles within the user interface and software architecture of Tricia. The next two sections will describe them.

3.2. User interface of Tricia

This section discusses current obstacles in Tricia from the point of the end-user. It is crucial to note here, that at the time of writing, Tricia 3.3 is the most recent and therefore discussed version in this thesis. In the next version Tricia 3.4 which includes a redesign of the user interface, the Tricia team itself will target some of the points mentioned in this section.

Overloaded options

The interface is in general overloaded and confuses new users. This is partly because of the many features Tricia supports, but also because less used options are not summarized. For example the options for editing, watching, browsing, copying, moving, deleting, sharing and printing of a page could be summarized in a drop-down menu.

Not scaling to screen size

Tricia uses a fixed width to display its content and therefore does not adapt to larger or smaller screens. On larger screens there is a lot of white space, which could be utilized but is left unused and on smaller screens, this means, that Tricia is harder to use. This is an especially important point with the recent development in technology namely the trend towards smaller devices and therefore smaller screens like tablet computers.

Browsing workspaces is not user friendly

Tricia uses tables to show the content of workspaces. Those tables are often overfilled due to the large amount of pages in it and often need to be displayed over more than one site. Browsing workspaces in Tricia is therefore not user friendly.

No Favorites

Tricia does not support the concept of favorites. It only allows to watch items for changes, but favorites for easy navigating to the most interesting items of a user is not supported in the current version of Tricia.

No context of the current viewed item

Tricia does not display the context of the current viewed item beside the breadcrumb which shows in which workspace the current page is. That means that in Tricia no other (similar) pages are shown the user might also be interested in when viewing a page in Tricia.

Confusing actions

The interface of Tricia is not just overloaded, but some of the links and actions are also confusing. Tricia does not make use of a lot of icons to familiarize the user with the expected outcome of a link. Therefore the user has to remember the wanted action by the name of its action. But some names for actions are even misleading or ambiguous. For example the click on the name of the current user leads to that users dashboard. An inexperienced user might not find the dashboard, because he tries to find a link named dashboard. On the other hand a new user might expect a link to their profile when clicking on their name.

URLs are not stable

When Tricia has been upgraded, some URLs are not valid anymore and point to an error page. Therefore it is hard to maintain bookmarks for a user in Tricia. Together with the missing option to add favorites, users might lose the grasp on for them interesting items.

No tree-structure for workspaces

Tricia does not allow the creation of workspaces inside of other workspaces. Therefore an important part to structure the hybrid wiki as described in section 2.3.1 *Spaces* is not possible.

No use of Ajax

Tricia misses the opportunity to make the interface faster for the user by integrating Ajax in some actions. Therefore the whole page is often loaded again, when only small parts of the

page change.

No emphasize on creating attributes

When a page in Tricia has currently no attributes assigned, the link for adding attributes to the page is small and gets lost in the shuffle of options. Therefore the emphasize on creating attributes, which is an important factor for a hybrid wiki is very low in Tricia.

3.3. Architecture of Tricia

This section discusses current obstacles in Tricia from the view point of a developer. Tricia's drawbacks in the current software architecture as well as its flaws in implementation are described here. The problems are categorized in 3.3.1 *Obstructions in development*, 3.3.2 *Technological straits* and 3.3.3 *Structural adversities*. It is important to note, that most points found could be put in more than one category. Those points are put into the category which is most affected by them. The category 3.3.1 *Obstructions in development* describes points which hinder faster development of Tricia, 3.3.2 *Technological straits* contains points where technological decisions of Tricia influence the project displeasingly and finally 3.3.3 *Structural adversities* includes the points in which the structure and architecture of Tricia have a bad effect to the project.

3.3.1. Obstructions in development

No standard project structure

Tricia does not use a standard project structure. This makes it very hard for new developers, who are not familiar with the project to start developing for the project. It also requires more attention of the software architect to the project.

No coding guidelines

The project Tricia has no coding guidelines. Missing guidelines for code means that the style of the code varies and is therefore harder to read and to maintain.

Code is not well documented

The codebase of Tricia is not well documented. Some parts of the code is not documented at all. This makes it harder to maintain the project and also harder for new developers.

Own frontend template language

Tricia uses its own template language for its frontend. It has been specifically designed for the use in Trica and is not used by any other project. The problem with this approach is, that everyone has to learn this template language first. There is no pool of developers available, which is already fluent in it. Besides that, there is no integrated development environment (IDE) available to support that custom language. Therefore suggestions and errors in the use of the template language will not be shown to the developer during development. Developers, which learn this template language do not directly benefit from the knowledge of it in future projects.

Project is dependent on specifics

Tricia is written in Java. However it does only support the Sun Java Development Kit (JDK) in version 6. It is not possible to build Tricia with Java's most recent version 7 or another implementation of the JDK (e.g. OpenJDK⁴).

3.3.2. Technological straits

Not scalable and low performance

Tricia is not horizontal scalable and suffers in general under low performance. The architecture of Tricia was not designed for horizontal scalability and it is very hard to migrate a big project like Tricia to a new scalable architecture. Tricia is also not performing well in large setups which results from the missing use of strong and performant third party libraries.

Complex and intransparent deployment process

Tricia's deployment process is overly complex and intransparent. The best practice for Java web applications is a single step deployment, in which the administrator is able to build a deployable web module (usually a WAR file) executing one command [Oracle, 2013].

⁴<http://openjdk.java.net/>

In Tricia a compressed zip-file is built with the help of an Ant-script⁵. This zip-file has then to be send to an integration server, which performs a couple of intransparent steps with the help of specific designed scripts before deploying it to the the target server. The disadvantage of this approach is that the developer is not able to deploy the application himself for quickly testing the deployed application.

Own database access implementation

The database layer as well as the code down to the access of the database is custom made. This makes it hard for new developers to get used to the codebase. It also increases the work to maintain the project a lot and makes it more vulnerable to errors. Furthermore, the performance of the database is shortened by neglecting the natural strengths of SQL databases. In Tricia, some fields in the database are objects on their own represented in JSON. There are also operations e.g. searches executed including those fields. A SQL database can not perform well on that kind of operations.

3.3.3. Structural adversities

Tests are scattered

Inside the code of Tricia, tests are scattered and partly mixed into packages with production code. This makes it very hard to maintain the tests, get an overview of how much tests are inside of the project and which parts of the code is tested. A consequence of this scattered test architecture is also that it is harder to maintain the prerequisites of the test environment, which are usually different from those used in production. In Tricia this can be seen by the fact, that it is currently not possible to see the logging output when running tests.

Libraries are committed to the CVS

The libraries used by Tricia are committed to its Concurrent Versions System (CVS), which is currently Mercurial. This is considered as bad practice because updating the libraries will leave the old versions inside the CVS and making it unnecessarily bigger and therefore slower.

Hardcoded values inside the code

Inside the codebase of the backend for Tricia are hardcoded values, which should be externalized to configuration files. Using configuration files makes it easier to configure the

⁵<http://ant.apache.org/>

project and adapt it to other needs. A consequence of this in Tricia is, that currently it is not possible to run the web application of Tricia and its database on different machines.

Code is not well tested

The codebase of Tricia is not well tested. This makes the project vulnerable to errors, especially when new features are being developed.

Contains unused code

The project contains unused code beside production code. Having unused code in the codebase makes the project harder to maintain and also hinders creating an overview of the project.

Project contains other small projects

Inside the codebase of Tricia and also beside production code, there are script-like small applications, which are not needed to run Tricia. These small projects should be externalized into their own sub-projects. Having them inside the main project makes it harder to maintain the codebase and unnecessarily blows up the size of the project.

Frontend code is mixed into the backend code

Some of the frontend logic is mixed into the backend code of Tricia. This is problematic because it makes the codebase harder to maintain. Beside that, frontend and backend developers are usually separated teams with own competencies.

Contains example code in production code packages

Tricia contains example code in the codebase beside packages with production code. This could lead to confusion among the developers. It also risks that the code mixes up with production code and results in errors in the product. It makes the project also harder to maintain and unnecessarily bigger.

Misuse of javadoc

The technology javadoc, which allows documenting Java code is misused in Tricia to place general documentation inside the code base. New developers will not expect project documentation inside the codebase.

Overly complex architecture

The architecture of Tricia is overly complex and has not been refactored faithfully when new features got integrated into the platform. This makes it harder to maintain the project as well as costlier to implement new features.

4. Trista requirements elicitation

The following chapter exposes the requirements found for Trista, which is the hybrid wiki developed in this thesis. Those requirements are based on the characteristics of a hybrid wiki explained in 2 *Hybrid wikis* as well as on the features of Tricia which are described in 3 *The hybrid wiki Tricia*. Trista's requirements are divided into 4.1 *Functional requirements* and 4.2 *Non-functional requirements*. The style of their description is based on the Volere Requirements Specification Template [Volere, 2013]. The requirements of Trista are essential for the next chapters, which concentrate on how to satisfy those requirements and implement them.

According to the requirements shell from Volere, each requirement is described by its attributes *Requirement Number, Description, Rationale, Originator, Fit Criterion, Dependencies* and *History*, which are used to describe the requirements of Trista. The attributes *Requirement Type, Event Number, Customer Satisfaction, Customer Dissatisfaction, Conflicts* and *Supporting Materials* are also part of the requirements shell from Volere but are left out in the following description of requirements for Trista because of their unsuitability. Additionally to the attributes from Volere, the attribute *Priority* has been added to the requirements shell used for Trista. The priority attribute of the requirement has one of the values low, medium or high.

4.1. Functional requirements

This section enumerates the functional requirements of Trista. Of the 16 functional requirements in Trista, 7 are of high, 6 are of medium and 3 are of low priority.

Trista functional requirement #1 - Spaces	
<i>Description</i>	The hybrid wiki shall have spaces
<i>Rationale</i>	To enable spaces contain other spaces and pages
<i>Priority</i>	High
<i>Originator</i>	Hybrid wiki theory
<i>Fit Criterion</i>	Spaces can be viewed and created
<i>Dependencies</i>	-
<i>History</i>	Created October 29, 2012

4. Trista requirements elicitation

Trista functional requirement #2 - Space Options	
<i>Description</i>	Spaces can be deleted and moved
<i>Rationale</i>	To enable easy deletion and moving of spaces
<i>Priority</i>	High
<i>Originator</i>	Matheus Hauder
<i>Fit Criterion</i>	Spaces can be deleted and moved with one click
<i>Dependencies</i>	#1
<i>History</i>	Created October 29, 2012

Trista functional requirement #3 - Space Settings	
<i>Description</i>	Spaces have a settings tab
<i>Rationale</i>	To enable displaying and modifying settings of spaces
<i>Priority</i>	Medium
<i>Originator</i>	Matheus Hauder
<i>Fit Criterion</i>	Spaces have a settings tab when shown
<i>Dependencies</i>	#1
<i>History</i>	Created October 29, 2012

Trista functional requirement #4 - Pages	
<i>Description</i>	The hybrid wiki shall have pages
<i>Rationale</i>	To enable pages containing attributes and information
<i>Priority</i>	High
<i>Originator</i>	Hybrid wiki theory
<i>Fit Criterion</i>	Pages can be viewed and created, they have editable attributes
<i>Dependencies</i>	-
<i>History</i>	Created October 29, 2012

Trista functional requirement #5 - Page Options	
<i>Description</i>	Pages can be copied, deleted and moved
<i>Rationale</i>	To enable easy copying, deletion and moving of pages
<i>Priority</i>	High
<i>Originator</i>	Matheus Hauder
<i>Fit Criterion</i>	Pages can be copied, deleted and moved with one click
<i>Dependencies</i>	#4
<i>History</i>	Created October 29, 2012

Trista functional requirement #6 - Page Settings	
<i>Description</i>	Pages have a settings tab
<i>Rationale</i>	To enable displaying and modifying settings of pages
<i>Priority</i>	Medium
<i>Originator</i>	Matheus Hauder
<i>Fit Criterion</i>	Pages have a settings tab when shown
<i>Dependencies</i>	#4
<i>History</i>	Created October 29, 2012

Trista functional requirement #7 - File Handling	
<i>Description</i>	Files can be uploaded and downloaded to the platform
<i>Rationale</i>	To enable managing files in the hybrid wiki
<i>Priority</i>	Medium
<i>Originator</i>	Matheus Hauder
<i>Fit Criterion</i>	Files can be uploaded, downloaded and are attached to pages
<i>Dependencies</i>	#4
<i>History</i>	Created October 29, 2012

Trista functional requirement #8 - Constraints	
<i>Description</i>	Attributes in pages may have constraints
<i>Rationale</i>	To enable specifying the kind of attribute
<i>Priority</i>	Low
<i>Originator</i>	Matheus Hauder
<i>Fit Criterion</i>	Constraints can be created and attached to attributes
<i>Dependencies</i>	#4
<i>History</i>	Created October 29, 2012

Trista functional requirement #9 - Versions	
<i>Description</i>	Objects in Trista are versioned
<i>Rationale</i>	To enable restoring and viewing history
<i>Priority</i>	Low
<i>Originator</i>	Matheus Hauder
<i>Fit Criterion</i>	The versions of objects in Trista are displayed
<i>Dependencies</i>	-
<i>History</i>	Created October 29, 2012

4. Trista requirements elicitation

Trista functional requirement #10 - Import and Export	
<i>Description</i>	Data of Trista can be imported and exported
<i>Rationale</i>	To enable simple migration and backups
<i>Priority</i>	Low
<i>Originator</i>	Matheus Hauder
<i>Fit Criterion</i>	Data of Trista can be imported and exported over the interface
<i>Dependencies</i>	-
<i>History</i>	Created October 29, 2012

Trista functional requirement #11 - Users	
<i>Description</i>	Trista supports users
<i>Rationale</i>	To enable authorship and individual options
<i>Priority</i>	High
<i>Originator</i>	Hybrid wiki theory
<i>Fit Criterion</i>	Users can register, login and logout
<i>Dependencies</i>	-
<i>History</i>	Created October 29, 2012

Trista functional requirement #12 - Access-Management	
<i>Description</i>	Users have rights to view and modify content
<i>Rationale</i>	To enable access restriction and privacy
<i>Priority</i>	Medium
<i>Originator</i>	Matheus Hauder
<i>Fit Criterion</i>	Content has attributes which allows to set the access rights for specific users and groups
<i>Dependencies</i>	#11
<i>History</i>	Created October 29, 2012

Trista functional requirement #13 - Watch	
<i>Description</i>	Content in Trista can be watched
<i>Rationale</i>	To enable a quick overview of updates
<i>Priority</i>	Medium
<i>Originator</i>	Matheus Hauder
<i>Fit Criterion</i>	Content can be added and removed from a personal watchlist, changed items on this list will be shown on the dashboard of the user
<i>Dependencies</i>	#11, #16
<i>History</i>	Created October 29, 2012

Trista functional requirement #14 - Favorites	
<i>Description</i>	Content in Trista can be added to favorites
<i>Rationale</i>	To enable a quick access of favorites
<i>Priority</i>	Medium
<i>Originator</i>	Matheus Hauder
<i>Fit Criterion</i>	Content can be added and removed from a personal favorites list, the list is displayed to the user
<i>Dependencies</i>	#11
<i>History</i>	Created October 29, 2012

Trista functional requirement #15 - Search	
<i>Description</i>	Content in Trista can be searched
<i>Rationale</i>	To enable the search for information
<i>Priority</i>	High
<i>Originator</i>	Hybrid wiki theory
<i>Fit Criterion</i>	A searchbar allows searching for objects in Trista by its name and content
<i>Dependencies</i>	-
<i>History</i>	Created October 29, 2012

Trista functional requirement #16 - Dashboard	
<i>Description</i>	A user in Trista has its own dashboard
<i>Rationale</i>	To enable showing user specific information
<i>Priority</i>	High
<i>Originator</i>	Matheus Hauder
<i>Fit Criterion</i>	A user has their own dashboard
<i>Dependencies</i>	#11
<i>History</i>	Created October 29, 2012

4.2. Non-functional requirements

This section lists the non-functional requirements of Trista. All 4 non functional requirements in Trista are of high priority. Their fit criterion is not strict due of the prototype nature of Trista. Their purpose is to ensure that design decisions and decisions for chosen technologies and frameworks are based on the desire to fulfill them.

4. Trista requirements elicitation

Trista non functional requirement #1 - Scalability	
<i>Description</i>	All parts of Trista are scalable
<i>Rationale</i>	To enable scaling up to a big amount of users
<i>Priority</i>	High
<i>Originator</i>	Matheus Hauder
<i>Fit Criterion</i>	All technical parts and frameworks used in Trista are scalable
<i>Dependencies</i>	-
<i>History</i>	Created October 29, 2012

Trista non functional requirement #2 - Performance	
<i>Description</i>	All actions in Trista resolve fast
<i>Rationale</i>	To enable a fluid and fast user experience
<i>Priority</i>	High
<i>Originator</i>	Matheus Hauder
<i>Fit Criterion</i>	All pages in Trista load in a reasonable amount of time
<i>Dependencies</i>	#1
<i>History</i>	Created October 29, 2012

Trista non functional requirement #3 - Usability	
<i>Description</i>	Trista is a hybrid wiki with high usability
<i>Rationale</i>	To enable a fluid and effective user experience
<i>Priority</i>	High
<i>Originator</i>	Matheus Hauder
<i>Fit Criterion</i>	Users of Tricia know instinctively how to use Trista
<i>Dependencies</i>	-
<i>History</i>	Created October 29, 2012

Trista non functional requirement #4 - Fostering development	
<i>Description</i>	Trista allows rapid development
<i>Rationale</i>	To enable quick and easy extensions
<i>Priority</i>	High
<i>Originator</i>	Matheus Hauder
<i>Fit Criterion</i>	Trista uses only standard technology
<i>Dependencies</i>	-
<i>History</i>	Created October 29, 2012

Part II.

Architectural Design

5. Scala

Scala is a multi-paradigm programming language built on top of the Java virtual machine (JVM)¹. Scala is the main programming language for Trista, the hybrid wiki developed in this thesis. The following chapter introduces Scala and explains its main attributes in 5.1 *Introduction to Scala*. Scala is then compared to Java in 5.2 *Scala in comparison to Java*. This thesis expects the reader to be familiar with the programming language Java, which has been specified in [Joy et al., 2000]. The last section of this chapter 5.3 *Scala as domain specific language for the web* explains why Scala has been chosen for Trista.

5.1. Introduction to Scala

Scala is a multi-paradigm programming language built on top of the Java virtual machine (JVM)¹. It fuses object-oriented and functional programming in a statically typed programming language. The goal of the language is to enable construction of software components and component systems in a scalable manner [Odersky et al., 2004a].

Scala has been designed starting in 2001 at the École Polytechnique Fédérale de Lausanne (EPFL)² by Martin Odersky. Its first release was in 2003 [Odersky, 2006]. Version 2.0, a second version of the language was released in 2006 [Odersky, 2008]. In 2011 the Scala team won a research grant of over 2.3 million Euro from the European Research Council [Scala, 2011b]. Later that year, the team launched Typesafe³, a well-founded company to provide commercial support, training, and services for Scala [Scala, 2011a].

5.1.1. Scala introductory examples

Listing 5.1 shows the typical hello world example in Scala. The singleton object `HelloWorld` extends the trait `App`. Upon creation of the object, the code inside is executed. This code consists of the `println` method, which is similar to Java's `System.out.println`. The argument which will be shown on the screen is the `Hello World!` string.

¹<http://www.java.com/en/>

²<http://epfl.ch/>

³<http://typesafe.com/>

Listing 5.1: Hello World in Scala

```
object HelloWorld extends App {  
  println("Hello World!") // prints Hello World!  
}
```

The next listing 5.2 shows a more complex example of Scala in action taken from [Odersky, 2011]. It is the quicksort algorithm [Cormen et al., 2001] implemented in functional and recursive manner as the method `sort`. It takes an array of integers as argument and returns a new array with the same elements but sorted ascending. The method `sort` is defined by the keyword `def`, its name, arguments and return type, which can often be omitted, because the compiler can infer it from the context. The keyword `val` defines a value (read only variable), whereas variables are defined by the keyword `var`. Array selections are written `a(i)`. The return value of a method in Scala is its last executed statement or expression.

Listing 5.2: Quicksort function of an array of integers in Scala

```
def sort(xs: Array[Int]): Array[Int] =  
  if (xs.length <= 1) xs  
  else {  
    val pivot = xs(xs.length / 2)  
    Array.concat(  
      sort(xs filter (pivot >)),  
      xs filter (pivot ==),  
      sort(xs filter (pivot <))  
    )  
  }
```

The logic of the `sort` method is as follows:

- If the array's length is equals or less than one, it is already sorted, return it. Else:
- Define a pivot element, which is the element in the middle of the array.
- Build the sorted array by concatenating three arrays: All elements sorted, which are less than the pivot element (1), all elements, which are equals to the pivot element (2) and all elements sorted, which are bigger than the pivot element (3).

The last step is the most interesting one and utilizes the functional nature of Scala: `xs filter (pivot >)` calls the method `filter`, which is a library method of a sequence in Scala, on the array `xs`. The method `filter` has the signature shown in listing 5.3.

Listing 5.3: filter method signature

```
def filter(p: T => Boolean): Array[T]
```

It takes a predicate function, which has to return a `Boolean` for each element of the array, and returns an array consisting of all the elements of the original array for which the given predicate function is true. Functions like `filter` that take another function as argument or return one as result are higher-order functions. In `xs filter (pivot >)` the method `filter` is called with the anonymous function `pivot >`. It is a partially applied function, because of the implicit argument. An explicit way to write this function would be `x => pivot > x`. The type of `x` is `Int` but is omitted, because Scala's compiler can infer it automatically from the context where the function is used. In summary `xs filter (pivot >)` returns an array of elements of the array `xs`, which are less than the value of `pivot`.

Listing 5.4 shows an example of how classes are implemented in Scala. In this example taken from [Odersky et al., 2011], the class `Rational` is implemented as immutable. Objects of `Rational` represent a rational number and have a numerator `numer` as well as a denominator `denom`.

Listing 5.4: Class `Rational` in Scala

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g
  def this(n: Int) = this(n, 1)
  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom)
  def * (that: Rational): Rational =
    new Rational(numer * that.numer, denom * that.denom)
  override def toString = numer + "/" + denom
  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

A rational number can be created by calling `new Rational(3, 4)`. As rational numbers are not allowed to have a negative denominator, the line `require(d != 0)`, which is executed with every creation of a new object hinders the creation with illegal values. On violation it throws a `IllegalArgumentException`. When the arguments are valid, the private value `g` is initialized, which is the greatest common divisor (gcd) between `n` and `d`. The gcd is calculated by the private method `gcd` and then used to initialize the (public) values `numer` and `denom`, which is the reduced fraction of the constructor arguments. Another way to create a `Rational` object is by calling `new Rational(4)` which just takes the numerator and implicitly sets the denominator to 1. This constructor is defined by the `this` method. After creation, `Rational` objects can be added and multiplied with each other as seen in listing 5.5. The method `println` implicitly calls the overridden method `toString` to convert a `Rational` object to a `String`.

Listing 5.5: Using `Rational` objects in Scala

```
val a = new Rational(3, 4)
val b = new Rational(4)
println(a + b) // prints 19/4
println(a * b) // prints 3/1
```

5.1.2. Scala features overview

This subsection gives an overview of selected basic features in Scala and how the language can be used. Scala is described in detail in [Odersky et al., 2011].

Syntactic flexibility

Scala has syntactic flexibility. It does not require semicolons. The compiler tries to join lines, when they end with a token that cannot normally come in this position. Method calls can be written simplified for certain cases. For example, listing 5.6 shows multiple options to call the same method. The syntactic flexibility of Scala enhances its readability.

Listing 5.6: Using `Rational` objects to show syntactic flexibility in Scala

```
val a = new Rational(3, 4)
val b = new Rational(4)

val sum1 = a.+(b)
val sum2 = a + b
println(sum1 + ", " + sum2) // prints 19/4, 19/4

val string1 = sum1.toString()
val string2 = sum1 toString ()
val string3 = sum1.toString
// prints 19/4, 19/4, 19/4:
println(string1 + ", " + string2 + ", " + string3)
```

Unified type system

In Scala, all types inherit from a top-level class `Any`, which has `AnyVal` and `AnyRef` as immediate children. `AnyVal` are value types like numbers and characters while `AnyRef` are referencing types like the values referencing a `Rational` object. The main advantage of the Scala Type system is its built-in richness. Figure 5.1 shows the hierarchy of Scala's types.

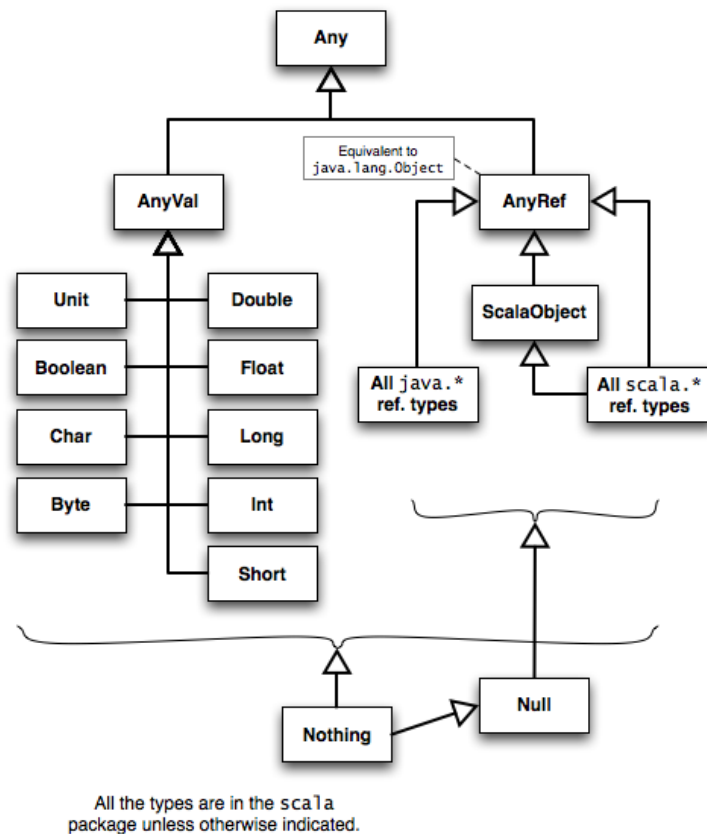


Figure 5.1.: Scala's type hierarchy [Wampler and Payne, 2009].

For-comprehensions

Scala has powerful for-loops. Scala's for loops allow to run over multiple sequences, making nested for-loops unnecessary. Additionally it is possible to filter the elements of a loop using the `if` keyword. Another feature is the keyword `yield` which sums up the executed expressions inside the for-loop to a new sequence. Listing 5.7 shows an example of a for-loop which iterates over two integer lists and sums up the results to a new list which is saved in the value `loop`.

Listing 5.7: for-loop example in Scala

```

// 2 + 1, 3 + 1, 3 + 2:
val loop = for {
  i <- List(1, 2, 3)
  j <- List(1, 2)
  if i > j
} yield i + j

println(loop) // prints List(3, 4, 5)

```

Functional tendencies

Scala is a multi-paradigm programming language. One of its features is enabling a functional programming style. In the last section *5.1.1 Scala introductory examples* in listing 5.2 the quicksort algorithm was implemented in a functional way. That example features type inference, anonymous functions with capturing semantics (i.e. closures), higher-order functions and nested functions.

Additional to the tendencies already seen, Scala allows lazy evaluation. Listing 5.8 shows an example of lazy values in Scala, whereas listing 5.9 shows an example of lazy collections, which can be made lazy with the `view()` method. Laziness in Scala allows the program to run faster, when the execution of the code is not needed.

Listing 5.8: Lazy value example in Scala

```
lazy val date = new Date

println(date) // First creation of the new Date
```

Listing 5.9: Lazy function example in Scala

```
def printFunction(run: Boolean, stringFunction: => String) {
  if (run) println(expensive)
}

// (0 to 1000000).toString is never called:
printFunction(false, (0 to 1000000).toString)
// (0 to 10).toString is called and used this time:
printFunction(true, (0 to 10).toString)
```

Scala also allows to define multiple parameter lists in functions. When a method is called with a fewer number of parameter lists, then this will yield a function taking the missing parameter lists as its arguments. This is called currying [Scala, 2008]. An example is given in listing 5.10. The example is inspired by [Odersky et al., 2011].

Listing 5.10: Currying in Scala

```
def curriedSum(x: Int)(y: Int) = x + y

println(curriedSum(1)(2)) // prints 3

def plusThree(a: Int) = curriedSum(3)

println(plusThree(1)) // prints 4
```

Scala has built-in support for pattern matching, which is similar to a switch-statement but by far more advantageous. In Scala's pattern matching each case expression is tried in turn to see if it will match and the first match determines the result. The quicksort algorithm as

seen in 5.1.1 *Scala introductory examples* can be rewritten using pattern matching as in listing 5.11. In this method, the argument `list` is matched against `Nil` and `pivot :: tail`. If the list is empty, the first case will trigger and return the empty list. The logic for the second case is as follows:

- The list will be divided in the `pivot` element, which is at the head of the list and the `tail`, which is the rest of the list (can be `Nil`).
- A tuple (`smaller, rest`) is created, which consists of a list of smaller elements than `pivot` in `list` (`smaller`) and elements which are equals to or greater than `pivot` (`rest`). For this the `partition()` method is used, which takes a predicate function to partition a sequence into two parts. In this case the predicate function is `_ < pivot`.
- The return value will be built of a list (1), the pivot element and another list (2). The method `::` is concatenating an element and a list in Scala, whereas `:::` is concatenating two lists which each other. The first list (1) is the result of the `qsort` method called on `smaller` and the second list (2) is the result of the method called on `rest`.

Listing 5.11: The quicksort algorithm using pattern matching in Scala inspired by [Wikipedia, 2013]

```
def qsort(list: List[Int]): List[Int] = list match {
  case Nil => Nil
  case pivot :: tail => {
    val (smaller, rest) = tail.partition(_ < pivot)
    qsort(smaller) ::: pivot :: qsort(rest)
  }
}
```

The example also shows the use of tuples in Scala, which are handy containers. They differ from other types of containers, in that they are always of fixed size and their elements can be of different types. Listing 5.12 shows some examples of how tuples in Scala can be used.

Listing 5.12: Tuples in Scala

```
val pair = (3, "hello")
println(pair) // prints (3,hello)
println(pair._2 + " you!") // prints hello you!

val swappedPair = pair.swap
println(swappedPair) // prints (hello,3)
```

Object-oriented extensions

The other programming paradigm which is implemented in Scala is object-orientation. Scala is a pure object-oriented language in which every value is an object. Data types and the behavior of objects in Scala are described in classes and traits. Scala allows extending classes as subclasses and has a flexible mixin-based inheritance. Mixin inheritance is a relatively recent form of inheritance, offering new capabilities compared to single inheritance, without the complexity of multiple inheritance [Schinz, 2005]. Traits in Scala are similar to interfaces in Java but allow the implementation of methods. Traits can not be instantiated on their own and always have to be mixed in to classes or be their superclass. Multiple traits can be mixed in to classes which enable the aforementioned mixin-based inheritance. When multiple traits implement the same method, the method of the last mixed in trait is called. By calling the superclass' method of the same name the previous trait (or the class itself) will be called. This functionality allows a chained composition. An example of this behavior is shown in listing 5.13.

Listing 5.13: Mixin-based inheritance in Scala inspired by [Wikipedia, 2013]

```
abstract class Window {
  // abstract
  def draw()
}

class SimpleWindow extends Window {
  def draw() {
    println("in SimpleWindow")
    // draw a basic window
  }
}

trait WindowDecoration extends Window { }

trait HorizontalScrollbarDecoration extends WindowDecoration {
  // "abstract override" is needed here in order for "super()" to work
  // because the parent
  // function is abstract. If it were concrete, regular "override"
  // would be enough.
  abstract override def draw() {
    println("in HorizontalScrollbarDecoration")
    super.draw()
    // now draw a horizontal scrollbar
  }
}

trait VerticalScrollbarDecoration extends WindowDecoration {
  abstract override def draw() {
    println("in VerticalScrollbarDecoration")
    super.draw()
    // now draw a vertical scrollbar
  }
}
```



```
}  
  
trait TitleDecoration extends WindowDecoration {  
  abstract override def draw() {  
    println("in TitleDecoration")  
    super.draw()  
    // now draw the title bar  
  }  
}  
  
// prints  
// in TitleDecoration  
// in HorizontalScrollbarDecoration  
// in VerticalScrollbarDecoration  
// in SimpleWindow:  
val mywin = new SimpleWindow with VerticalScrollbarDecoration with  
  HorizontalScrollbarDecoration with TitleDecoration
```

5.2. Scala in comparison to Java

Scala the Java-like language

According to the paper [Odersky et al., 2004a], which describes the design of Scala, Scala has been created as a Java-like language. It shares most of the basic operators, data types and control structures with Java and other mainstream platforms such as C#. Scala's syntax is intentionally conventional which allows developers from other platforms to learn it fast. In comparison to Java's syntax Scala differs mainly in the following points:

- Scala does not require semicolons at the end of statements.
- Scala uses the *id: type* syntax (e.g. `name: String`) for definition in opposition to Java which uses prefix types (e.g. `String name`).
- Scala's syntax is more regular as shown in section 5.1.2 *Scala features overview*.
- In Scala arrays are standard classes which do not require any special syntax.
- Scala adopts most of the control structures from Java except the traditional for-statement which has been replaced by for-comprehensions similar to the extended for-loop which was introduced in Java 5.0 but is more restrictive than the for-loop available in Scala.

Besides the small differences in syntax, Scala runs on the Java virtual machine (JVM)⁴ as Java programs do. Scala has been designed to inter-operate with Java programs. Most Scala programs access Java classes and libraries. An example for this is the `String` class of Java, which is being used in Scala. It is also possible to use Scala code from inside a Java program. To attract new developers to Scala which might be hesitant to try new languages, the project Scala IDE⁵ built on top of the Eclipse platform was created [McDirmid and Odersky, 2006].

Code examples

In this following section code examples show differences between Scala and Java when they are used in practice. The focus of this examples is to show that Scala programs usually need by far less written code due to their rich libraries and functional nature. The examples are inspired from [Stackoverflow, 2010].

The first example, which is about writing a program to index a list of keywords by their first letter shows how the functional style in Scala can compress and simplify Java code. The listing 5.14 shows the code written in Scala whereas the listing 5.15 shows the Java code to solve the problem. The logic for the program is as follows. The input is a list of strings. The output is a map, which can be queried for a letter to deliver a list of strings beginning with that letter. Those strings have been taken from the input string and are sorted alphabetically.

Listing 5.14: Indexing a list of keywords in Scala

```
object Main extends App {
  val keywords = List("Apple", "Ananas", "Mango", "Banana", "Beer")
  val result = keywords.sorted.groupBy(_.head)
  println(result)
}
```

Listing 5.15: Indexing a list of keywords in Java

```
import java.util.*;

class Main {
  public static void main(String[] args) {
    List<String> keywords = Arrays.asList("Apple", "Ananas", "Mango",
      "Banana", "Beer");
    Map<Character, List<String>> result = new HashMap<Character,
      List<String>>();
    for(String k : keywords) {
      char firstChar = k.charAt(0);
      if(!result.containsKey(firstChar)) {
        result.put(firstChar, new ArrayList<String>());
      }
    }
  }
}
```

⁴<http://www.java.com/en/>

⁵<http://scala-ide.org/>

```

    result.get(firstChar).add(k);
  }
  for(List<String> list : result.values()) {
    Collections.sort(list);
  }
  System.out.println(result);
}
}

```

The next listing shows the definition of a class in Scala (listing 5.16) and in Java (listing 5.17). The defined class is a `Person` with the attributes `firstName` and `lastName`. Additionally the class is `Serializable`, has a convenient constructor and constructors to create a copy of the person but with another `firstName` or `lastName`, getter-methods to retrieve the value of its attributes and the methods `equals`, `hashCode` and `toString` to compare instances, put them into maps and display their representation as `String`. Those methods are so commonly used, that some Java IDEs allow to generate the code for them [Holzner, 2004]. In Scala all that boiler plate code is not needed because of the `case` keyword in the class declaration. Case classes in Scala implicitly override some method definitions of `scala.AnyRef` to avoid code repetition [Odersky et al., 2004b].

Listing 5.16: Defining a class person with typical methods in Scala

```
case class Person(firstName: String, lastName: String)
```

Listing 5.17: Defining a class person with typical methods in Java

```

public class Person implements Serializable {
  private final String firstName;
  private final String lastName;

  public Person(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  public String getFirstName() {
    return firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public Person withFirstName(String firstName) {
    return new Person(firstName, lastName);
  }

  public Person withLastName(String lastName) {
    return new Person(firstName, lastName);
  }
}

```

```
}

public boolean equals(Object o) {
    if (this == o) {
        return true;
    }
    if (o == null || getClass() != o.getClass()) {
        return false;
    }
    Person person = (Person) o;
    if (firstName != null ? !firstName.equals(person.firstName) :
        person.firstName != null) {
        return false;
    }
    if (lastName != null ? !lastName.equals(person.lastName) :
        person.lastName != null) {
        return false;
    }
    return true;
}

public int hashCode() {
    int result = firstName != null ? firstName.hashCode() : 0;
    result = 31 * result + (lastName != null ? lastName.hashCode() :
        0);
    return result;
}

public String toString() {
    return "Person(" + firstName + "," + lastName + ")";
}
}
```

5.3. Scala as domain specific language for the web

After introducing Scala in *5.1 Introduction to Scala* and comparing it to Java *5.2 Scala in comparison to Java* this section explains why Scala is a fitting programming language for the web. As [Odersky et al., 2004a] points out, Scala was built as a scalable multi-purpose language. It has been designed to be scalable in the sense that the same concepts can describe small as well as large parts of a software. Scala is also scalable in the means of having concurrency features. Scala uses an actor library which provides actor-based concurrency supporting high-level communication through messages and pattern matching [Haller and Odersky, 2009]. This actor approach is made possible in Scala by enforcing a distinction between immutable and mutable objects. Immutable objects are unmodifiable and read-only whereas mutable objects can be changed.

As discussed in the chapter 1 *Introduction* the modern web needs horizontal scalability to satisfy the demand of the growing user base and with it the rising number of requests per second. Scala with its actor library is built to be able to handle this demand. Akka⁶ is a toolkit written in Scala for building concurrent, distributed applications on the JVM using the actor model [Thurau, 2012]. The actor model is based on small, independent and concurrent primitives which are called actors. Actors communicate with each other by sending messages. After receiving such a message an actor can do one of the following things:

- send messages to other actors
- create new actors
- change data saved within the actor

An example of how Scala utilizes Akka and the actor model can be seen in listing 5.18. In this example, a Scala application is started to initialize an actor `Greeter` which receives then a message. The message is a string which the actor uses to print out to the screen.

Listing 5.18: Hello World using Akka in Scala [Thurau, 2012]

```
import akka.actor._

class Greeter extends Actor {
  def receive = {
    // if we receive a String we print a greeting
    case name: String => println("Hello " + name)
  }
}

object Main extends App {
  // create the actor system
  val system = ActorSystem("GreeterSystem")
  // create an actor
  val greeter = system.actorOf(Props[Greeter], name="greeter")
  // send greeter a message
  greeter ! "World"
  //shutdown the actor system
  system.shutdown()
}
```

By distributing Akka's actors over different servers parallel algorithms as described in [Xavier and Iyengar, 1998] can be used to achieve horizontal scalability. For web pages Scala offers even more tools and techniques to ease the development of a scalable application. One of the is the Play framework, which will be discussed in detail in the next chapter 6 *Description of used Technologies and Frameworks*.

⁶<http://akka.io/>

6. Description of used Technologies and Frameworks

This chapter introduces the major technologies and frameworks chosen to empower Trista. In the first section *6.1 Architectural overview of Trista* an overview of Trista will be given from an architectural high level perspective. This overview helps to understand, which technological requirements in Trista require the chosen technologies in the following sections. Among them is the play! web framework, which is described in *6.2 The play! web framework*. A description of the NOSQL database MongoDB follows in *6.3 The MongoDB database*. The last section of this chapter *6.4 The Hadoop Distributed File System* introduces the HDFS Filesystem, which is used to allow Trista handling files in a horizontal scalable manner.

6.1. Architectural overview of Trista

This section gives an architectural overview of Trista which is required to understand the chosen technologies within the designed hybrid wiki. The architecture here is directly derived from the requirements described in *4 Trista requirements elicitation* and the nature of the project as a content oriented web application as described in [Shklar and Rosen, 2004] and [Fowler, 2003]. Figure 6.1 displays the overview of Trista's architecture.

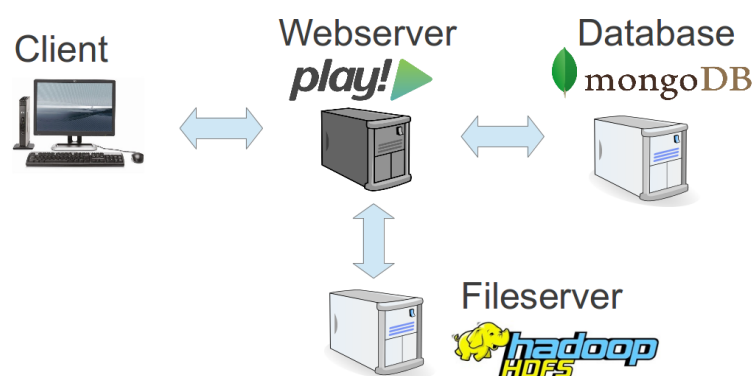


Figure 6.1.: Overview of Trista's architecture.

The main elements of this graphic are the web application itself, the database and the filesystem. The only component which is directly exposed to the user is the web application.

The database and the filesystem are in a supporting role. The database is used to store the content of the hybrid wiki, while the filesystem is used to store uploaded files from users and to provide them for download when requested. As a major non-functional requirement of Trista is its horizontal scalability all components shown must be able to satisfy this requirement. The following sections will explain in detail why the corresponding technologies have been chosen.

6.2. The play! web framework

The play! web framework¹ has been chosen as the web framework to power the web application itself. It is advertised as the high velocity web framework for Java and Scala. It is open source and based on a lightweight, stateless, web-friendly architecture. It is built on Akka to enable highly scalable applications [Play, 2013a].

It is designed to allow creating and deploying web applications in a more convenient way similar to frameworks like Ruby On Rails², Grails³ or Django⁴. This approach to web development is called rapid web development [Kolbe, 2012]. To enable it, the play! framework follows the principle of convention over configuration as described in [Miller, 2009]. Another beneficial fact is, that the play! framework is a full stack framework in which all necessary components and APIs are integrated already. Among them are Akka!⁵ and typical used libraries of a web framework like JSON⁶ converters. It allows to develop in a fix and reload approach, in which the developer can fix something in code and test it immediately by just reloading the website of the running application. If there is an error, the play! framework displays it to the user in the requested website as shown in figure 6.2.



Figure 6.2.: A screenshot of a compilation error in play shown to the user after reloading the web page.

¹<http://www.playframework.com/>

²<http://rubyonrails.org/>

³<http://www.grails.org/>

⁴<https://www.djangoproject.com/>

⁵<http://akka.io/>

⁶<http://www.json.org/>

The play! framework follows a RESTful (representational state transfer) approach, which enables higher scalability and reliability [Tyagi, 2006, Richardson and Ruby, 2008]. One constraint of REST is that it does not allow states, that is why the play! framework is stateless. The way play! enables sessions for users is by storing session data in Cookies at the client site. For security reasons the values stored at the client site are signed with a secret key so that the client can not modify the cookie data without invalidating it [Play, 2013b].

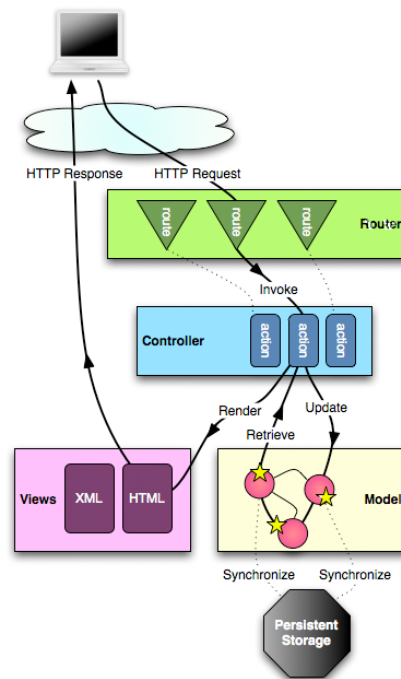


Figure 6.3.: The architecture of the play! framework.

The architecture of the play! framework follows the MVC (model-view-controller) pattern. Figure 6.3 shows the architecture of the play! framework and how a respond to a request is formed. When the play! application receives a HTTP request, its routes file determines which controller handles the request. The invoked controller, which code is in Java or Scala, may then manipulate the model and synchronize those manipulations with the data storage. After that optional step, a HTTP response for the request is created. Therefore the controller may render HTML from a template file. The result is then send to the client as a HTTP response [Reelsen, 2011]. The template files in play! use their own type safe template engine which is based on Scala and was inspired by ASP.NET Razor⁷. The files are written in a template markup syntax in HTML and Scala [Play, 2013c]. There is IDE support for template files and route files of the play! framework available from the Scala IDE project [IDE, 2013].

⁷<http://www.asp.net/>

6.3. The MongoDB database

The MongoDB⁸ database is an open source NoSQL database. Its name origins from the word humongous and it is developed by 10gen⁹. MongoDB stores structured data as JSON-like (BSON, Binary JSON) documents with dynamic schemas allowing faster development. According to [DB-Engines, 2013] it is the most popular NoSQL database.

Listing 6.1: An example page of Trista stored as BSON in MongoDB

```
{
  "_id" : { "$oid" : "5133b621e4b0e861ee6a9cd9" },
  "name" : "Welcome Page",
  "space" : "5133b621e4b0e861ee6a9cd8",
  "creator" : "5133b621e4b0e861ee6a9cd7",
  "lastEditor" : "5133b625e4b0e861ee6a9cda",
  "created" : 1362343457050,
  "lastEdit" : 1362347419936,
  "hybridType" : "Unknown",
  "attributes" : [ ],
  "text" : "<b>Welcome to the Page Welcome Page!</b>",
  "displayOptions" : {
    "showText" : true,
    "showAttributes" : true,
    "showIncomingReferences" : true,
    "showFilesTab" : true
  },
  "files" : [
    { "name" : "install.log", "mimetype" : "application/octet-stream" }
  ]
}
```

The listing 6.1 shows how an example page of Trista is stored in MongoDB. This page in Trista is modeled as a document containing various values. In contrast to a relational database management system, the values may contain array of values or even an entire subordinate document. In such a document, fields can be added, removed or modified at any moment due to the schemaless format. Documents in MongoDB are stored in so called collections. Usually, documents of the same type are stored within the same collection. MongoDB supports queries by field, range or regular expressions. It further allows to use Javascript functions, map-reduce [Dean and Ghemawat, 2008] and has its own Aggregation framework which provides a means to calculate aggregated values without having to use map-reduce [Chodorow and Dirolf, 2010].

One of the advantages of MongoDB is its capability of scalability and high availability. High availability in MongoDB is achieved by replication which ensures redundancy, backup and automatic failover. The replication occurs through groups of servers which are called

⁸<http://www.mongodb.org/>

⁹<http://www.10gen.com/>

replica sets. MongoDB achieves its horizontal scalability by an approach which is called sharding. The concept of sharding implies splitting data into ranges based on a shard key and distribute them across multiple servers which are called shards. By performing sharding, a single logical database system is distributed across a cluster of machines [Chodorow, 2011].

6.4. The Hadoop Distributed File System

HDFS (Hadoop Distributed File System) is part of Apache Hadoop¹⁰, which is an open source software framework to support data intensive distributed applications. The design of HDFS was derived from white papers of the Google File System (GFS), which is proprietary and developed by Google for its own use [Ghemawat et al., 2003]. The distributed and horizontal scalable file system is designed to store large data sets reliably on multiple machines and stream those data sets to user applications. The reliability mechanisms of HDFS allow hardware failure [Shvachko et al., 2010].

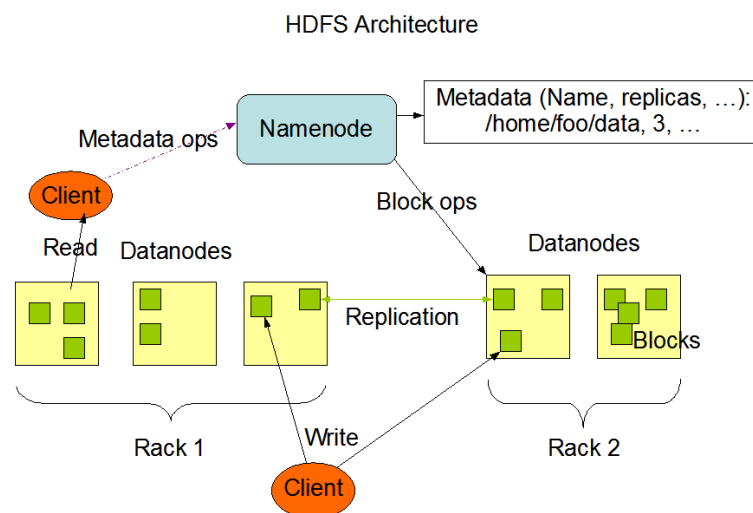


Figure 6.4.: The architecture of HDFS [Hadoop, 2013].

Figure 6.4 shows the master/slave architecture of HDFS including a NameNode and multiple DataNodes distributed across two data racks. When a client requests file system namespace operations like opening, closing, and renaming files and directories then those are executed by the NameNode. The NameNode also determines the mapping of blocks to DataNodes. Write and Read requests by the client are handled by the DataNodes. They also perform block creation, deletion, and replication upon instruction from the NameNode [Borthakur, 2007].

¹⁰<http://hadoop.apache.org/>

Part III.

Trista Implementation

7. Analysis

This chapter presents the result of the analysis of the 4 *Trista requirements elicitation*. It concentrates on identifying entity, control and boundary objects which can be directly concluded from the requirements. The first section 7.1 *Hybrid wiki objects* describes the objects which are used in the context of the hybrid wiki nature inside of Trista. User interface objects are used to bring information to the user and help building the web view elements. They and the mentioned web view elements are described in section 7.2 *Web view elements*.

7.1. Hybrid wiki objects

The hybrid wiki objects in Trista are those objects which are used for the basis functionality of the hybrid wiki and can be found in every hybrid wiki. Their purpose is described in detail in chapter 2 *Hybrid wikis*.

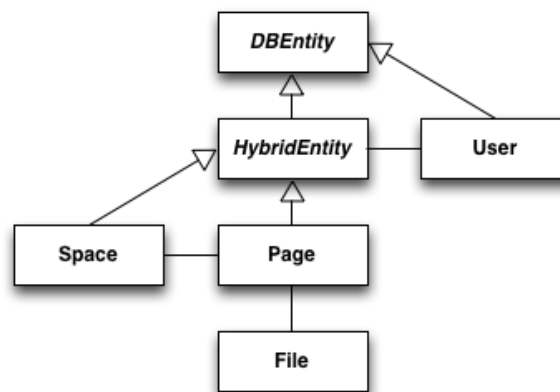


Figure 7.1.: An UML class diagram showing the hybrid wiki objects used in Trista and their relation to each other.

The UML diagram in figure 7.1 shows the hybrid wiki objects used in Trista and their relation to each other. The main elements are *Space*, *Page*, *File* and *User*. Section 2.3 *Core concepts* elaborates on their attributes and purpose in the context of the hybrid wiki. All objects except for *File* objects are *DBEntity* objects because they can be directly stored to the database. Besides that *Space* and *Page* are both objects which can be part of another *Space* and are therefore both *HybridEntity* objects. Those objects are also associated with

User objects because users create those objects in the hybrid wiki. File objects are always associated with a Page because files are attached to pages and can not exist on their own.

7.2. Web view elements

The web view elements in Trista are the webpages and their elements which are shown to the user. As the play! framework provides ways to extract often used elements of a web page to own objects, it allows Trista to structure its webpages in web view elements. The objects used to create those web view elements are the user interface objects in Trista. Those objects are of supporting nature, are not stored to the database and are not directly derived from the hybrid wiki concept. The identified user interface objects in Trista are:

- The `UiContainter` object is used to include all data which is needed to build the main web view element, which is used on every standard web page of Trista except for the login page.
- The `UiMessage` objects are used to contain messages which can be displayed to the user to confirm successful actions or warn of errors.

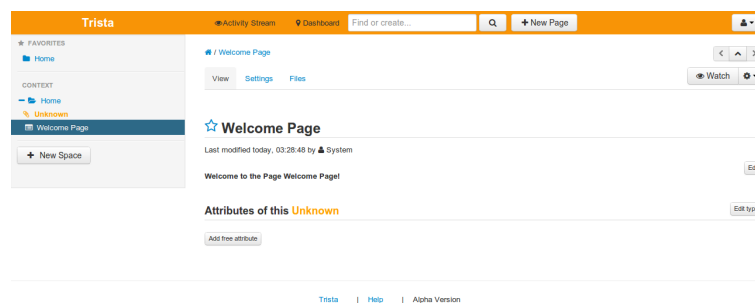


Figure 7.2.: A page displayed in Trista.

The web view elements are derived from the user interface of Trista which is based on the redesigned user interface of Tricia 3.4. Figure 7.2 shows the Trista user interface with an opened page object. Most web pages in Trista have the same layout and use the same elements. Those elements are the header on the top, the footer on the bottom, the explorer on the left and the main panel, which is individual to the requested web page. Figure 7.3 shows a high level overview of the web view elements in Trista. In this figure the main panel area is grey and it contents the two components breadcrumb and content, which are used when hybrid wiki entities such as spaces and pages are displayed.

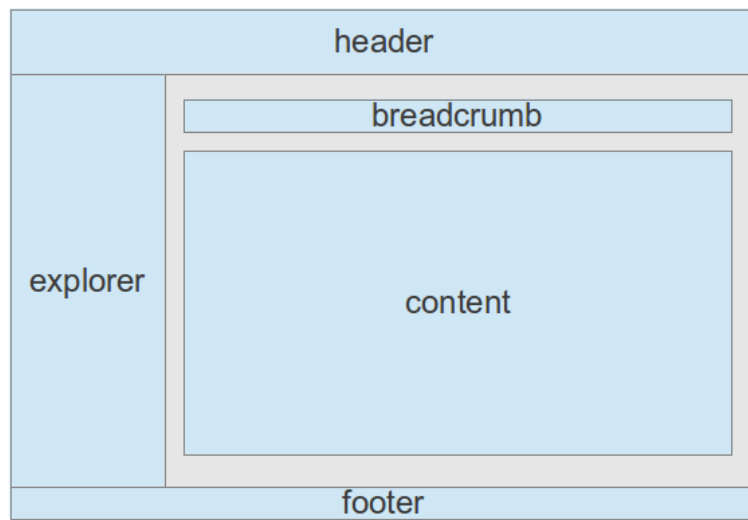


Figure 7.3.: High level overview of web view elements in Trista.

8. System Design

This chapter describes the system design of the hybrid wiki Trista. It gives an overview of the subsystems in Trista, their interaction with each other and their context in the MVC (model-view-controller) pattern.

The subsystems in Trista are oriented on the MVC (model-view-controller) pattern, which is described in [Burbeck, 1992]. It is a software pattern which separates the representation of information from the user's interaction with it. The pattern describes the three components controller, model and view which interact with each other. The controller can manipulate the model, which is then represented by the view. The view can not manipulate the model and is commanded by the controller to update its state when the model changes.

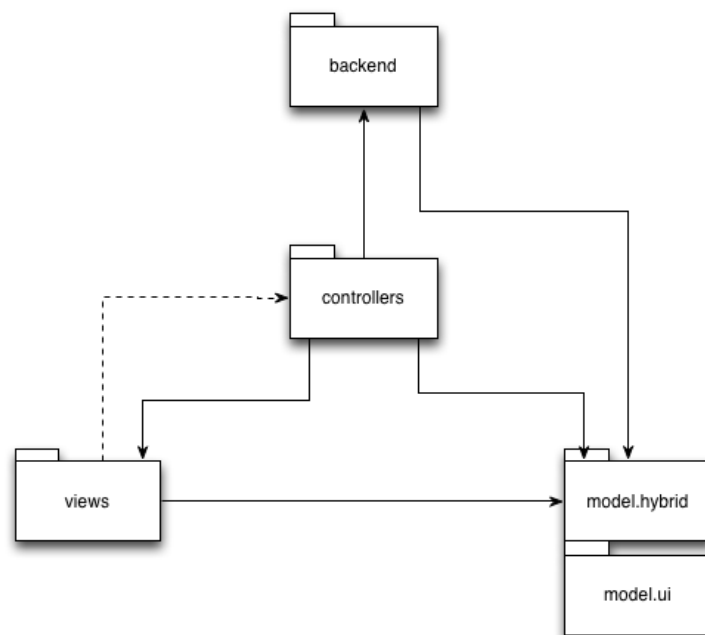


Figure 8.1.: An overview of the subsystems in Trista and their interaction with each other.

In Trista the MVC pattern is implemented as seen in figure 8.1. In this figure a line symbolizes a direct association whereas the dotted line symbolizes an indirect association. It conforms to the classical MVC pattern but includes more information on the packages used in Trista. In the classical MVC pattern, the `backend` subsystem is part of the `controllers` subsystem and the `model` subsystem is not displayed in detail. The interaction of the

displayed subsystems in Trista is as follows:

- The `controllers` subsystem is triggered by an user action on the web page, which is part of the `views` package. It then accesses information from the `backend` to retrieve information or manipulate it in the `model.hybrid` package. That information is then shown to the user by providing `model.ui` objects to the `views` package, which uses them to create the web page as described in 7.2 *Web view elements*.
- The `views` subsystem contains the web view elements, which are used to create the web page which is shown to the user. It accesses the objects in `model`, which have been created from the `controllers` package.
- The `backend` subsystem contains objects which are used by the `controllers` to retrieve and manipulate data in `model`.
- The `model.hybrid` subsystem contains objects which are part of the hybrid wiki concept as described in 7.1 *Hybrid wiki objects*. They are retrieved and manipulated by the `backend`, used by the `controllers` and shown by the `views` package.
- The `model.hybrid` subsystem contains objects which have been created by the `controllers` to be shown to the user by the `views` subsystem.

Furthermore the subsystem `backend` is divided into the packages `backend.core.file`, `backend.core.search` and `backend.data`. The `backend.core.file` package contains the classes which allow Trista to store, delete and retrieve files from various destinations. The `backend.core.search` subsystem is used to bundle the classes, which are used by the `controllers` to perform a search on the database. It uses the `backend.data` package, which contains the data access objects.

The `views` package, which contains the web view elements, which are written in the markup language of the `play!` framework, is also divided. It has the subpackages `views.activity`, `views.page`, `views.search` and `views.space`. Whereas the `views` package itself contains classes which are used by its subpackages or can not be assigned to another package, the package `views.activity` contains web elements in the context of hybrid wiki activities, `views.page` contains web elements which are used to display pages, `views.search` contains web elements for the search option in Trista and `views.space` contains web elements which are used to display spaces.

The top-down system design of Trista allows flexible extensibility by providing a coherent structure of subsystems. Additional features can either be put in one of the already existing subsystems or can be put in their own new subsystem.

9. Object Design and Implementation

This chapter deals with the object design and the implementation of the hybrid wiki Trista. It describes important aspects of the implementation as well as highlights decisions which have been made during the design process. The first section 9.1 *Hybrid wiki object implementation* describes how the objects of the hybrid wiki are implemented in Trista. The second section 9.2 *Database access* describes how the database access of MongoDB is abstracted in Trista. In the last section 9.3 *File handling* Trista's file handling is explained. It especially covers the different possibilities in Trista to store files on the local server as well as on the HDFS (Hadoop Distributed File System).

9.1. Hybrid wiki object implementation

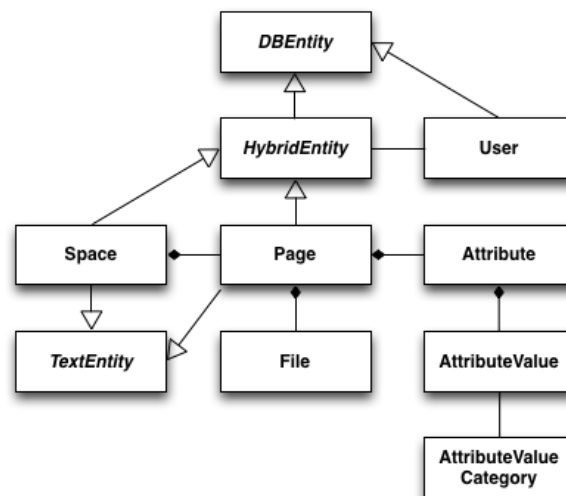


Figure 9.1.: An UML class diagram showing the object implementation of the hybrid wiki elements in Trista.

Figure 9.1 shows the object implementation of the hybrid wiki elements in Trista. The implementation of the hybrid wiki elements follows a simple approach to the objects and its properties described in section 2.3 *Core concepts*. It is based on the analysis done in section 7.1 *Hybrid wiki objects*. Additional to the already identified object attributes and associations, **Space** and **Page** classes extend the **TextEntity** trait in the implementation. It marks objects with a text attribute. Furthermore, **Space** and **Page** classes associate with display option

classes which contain attributes determining of how the objects will be shown to the users. The settings represented by those objects can also be modified by the user in the web application. The classes `Attribute`, `AttributeValue` and `AttributeValueCategory` are all used for the interaction of attributes in the context of the hybrid wiki pages. A `Page` object has a list of `Attributes`, which have a name and content. The content is a list of `AttributeValues`, which have a value and a category. The category is determined by the enumeration `AttributeValueCategory`. All classes are part of the `model.hybrid` package.

9.2. Database access

The database access in Trista satisfies the requirement, that its interfaces are independent from the actual used database. Nevertheless the database used by Trista is MongoDB, which is described in section 6.3 *The MongoDB database*. The design of the database access is based on the data access object (DAO) pattern. It provides an abstract interface to the data access without exposing details of the database and fulfills the separates of concerns design principle [Alur et al., 2003].

Figure 9.2 shows the UML diagram of the `backend.data` package in Trista. It contains the database access to MongoDB using the DAO pattern. The persistent classes `User`, `Page` and `Space` of Trista are handled by the corresponding DAOs `UserDao`, `PageDao` and `SpaceDao`. All DAOs are based on the `BaseDao`, which is a generic abstract class. It allows to save, modify and find entities, which are inherited from an abstract class `DBEntity`. For the access of the MongoDB database, it utilizes a `MongoDBLayer` object, which provides access to a `mongoDB` object from the `casbah` Scala MongoDB driver¹. The `BaseDao` class contains abstract write and read converter methods, which have to be implemented by its non-abstract subclasses. Those methods are used to write and read objects from the MongoDB database, which are in a JSON-like format. The `UserDao` object directly inherits from the `BaseDao` class and provides the database access for the `User` class.

The `SpaceDao` and `PageDao` objects inherit from the `HybridEntityDao` class, which is a generic abstract class specifying the `DBEntity` to be a subclass of a `HybridEntity`. The `HybridEntityDao` is a subclass of the `BaseDao` providing more functionality specified to instances of `HybridEntity` subclasses. Additionally to the inheritance of the `HybridEntityDao`, the `SpaceDao` and `PageDao` objects also inherit from the `TextEntityDao` trait. Similar to the `HybridEntityDao`, it provides more functionality to `HybridEntity` instances which have a text attribute. As `Page` and `Space` objects are both inheriting from `HybridEntity` and `TextEntity`, the inheritance of the DAOs is respective. Furthermore all implementations of DAOs provide beside the implementation of the converter methods their own specific domain object specific data access functions.

¹<https://github.com/mongodb/casbah>

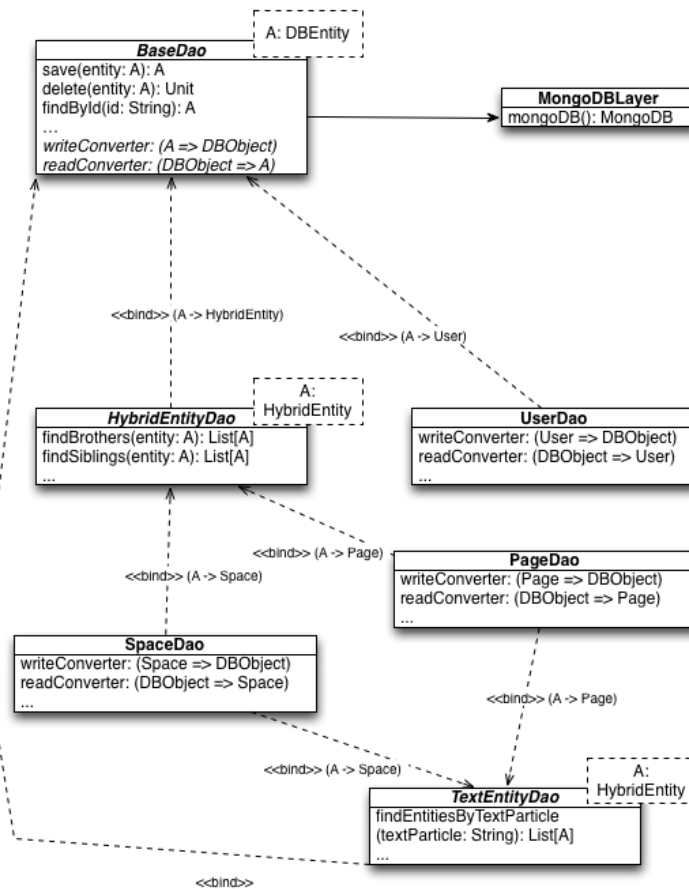


Figure 9.2.: The database access in Trista using MongoDB and the data access object (DAO) pattern as described in [Alur et al., 2003].

9.3. File handling

This section describes how file handling in Trista is done. Trista allows to upload files to pages and downloading them later. Those files can also be deleted. The technology for the file storage is the HDFS (Hadoop Distributed File System). It has been described in section 6.4 *The Hadoop Distributed File System*. To ease development in Trista and also allow new developers a quick access, there is an alternative file storage mechanism. It uses the standard Java IO API² and stores the files on the local file system. The implementation is so designed, that it allows to run Trista in development mode without installing and configuring the HDFS (Hadoop Distributed File System) on the machine. The advantage is, that a developer who does work in an area beside the file handling mechanisms of Trista is not needed to set up HDFS (Hadoop Distributed File System). This allows a quicker access to start developing.

²<http://docs.oracle.com/javase/7/docs/api/java/io/package-summary.html>

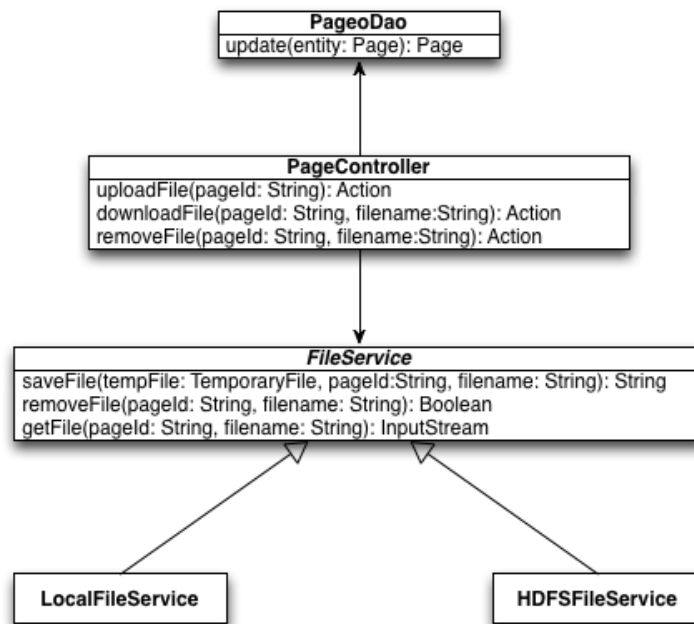


Figure 9.3.: A UML diagram showing the content of the `backend.core.file` package and how it is access by the `PageController` in Trista.

Figure 9.3 shows the UML diagram of how the `backend.core.file` package and how it is access by the `PageController` in Trista. It contains all classes and traits which provide the file system access in Trista. Similar to the database access, the underlying technology is exchangeable without the need of a modification of the interface. That interface is the trait `Fileservice`. It defines methods to save, remove and get files from the file system. Both the `HDFSFileService` and the `LocalFileService` are classes which implement that trait and provide the implementation of the corresponding technology.

The `HDFSFileService` as well as the `LocalFileService` can be configured in the `application.conf` file found in `conf`. The `HDFSFileService` requires a `core-site.xml` and a `hdfs-site.xml` for configuration³. Both paths may be attached to the mentioned configuration file. If the paths are defined at the start of Trista, the `HDFSFileService` will be chosen as implementation for the `FileService`. If they are not defined, the `LocalFileService` object will be used. The `LocalFileService` can also be configured. The path where Trista stores the files is definable in the `application.conf` file.

³http://hadoop.apache.org/docs/r1.0.4/hdfs_user_guide.html

Part IV.

Evaluation

10. Achievements in Trista

This chapter discusses the achievements in the development of Trista, the hybrid wiki developed in this thesis. The achievements are bound to the requirements set up in chapter 4 *Trista requirements elicitation*. It contains 16 functional requirements and 4 non-functional requirements.

Of the 16 functional requirements, 7 are of high, 6 are of medium and 3 are of low priority. The following table gives an overview of which requirements have been achieved and what their priority was. The not achieved requirements are written in cursive letters. All other requirements have been achieved. The table uses the short names of the requirements. Their description can be found in section 4.1 *Functional requirements*.

High priority	Medium priority	Low priority
#1 Spaces	#3 Space Settings	#8 <i>Constraints</i>
#2 Space Options	#6 Page Settings	#9 <i>Versions</i>
#4 Pages	#7 File Handling	#10 <i>Import and Export</i>
#5 Page Options	#12 <i>Access-Management</i>	
#11 Users	#13 Watch	
#15 Search	#14 Favorites	
#16 Dashboard		

As the table shows, all functional requirements of high priority have been achieved. Of the requirements with medium priority, only one requirement, the #12 *Access-Management*, has not been achieved. None of the 3 requirements with low priority have been achieved in the current implementation of Trista.

To implement the #12 *Access-Management* requirement in Trista, it would require to establish groups, roles and rights associations. Those concepts are described in 2.3 *Core concepts*. The implementation would furthermore require user interface elements to manage those rights as well as mechanisms in the application code of Trista to enforce those rules. None of the non implemented requirements depends on other tasks which have not been implemented yet. Therefore the attempt of their implementation is not dependent on any other requirements.

All of the non functional requirements in Trista are of high priority. They are #1 Scalability, #2 Performance, #3 Usability and #4 Fostering development. As the current implementation of Trista is of a prototype nature, the fit criteria of those requirements are formulated less strict and can therefore not be evaluated strictly.

Trista uses only horizontal scalable technologies and frameworks, which are described in the chapter 6 *Description of used Technologies and Frameworks*. Therefore the #1 Scalability requirement is fulfilled. because of the lightweight approach of Trista and its reduction to core features of a hybrid wiki, it is very responsive and performs fast. Together with the fulfilled dependency #1 Scalability the requirement #2 Performance can be viewed as fulfilled as well. The #3 Usability requirement is worth to be discussed. Trista's user interface is based upon the redesigned user interface of Tricia, which is released as Tricia 3.4. Therefore it shares a common ancestor with Tricia as well as the improvements of the Tricia community which have been developed with the experience of years in use of hybrid wikis. Therefore the #3 Usability requirement is most likely fulfilled. The last non functional requirement #4 Fostering development is also worth a discussion. Unlike Tricia, Trista uses only standard technology which has been proven useful by the open source community. Their documentation and source code is available as described in chapter 6 *Description of used Technologies and Frameworks*. Additionally rapid development has been enabled by using the play! framework. An easy access to development in Trista is made possible by reducing the install and configuration requirements for developers by allowing to run Trista without HDFS in development mode.

11. Conclusion

This chapter gives a conclusion over the thesis and the development of the Trista hybrid wiki. In the section *11.1 Summary* the thesis is summarized and the novel benefits of the thesis are underlined. The section *11.2 Outlook* gives an outlook to the future work possible in Trista and thoughts of the author about the potential of it.

11.1. Summary

The major goal of this thesis was to develop the hybrid wiki Trista. Furthermore Trista was developed to be a lightweight hybrid wiki, which is horizontal scalable and provides a high useability. Trista provides an easy access to developers by its key usage of open source technology and is an highly extensible platform enabling rapid development. The design and implementation of the hybrid wiki benefits from its predecessor Tricia. Unlike Tricia, Trista is reduced to the core functionality of hybrid wikis and has been built on modern technologies.

In the first part of this thesis, the hybrid wiki concept is explained. It contains spaces, pages and their types and attributes. A major thought behind it is that unlike in semantic wikis, elements of the hybrid wiki are schemaless and allow managing adaptive data models. As initially shown by [Neubert, 2013] and described in this thesis, hybrid wikis are more accessible for business users and empower them to capture easily content in structured form. It allows to incrementally structure and classify content objects without the struggle of being enforced to use strict information structures.

The second part of the thesis concentrates on the architectural design of Trista and the technologies as well as frameworks used. Among them is Scala, the multi-paradigm programming language built on top of the Java virtual machine (JVM), the play! web framework, the document-oriented database MongoDB and the HDFS (Hadoop Distributed File System) of Apache Hadoop. Those technologies enable Trista to fulfill its requirements. All of them are designed to be horizontal scalable and easy accessible for new users allowing quick development.

The implementation of Trista is then discussed in the third part of the thesis. It contains the analysis on the requirements elicitation for Trista, in which entity, control and boundary objects are identified. It also describes the architecture and used subsystems of Trista, giving an insight in the modular built of Trista. Trista follows the model-view-controller (MVC)

pattern in its architecture which is a flexible architecture pattern allowing easy extensions as well as reuse of components. In the last chapter of this part, the object design and implementation of Trista is discussed. It explains some design as well as implementation decisions made. Among them is the database access, which has been implemented using the data access object (DAO) pattern. The implementation of the file handling in Trista allows a high flexibility of the underlying filesystem. In Trista a file handler using the standard Java IO API as well as a file handler using the HDFS (Hadoop Distributed File System) is implemented.

In the last part of this thesis, the achievements of Trista are summarized. Of the 16 functional requirements in Trista, of which 7 are of high, 6 are of medium and 3 are of low priority, 12 have been implemented. All requirements of high priority have been achieved as well as all of medium priority except for one. None of the low priority requirements has been implemented in Trista.

11.2. Outlook

As explained in the previous section *11.1 Summary*, the goal of the implementation of Trista was to make it extendable. Therefore it is worth a discussion in which direction Trista will go after this thesis. Trista was designed with the knowledge of all the features available in Tricia. Therefore those features or a subset of them would fit naturally in to the current platform. Besides those there are even more obvious features which have already been described in this thesis. They are the functional requirements which have not been implemented yet.

One of the major strengths of the new platform is its easy access to it as a new developer. Documentation on the used technologies is available for free in the internet as well as their source code due to their open source nature. This allows to attract new developers and promises quick results in development. The chosen technologies even go one step further. The play! framework allows to develop and test on Trista on the fly by coding and just reloading pages to test it. This enables the aforementioned rapid development. Therefore Trista is not limited to go the same way which Tricia has gone and allows to build new ideas on it.

A main advantage and big theme of Trista is horizontal scalability. This thesis and the implementation of Trista lay the building block for interesting questions regards the deployment of Trista. It is desirable to test Trista's capabilities in a deployment of a cluster instance. Because of the design in Trista, which allows to exchange the underlying technologies used, Trista could be used to evaluate new technologies, e.g. new databases and test their performance in the context of a hybrid wiki. For future development and especially the use of new technologies in Trista it is important to reflect on their influence on the horizontal scalability of Trista.

List of Figures

2.1.	An UML class diagram showing the model of hybrid wikis.	8
2.2.	A space displayed in Trista.	9
2.3.	A page displayed in Trista.	10
2.4.	The files tab of a page shown in Trista.	11
2.5.	Types of an opened space as shown in the context menu in Trista.	11
2.6.	Attributes of a page in Trista.	12
2.7.	Incoming references of a page in Trista.	12
2.8.	The favorites menu of a user in Trista.	13
2.9.	A dashboard of a user in Trista.	14
2.10.	Suggested attribute keys in cursive letters for a page in Trista.	16
2.11.	An UML class diagram showing the concept of hybrid wikis as interpreted by Christian Neubert in his dissertation [Neubert, 2013]	17
3.1.	Tricia in its current deployment as homepage of infoAsset.	20
5.1.	Scala’s type hierarchy [Wampler and Payne, 2009].	41
6.1.	Overview of Trista’s architecture.	51
6.2.	A screenshot of a compilation error in play shown to the user after reloading the web page.	52
6.3.	The architecture of the play! framework.	53
6.4.	The architecture of HDFS [Hadoop, 2013].	55
7.1.	An UML class diagram showing the hybrid wiki objects used in Trista and their relation to each other.	59
7.2.	A page displayed in Trista.	60
7.3.	High level overview of web view elements in Trista.	61
8.1.	An overview of the subsystems in Trista and their interaction with each other.	63
9.1.	An UML class diagram showing the object implementation of the hybrid wiki elements in Trista.	65
9.2.	The database access in Trista using MongoDB and the data access object (DAO) pattern as described in [Alur et al., 2003].	67
9.3.	A UML diagram showing the content of the <code>backend.core.file</code> package and how it is access by the <code>PageController</code> in Trista.	68

Bibliography

- [Alur et al., 2003] Alur, D., Crupi, J., and Malks, D. (2003). *Core J2EE Patterns: Best Practices and Design*. Prentice Hall.
- [Bass et al., 2013] Bass, L., Clements, P., and Kazman, R. (2013). *Software Architecture in Practice*. Addison-Wesley, USA, third edition.
- [Beck, 2003] Beck, K. (2003). *Extreme Programming*. Addison-Wesley, München, Germany.
- [Boehm, 1991] Boehm, B. W. (1991). Software risk management: principles and practices. *Software, IEEE*, 8(1):32–41.
- [Borthakur, 2007] Borthakur, D. (2007). The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11:21.
- [Burbeck, 1992] Burbeck, S. (1992). Applications programming in smalltalk-80 (tm): How to use model-view-controller (mvc). *Smalltalk-80 v2. 5. ParcPlace*.
- [Chodorow, 2011] Chodorow, K. (2011). *Scaling MongoDB*. O’Reilly Media.
- [Chodorow and Dirolf, 2010] Chodorow, K. and Dirolf, M. (2010). *MongoDB: the definitive guide*. O’Reilly Media.
- [Cormen et al., 2001] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms, Second Edition*. The MIT Press, 2nd edition.
- [DB-Engines, 2013] DB-Engines (2013). Db-engines ranking. <http://db-engines.com/en/ranking>. [Online; Accessed: 2013-03-19].
- [Dean and Ghemawat, 2008] Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.
- [Devlin, 1993] Devlin, K. J. (1993). *The joy of sets: fundamentals of contemporary set theory*. Springer Verlag.
- [Fensel, 2011] Fensel, D. (2011). *Semantic Web Services*. Springer.
- [Fowler, 2003] Fowler, M. (2003). *Patterns of enterprise application architecture*. Addison-Wesley Professional.

- [Gamma et al., 2001] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (2001). *Design patterns: Abstraction and reuse of object-oriented design*. Springer.
- [Ghemawat et al., 2003] Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM.
- [Ghidini et al., 2009] Ghidini, C., Kump, B., Lindstaedt, S., Mahbub, N., Pammer, V., Rospocher, M., and Serafini, L. (2009). Moki: The enterprise modelling wiki. *The Semantic Web: Research and Applications*, pages 831–835.
- [Hadoop, 2013] Hadoop (2013). Hdfs architecture guide. http://hadoop.apache.org/docs/r1.0.4/hdfs_design.html. [Online; Accessed: 2013-03-19].
- [Haller and Odersky, 2009] Haller, P. and Odersky, M. (2009). Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202–220.
- [Henderson, 2006] Henderson, C. (2006). *Building Scalable Web Sites*. O’Reilly Associates, Inc., Sebastopol, CA, USA, first edition.
- [Holzner, 2004] Holzner, S. (2004). *Eclipse Cookbook*. O’Reilly Media, Incorporated.
- [IDE, 2013] IDE, S. (2013). Play2 support 0.1.0. <http://scala-ide.org/blog/play-0.1.0-announcement.html>. [Online; Accessed: 2013-04-10].
- [Joy et al., 2000] Joy, B., Steele, G., Gosling, J., and Bracha, G. (2000). *Java(TM) Language Specification*. Addison-Wesley, second edition.
- [Kolbe, 2012] Kolbe, P. (2012). The play! framework.
- [Krötzsch et al., 2006] Krötzsch, M., Vrandečić, D., and Völkel, M. (2006). Semantic mediawiki. *The Semantic Web-ISWC 2006*, pages 935–942.
- [Mandrioli and Meyer, 1992] Mandrioli, D. and Meyer, B. (1992). *Advances in Object-Oriented Software Engineering*. Prentice Hall International (UK) Ltd, Cambridge, Great Britain, first edition.
- [McAfee, 2009] McAfee, A. (2009). *Enterprise 2.0: New Collaborative Tools for Your Organization’s Toughest Challenges*. Harvard Business School Press, 1 edition.
- [Mcafee, 2006] Mcafee, A. P. (2006). Enterprise 2.0: The Dawn of Emergent Collaboration. *Management of Technology and Innovation*, 47(3).
- [McDirmid and Odersky, 2006] McDirmid, S. and Odersky, M. (2006). The scala plugin for eclipse. In *Proceedings of Workshop on Eclipse Technology eXchange (ETX)*.
- [Miller, 2009] Miller, J. (2009). Patterns in practice-convention over configuration. *MSDN magazine*, page 39.

- [Neubert, 2013] Neubert, C. (2013). *Facilitating Emergent and Adaptive Information Structures in Enterprise 2.0 Platforms*. PhD thesis, Technische Universität München, München, Germany.
- [Odersky, 2006] Odersky, M. (2006). A brief history of scala. <http://www.artima.com/weblogs/viewpost.jsp?thread=163733>. [Online; Accessed: 2013-03-14].
- [Odersky, 2008] Odersky, M. (2008). Scala's prehistory. <http://www.scala-lang.org/node/239>. [Online; Accessed: 2013-03-14].
- [Odersky, 2011] Odersky, M. (2011). *Scala By Example*.
- [Odersky et al., 2004a] Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M. (2004a). An overview of the scala programming language. Technical report, Citeseer.
- [Odersky et al., 2004b] Odersky, M., Altherr, P., Cremet, V., Emir, B., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M. (2004b). The scala language specification.
- [Odersky et al., 2011] Odersky, M., Spoon, L., and Venners, B. (2011). *Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition*. Artima Inc, 2 edition.
- [Oracle, 2013] Oracle (2013). The java ee 5 tutorial - web modules. <http://docs.oracle.com/javase/5/tutorial/doc/bnadx.html>. [Online; Accessed: 2013-03-11].
- [O'Reilly, 2013] O'Reilly, T. (2013). What is web 2.0. <http://oreilly.com/web2/archive/what-is-web-20.html>. [Online; Accessed: 2013-02-16].
- [Passant, 2011] Passant, A. (2011). *Semantic Web Technologies for Enterprise 2.0*. IOS Press.
- [Play, 2013a] Play (2013a). Play framework. <http://www.playframework.com/>. [Online; Accessed: 2013-03-19].
- [Play, 2013b] Play (2013b). Session and flash scopes. <http://www.playframework.com/documentation/2.0/ScalaSessionFlash>. [Online; Accessed: 2013-03-19].
- [Play, 2013c] Play (2013c). The template engine. <http://www.playframework.com/documentation/2.0/ScalaTemplates>. [Online; Accessed: 2013-03-19].
- [Reelsen, 2011] Reelsen, A. (2011). *Play Framework Cookbook*. Packt Pub Limited.
- [Reussner and Hasselbring, 2006] Reussner, R. and Hasselbring, W. (2006). *Handbuch der Software-Architektur*. dpunkt.verlag GmbH, Oldenburg, Germany, first edition.
- [Richardson and Ruby, 2008] Richardson, L. and Ruby, S. (2008). *RESTful web services*.

O'Reilly Media.

- [Rosenfeld and Morville, 2002] Rosenfeld, L. and Morville, P. (2002). *Information Architecture for the World Wide Web*. O'Reilly Associates, Inc., Sebastopol, CA, USA, second edition.
- [Scala, 2008] Scala (2008). A tour of scala: Currying. <http://www.scala-lang.org/node/135>. [Online; Accessed: 2013-03-16].
- [Scala, 2011a] Scala (2011a). Commercial support for scala. <http://www.scala-lang.org/node/9484>. [Online; Accessed: 2013-03-14].
- [Scala, 2011b] Scala (2011b). Scala team wins erc grant. <http://www.scala-lang.org/node/8579>. [Online; Accessed: 2013-03-14].
- [Schinz, 2005] Schinz, M. (2005). *Compiling Scala for the Java virtual machine*. PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE.
- [Schmidt et al., 2001] Schmidt, D., Stal, M., Rohnert, H., and Buschmann, F. (2001). *Patterns for Concurrent and Networked Objects*, volume 2 of *Pattern-Oriented Software Architecture*. John Wiley & Sons, Ltd, West Sussex, England, reprinted edition.
- [Shklar and Rosen, 2004] Shklar, L. and Rosen, R. (2004). *Web application architecture: Principles, protocols and practices*. Wiley.
- [Shvachko et al., 2010] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE.
- [Stackoverflow, 2010] Stackoverflow (2010). Samples of scala and java code where scala code looks simpler/has fewer lines? <http://stackoverflow.com/questions/2952732/samples-of-scala-and-java-code-where-scala-code-looks-simpler-has-fewer-lines>. [Online; Accessed: 2013-03-18].
- [Thurau, 2012] Thurau, M. (2012). Akka framework.
- [Twitter, 2013] Twitter (2013). Year in review. <http://yearinreview.twitter.com/en/tps.html>. [Online; Accessed: 2013-02-12].
- [Tyagi, 2006] Tyagi, S. (2006). Restful web services. <http://www.oracle.com/technetwork/articles/javase/index-137171.html>. [Online; Accessed: 2013-03-19].
- [Varhol, 1993] Varhol, P. D. (1993). *Object-Oriented Programming*. Computer Technology Research Corp., Charleston, South Carolina, USA, updated edition.
- [Volere, 2013] Volere (2013). Volere requirements specification template. <http://www>.

volere.co.uk/template.htm. [Online; Accessed: 2013-03-12].

[Wampler and Payne, 2009] Wampler, D. and Payne, A. (2009). *Programming Scala: Scalability= Functional Programming+ Objects*. O'Reilly Media.

[Wikipedia, 2013] Wikipedia (2013). Scala (programming language). [http://en.wikipedia.org/wiki/Scala_\(programming_language\)](http://en.wikipedia.org/wiki/Scala_(programming_language)). [Online; Accessed: 2013-03-17].

[Xavier and Iyengar, 1998] Xavier, C. and Iyengar, S. S. (1998). *Introduction to parallel algorithms*, volume 1. Wiley-Interscience.