Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

Liping Wang

MASTER THESIS

A Wiki Framework for the Sweble Engine

Submitted on 10.06.2014

Supervisor:
Dipl.-Inf. Hannes Dohrn
Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander University Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

[CITY], [DATE]

License

[CITY], [DATE]

Acknowledgements

First and foremost, I would like to express my sincere gratitude to Mr. Hannes Dohrn, who has given his constructive guidance both on my program implementation and thesis writing. I would also like to express my great appreciate to Prof. Dirk Riehle for his inspiration and support.
Last but not least, I would like to express my deepest gratitude to my family for their always unconditional encouragement and support.

Abstract

Along with the popularization of wikis, the underlying wiki technologies become hot topics in the research community. The open source research group[1] developed a formal parser for Wikitext, the wiki markup language of MediaWiki. The parser generates a high-level and machine-accessible representation of wiki content so that it can be easily queried and translated into arbitrary target formats. Furthermore, they proposed a framework based on the wiki object model (WOM) to simplify and improve wiki content transformation and refactoring. The WOM is a generalized data structure that uncouples from the syntactic idiosyncrasies of wiki markup languages. The goal of this thesis is to develop a fully-functional wiki engine, which connects the above mentioned components, adds a storage solution with revision control and a web frontend. Meanwhile, the software implementation should be well-structured and modularized in order to favor the further extensions in the future.

Keywords: wiki, wiki software, Sweble, Wiki object model, WOM.

1 The open source research group, Computer Science Department, Friedrich-Alexander-University (FAU) of Erlangen-Nuernberg.

Table of Contents

List of Figures

1 Introduction

1.1 Original thesis goals

The wiki technologies and their business usages are the research fields of the open source research (OSR) group in Computer Science Department, FAU. During the past years, OSR has done extensive research work in this field and made significant progress in several aspects.

First, OSR group developed a novel parser, which is called Sweble [11], for Wikitext. Wikitext is the wiki markup language of MediaWiki, which is a wiki engine that drives Wikipedia and many other wiki instances. Apart from converting the content to hypertext markup language (HTML), Sweble also produces an intermediate format, namely, abstract syntax tree (AST) to represent the page’s content in a high-level and machine-readable way [11]. The data stored in AST can be easily queried and translated into arbitrary target formats. OSR has also integrated the Sweble parser into MediaWiki such that the user can take advantage of this alternative parser in the applications powered by MediaWiki [28]. Furthermore, in order to simplify and improve wiki content transformation and refactoring, OSR proposed a framework for defining and performing transformations on wiki article [12].

However, it is difficult to make use of the above components directly. They can play a role only when integrating to a wiki engine. Thus, the goal of this thesis is to develop and implement a wiki engine which connects the above mentioned components. Since the above components are developed in Java, the new wiki engine will be programmed in the Java as well. It is named as Sweble wiki engine due to the new wiki engine is supported by the Sweble parser. Besides integrating the novel components, the Sweble wiki engine also attaches a storage solution with revision control and adds a web frontend to build a fully functional wiki.

2 Research Chapter

2.1 Introduction

Wiki is essentially a type of web application that allows people to manage web pages collaboratively by virtue of a web browser [26]. All the visitors or members within a prescribed community have the right to create, edit or delete HTML documents. Whereas, the HTML pages in previous web application are only managed and updated by the website's masters [28]. Moreover, the web editors of wiki are not required to master HTML knowledge. Instead they are allowed to create or edit web pages through wiki markup language (WML), which is a simplified substitute to HTML. Eventually, the wiki application lets users not only read but also create or edit their own content in Internet [33].

A wiki engine (also called wiki software or wiki application) is a software that is designed to manage documents collaboratively and powers a wiki system described as above [33]. The first wiki system was 'WikiWikiWeb' which is developed by Ward Cunningham in 1994 [26]. During the past years, a great amount of wiki systems running with different kinds of wiki software have emerged. Accompany with the development of technology, wiki systems are playing more and more roles in people's work and daily lives. Currently, Wikipedia is the most famous wiki site with the slogan that *the free encyclopedia that anyone can edit* [33]. It already has a worldwide influence and was rated as one of the top ten most popular websites in 2007. The wiki engine of Wikipedia is called MediaWiki [3], which empowers the full-blown features of Wikipedia. Besides Wikipedia, MediaWiki is also widely used to manage many other wiki instances. Usually, each wiki engine defines its own WML. MediaWiki defines its WML as Wikitext. Users of Wikipedia edit content by using Wikitext.

Abundant research works have been done to push forward the technologies of wiki engines. Recently, Dohrn and Riehle [11] developed Sweble Wikitext parser, which generates an intermediate representation of Wikitext content. By using Sweble Wikitext parser, the content stored in wikis can be accessed by machined easily. Furthermore, they proposed wiki object model (WOM) in [12], which unlocks from the syntactic idiosyncrasies of Wikitext. With the help of WOM, further operations such as transformation and refactoring, can be performed in a standardized and simplified way.

The object of this thesis is to design and implement a web application, which combines the above components and realizes the basic wiki functions, e.g., search, view, edit, revision control, etc. In order to achieve this object, several tasks need to be accomplished. First, suitable technologies and frameworks have to be adopted to ensure a modular and extensible design. Second, a storage solution has to be designed to enable the content retrieval, searches over the whole wiki and revision control. Third, a web application should be implemented as a whole that connects and integrates the above mentioned components and the storage solution.

The main contributions of this thesis include:

1. Implementation of Sweble wiki engine which makes use of Sweble parser and empowers the basic wiki functions

2. Establish a well-structure software architecture to facilitate the further extensions in the future.

The structure of this thesis is organized as follows. The related technologies are introduced in section 2.2. The requirements of Sweble wiki engine is analyzed in section 2.3. To achieve these requirements, the implementation process is described in section 2.4. Section 2.5 summarizes the research results and the results are discussed in section 2.6. Finally, the conclusions of this thesis are drawn in section 2.7. Chapter 3 includes necessary elaboration of research chapter. Section 3.1 introduces the software frameworks which are used in the development process. The supplementary explanation of the concrete implementation on presentation layer and persistence layer are presented in section 3.2 and section 3.3 respectively.

2.2 Related Work

2.2.1 Separation of concerns

In order to simplify the problem analysis and software design, the separation of concerns (SoC) strategy is adopted throughout the development of Sweble wiki engine. In software engineering, SoC is one of the important strategies introduced by Dijkstra [7] and Parnas [22] to simplify the complexity in software design. By means of SoC, a global software system is decomposed into separated and correlated modules, so that each module addresses its "concern" or responsibility respectively and transparent to each other. As a result of SoC, the individual modules are combined to work collaboratively to achieve the original goal.

Through proper SoC, a well-organized system can be built and better maintenance and reusability can be achieved. Thus, various techniques are applied for the purpose of SoC within the development of Sweble wiki engine. For example, Java is chosen as the programming language. Its object-oriented nature can separate concerns into objects. Furthermore, the whole project is basically divided into three layers to play different roles in the system [16], which will be introduced in section 2.2.3. In particular, cross-cutting concerns such as transactions, security, logging, etc. are difficult to be separated by object oriented techniques, since they are scattered throughout multiple objects in the whole system. Aspect-orientated programming (AOP) [14] is a programing pattern that proposed to separate cross-cutting concerns from the core concerns. The interception-based approach of AOP is applied in the process of transaction management and access control in my project. In addition, Apache Wicket framework is selected for the web application development in my project. It separates the web page construction into Java code and HTML markup, which leads SoC to a new level.

2.2.2 Client server architecture

Since Sweble wiki engine is a web application, it is installed in a network environment. How to deploy the web application in the network environment directly impacts the concrete software design. First of all, it is decided that Sweble wiki engine is run in the form of client-service architecture, which is a widely used coordination pattern in distributed systems [1]. In this architecture, the client and server are separated physically. The client requests service from the server, while the server plays a role as centralized resource, which performs service according to the request and returns response to the client. In my case, the client is a browser of any user, which is responsible to send request to the server, receive response and present the content to the user. While the server hosts the software of Sweble wiki engine to provide service to multiple clients.

2.2.3 Three layer architecture

Based on the above analysis, the development of Sweble wiki engine is to construct a server-side application. It is commonly built by layering techniques, which separates the complicated system into multiple layers according to different responsibilities. Sweble wiki engine will be developed by using three layer architecture, which is a popular layering pattern [16]. It divides the system into three independent layers, which are presentation layer, logic layer and persistence layer. The presentation layer is used to provide user interface (UI) and handle the interactions with the client (or browser). The business logic layer contains a set of business rules for processing information. The back-end of the system is the persistence layer, which is mainly composed of the data persistence objects and data access mechanisms.

2.2.4 Dependency injection

The three layer architecture decouples the system loosely. However, I need to address another problem: how to wire up the layers? Traditionally, the objects create the dependencies in their code, but this pattern lacks of modularity. On the contrary, dependency injection (DI) claims that objects can ask an outsider to create the dependencies for them [24]. Thus, DI pattern is preferable for my project. The dependencies exist between the layers are fixed by DI pattern.

Dependency injection (DI) [24], which was coined by Fowler in 2004, is a specific variation of inversion of control (IoC) design pattern [17]. Its core idea is to manage object dependencies in this way: push (inject) dependencies into objects at runtime, instead of allowing the objects itself to pull (create or find) their dependencies from their environment [24]. In a DI framework, the dependencies are offered by a container and injected into the objects which need them. There are three forms to inject the dependencies, which are setter injection, constructor injection and method injection. Some open source DI frameworks are used widely, such as Spring[32] and Google Guice [19]. Google Guice is chosen as the DI framework in my project.

2.2.5 Object-relational mapping

The software of Sweble wiki engine is developed by using Java language. It will be integrated to an external relational database. There is a problem of object-relational impedance mismatch in this situation. The approach of object-relational mapping (ORM) [4] is proposed to fix this problem. In object-oriented programming, data management generally concerns the manipulation of objects but not scalars. However, most of the existing databases are relational databases, in which only scalar values can be stored and manipulated. By virtue of ORM, data in relational databases are mapped into Java objects, which further make up a domain model. The traditional ORM frameworks try to solve this problem from a stand point of Java. For example, Hibernate [25] maps Java classes to database tables and shields from the underlying SQL. As a result, its API is quite different from SQL and cannot support all the database features. In contrast, Java object oriented querying (jOOQ) [8] stands on the opposite side.

SQL is put in the first place in the database integration. The closeness of SQL enables jOOQ to support almost every features that database offers. Thus, jOOQ is chosen as the ORM framework in my project.

2.2.6 Domain specific language

Since jOOQ is characterized by its internal DSL, which simulates the SQL statement in Java. Before using it, it's necessary to explain the domain specific language (DSL) first. DSL, in contrast to a general-purpose language (GPL), is usually a ‘simple’ computer language designed to solve problems in a specific domain [18]. By properly using notations and abstractions, a DSL can make the task of programming easier. Besides that, the usage of DSL can also enhance the readability, maintainability and portability of a program.

The existing DSLs can be classified into the external and internal forms [18]. External DSL has its own syntax and domain specific tooling so that it is parsed independently. A representative example of external DSL is the plain structured query language (SQL). Internal DSL is built on an existing host GPL through fluent application programming interface (API), which is a particular form of API. For example, jOOQ embeds SQL into Java as an internal DSL, by which users can build type-safe SQL queries.

2.2.7 Sweble Wikitext parser

Because Sweble wiki engine will make use of Sweble Wikitext parser, it's necessary to introduce it in this section. The content of wiki system is generally written by the WML, which is thought to be simpler than HTML. However, various WMLs are nowadays getting more complex and without well-defined grammar [11]. A critical task of any wiki engines is how to parse the WML precisely such that the content can be converted to different formats as desired and further the generated results are well prepared for further operations.

However, the state of art of the parsing work is still far from a satisfactory level, e.g., the parser of MediaWiki directly converts Wikitext into HTML without generating any high-level representation [11]. This parsing approach may produce invalid HTML since it’s insufficient to overcome the complexity of Wikitext. Thus, it’s highly desired that a new parsing approach can be designed.

Recently, the Sweble Wikitext Parser is designed and implemented by Dohrn and Riehle [11]. Compared with the conventional parsers, Sweble parser is characterized by its ability to construct a clean intermediate representation of Wikitext content. As shown in figure 2.1[28], the Wikitext is first transformed into an AST, which is a well-defined machine-readable structure. Further, the AST can be converted into different kinds of output format, e.g., HTML, document object model (DOM), PDF, or even converted back to Wikitext.

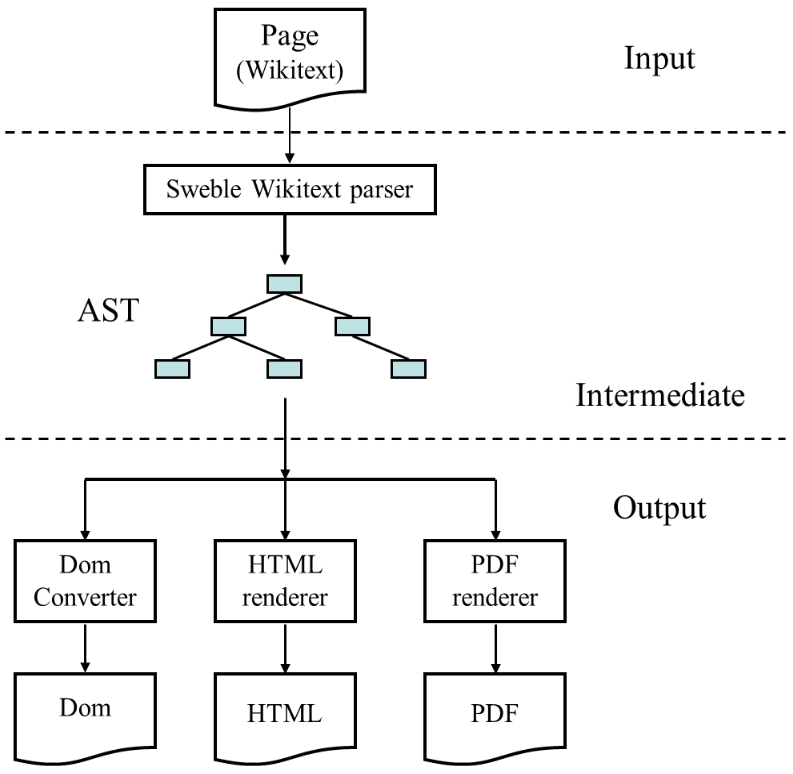


Figure 2.1: Sweble Wikitext parser processing pipeline

2.2.8 Wiki object model (WOM)

As mentioned before, by using AST, the Wikitext content is mapped into a well-structured object model so that the purely textual content can be accessed and queried precisely by software program. However, the nodes of above AST is defined according to the syntax of Wikitext, which means that it still couples with the syntactic idiosyncrasy of the original WML. In order to achieve a more universal object model that is applicable to all WMLs, Dohrn and Riehle proposed wiki object model (WOM) in [12], which is the generalized version of Wikitext object model as defined in [10].

WOM is also a tree-like structure with the nodes to represent the elements commonly found in all WMLs. The nodes are mapped into a set of Java objects in software programs and further serialized as WOM extensible markup language (XML) for storage. With the help of WOM, various WMLs can be represented in an easy, accessible and standardized way and meanwhile the WML-specific information is retained.

Some new features, such as data analysis, content transformations, have emerged into the world of wikis. They only require the semantic content of wiki articles but not the specific syntax of the original WML [10]. The significance of WOM is that it provides a standardized domain to such kinds of operations and unlocks them from the syntactic intricacies of the underlying WMLs. Through a standardized high-level representation of WOM, further operations are decoupled from the parsing process so that they can be designed and performed in a standardized way [12].

2.3 Research Question

As mentioned in Chapter 1, my task is to implement Sweble wiki engine which invokes Sweble parser, attaches a storage solution and adds a web frontend to build a fully functional wiki engine. Thus, the research questions of this thesis are:

1. What are the features of Sweble wiki engine?
2. How to implement Sweble wiki and how to implement it in an efficient way?

In order to well define the first question, I perform the requirement analysis of Sweble wiki in this chapter. The second question is related to selection of technologies and frameworks to implement Sweble wiki engine, which I will address in the next section.

For a clear explanation, first I introduce the concepts of "article" and "revision" in the context of wiki engine. A wiki article refers to a web page that contains title and content. Title represents a well-defined topic, while the content describes relevant information around the topic. As we know, articles can be continuously modified and improved by multiple users through wiki's collaborative editing function. Every time the modification is saved, a new version of content, i.e. a revision of article, will be generated. As a result, one article has many revisions, which have the same title but different content. The revisions can keep track of the revising history of the article.

2.3.1 Homepage

As a wiki engine, Sweble wiki engine should hold a home page, which is used as the initial portal of a wiki instance. This page presents the basic information of the wiki instance website, such as its name, slogan, license, organization and brief introduction. In addition, the navigation menu is also included. In order to let the user access the articles that the wiki instance is holding, a search panel is provided in the homepage.

2.3.2 View articles

While the browser requests to view a wiki article, Sweble wiki engine should return the web page by converting the content from Wikitext to HTML. We should notice that one article has many revisions. The feature of "view an article" refers to returning the content of the most updated revision. Besides that, the user also have opportunities to view the content of other previous revisions through "Revision history" feature, which will be mentioned later. In the case of no content is available to be displayed, a feedback message is returned instead.

2.3.3 Edit articles

The principle feature of the wiki engine is to allow users collaboratively edit the content of a web page online. Besides changing the content, users can also structure or format wiki pages through editing function. Usually, users edit wiki pages with simplified WML. Some engines also provide visual editors to simply their work.

Sweble wiki engine also enables the user to edit a wiki page when he is reading it. If the user shifts to a page for editing, he can edit the content in Wikitext directly. At the beginning, the content of the latest revision is shown in the text edit box so that the user can continue his editing based on the predecessor's work, which facilitates the collaborative working. In the process during editing, the user can preview his input when needed. He can also choose to save his input as a new revision in the database. After that, this new revision is displayed to the user so that he can check it in time.

2.3.4 Revision history

As mentioned before, the articles may undergo modifications frequently. One useful feature of Sweble wiki engine is to summarize the revision history for any article. By virtue of the list of changes, users can trace the overall evolution history of such article and evaluate whether the current revision is acceptable. Furthermore, it also enables the user to revert to an older revision, which is helpful if the latest revision is destroyed or any mistake is found. In the page that lists revision history, each revision is expressed as the information of its creation time and author. The creation time is rendered as a link that directs to the view page of this revision. The user can choose to view any specific revision content by clicking one of the links.

2.3.5 Search articles

To facilitate the user to search wiki articles according to their interests, Sweble wiki engine also provides search function. As long as the user inputs keywords and clicks the search button, Sweble wiki engine connects to the database and searches the keywords both in titles and the contents of the most updated revisions. If some matching articles are found, their corresponding titles are returned back. More specifically, every title is rendered as a link, which directs the user to view the concrete wiki article. Otherwise, if no article matches the keywords, a feedback message is presented to indicate that no article is found.

2.3.6 Create articles

If the user wants to raise a new topic, he can create a new wiki article through Sweble wiki engine. There are two steps to accomplish the creation of an article. The first is to create a new article title which must not duplicate to the already existing titles. If the title is duplicate, the user will be informed and asked to view the existing article. If the title is created successfully, the second step is to add article content. This step is similar to the feature of "edit article" mentioned before. The only difference is that the user has to start the first revision without previous content for reference.

2.3.7 Hard deletion and soft deletion

Sweble wiki engine also provides deletion features to help wiki instances to handle the inappropriate contents. If the wiki article involves in copyright violations, vandalism, or law-breaking issues, it should be deleted from database, which is called hard deletion. Compared with this irreversible deletion, Sweble wiki also supports reversible deletion, which is the so-called soft deletion. It is used in the situation that the article contains some inappropriate contents that can be improved by adding new sources. Soft deletion means the article is deleted logically but still existing in the database physically. If the article is softly deleted, all the revisions of this article are hidden from the user. It can be later undeleted if the inappropriate contents are fixed. The soft deletion concept can be also applied on revisions. If some revisions are softly deleted, they will not appear in the revision history.

2.3.8 Access control

Wiki opens the web page creation and editing to users, which facilitates collaborative working. However, meanwhile, it also leads the potential risks of vandalism or abuse. Thus, how to manage the user behaviors effectively becomes an issue that wiki engines have to address. Sweble wiki engine provides an access control policy to restrict creation and editing of wiki articles. More precisely, it's a role-based access control (RBAC), in which the different behavior permissions are associated with roles [15]. Users are separated into groups by assigning different roles, which stands for different levels of access.

a. Access control policy

In Sweble wiki engine, there are five kinds of roles, which are super admin, admin, moderator, editor and viewer. Their privileges follow the role-based inheritance, which means the editor inherit the privileges from the viewer and adds its own extra privileges, further, the moderator inherits the privileges from the editor and adds some extra privileges, and so forth. As a result, the super admin has the highest privilege whereas the viewer has the lowest privilege.

Only the creation and editing features are restricted, which require the user at least has the role of editor. Other features, i.e., search, view and revision history, are open to all users. In order to identify who the user is, and further identify his role, the registration and login procedure are necessary.

b. Registration

Registration is used to keep a record of user information. Specifically, the registration page asks the user to fill in the information of username, password, confirm password and email address. The inputs need to satisfy some preset conditions as follows: the username and email address should be unique; password should be at least six characters long; password and the confirm password should be the same. If the inputs fulfilled all these conditions, they can be submitted. Afterwards, Sweble wiki engine sends a confirmation email to the email address provided by the user. Consequently, the user data is saved in the database. By default, the registered user is assigned as the role of editor. The administrators have the right to modify the roles of users later.

c. Login

If the user wants to edit or add articles, he has to log in first. The login page requires the user enter user name and password. If the inputs match one of the registered users' information, the user is logged in successfully. Otherwise, the login page stays and indicates the failure reason to help him correct his credentials. Next, the role of the user is checked. If the user owns the role more senior than editor, his intended action is allowed.

d. Show authentication status and logout

Sweble wiki engine also provides a panel to indicate the user's authentication status. At first, the "Register" and "Login" links is displayed, which link to the registration and login pages respectively. After the user is logged in, his username is displayed instead. In addition, the "Logout" link is displayed alongside. If the user clicks "Logout" link, he will be logged out and redirected to the home page.

The above functional requirements are the targets of the implementation. Besides that, the well-structured software architecture and appropriate selection of libraries also need to be taken into consideration in the implementation process. The implementation of Sweble wiki engine is discussed in the next section.

2.4 Research Approach

The functional requirements of Sweble wiki engine are proposed as the research questions in section 2.3. In the research approach section, I will describe the software implementation to fulfill the above requirements.

2.4.1 Software architecture

For the purpose of loosely decoupling and SoC, three layer architecture, which is introduced in section 2.2.3, is taken as the software architecture of Sweble wiki engine. It can bring about better maintainability, reusability, and flexibility for the future development.

Figure 2.2 shows the software of Sweble wiki engine is separated into three layers with their own responsibilities, which are presentation layer, logic layer and persistence layer [16]. From the top down, every layer is only dependent on the layer below but not able to see the upper layers. The presentation layer takes care of web UI, which contains all the web pages as well as their dynamic behaviors. Usually, the logic layer aggregates data from persistence layer, carries out the calculations, and then reports the results to the presentation layer. The persistence layer is the back-end of the logic layer. Data is retrieved from databases and converted to Java objects in this layer and then passed back to the logic layer. Besides that, an external database is used to store relevant data, such as wiki articles, revisions and user information. Moreover, several Java libraries are chosen to further simplify the implementation. Apache Wicket [2] is be used as the framework for developing web UI; JOOQ [8] is the ORM framework to for integrating the persistence layer and the database; Spring framework [32] is responsible for transaction management; Google Guice[19] is applied to wire up the layers. The implementation of every layer is introduced in detail as follows.

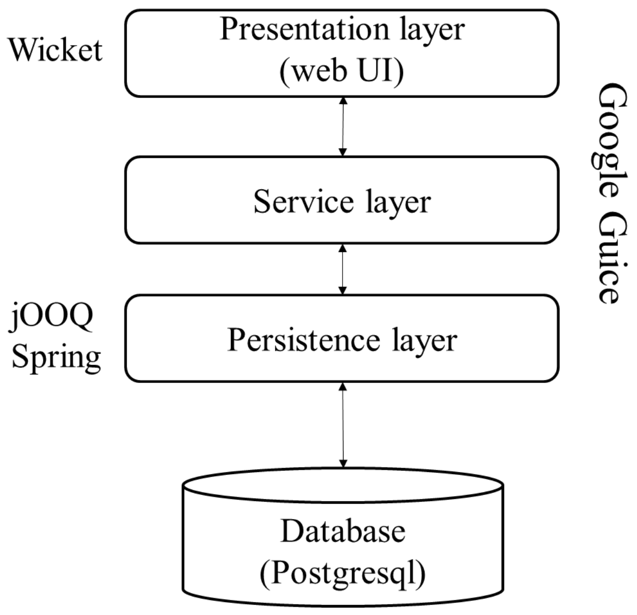


Figure 2.2: Three layer architecture of Sweble wiki engine

2.4.2 Implementation of presentation layer

Apache Wicket framework is chosen to facilitate the implementation of presentation layer [31]. It is a lightweight component-based web application framework. It makes it possible to develop web applications using Java, which satisfies the original intension of Sweble wiki engine. In addition, Wicket separates the web page development into writing just Java codes and just HTML markups which helps to separate the concerns of logic and presentation properly [5]. It also provides meaningful Java abstraction for web page components as well as the context of web application. Wicket leaves me out of the complicated underlying technology so that I can focus on the logic implementation [5]. Thanks to the object-oriented nature, Wicket makes it easy to create reusable components. The concrete introduction of Wicket will be presented in section 31.1.

In this layer, multiple web pages with dynamic behaviors are implemented to fulfill the requirements mentioned in section 2.3. For example, the view page is implemented for the view feature, the search page is for the search feature, the registration and login pages are for the access control, and so on. The concrete implementations of these pages are elaborated in section 3.2.

2.4.3 Implementation of logic layer

This layer plays a role as a service provider for presentation layer. In Sweble wiki engine, three service classes are created in logic layer. Firstly, the features, such as view, edit and search articles, involve retrieval, store and query data in database. Thus, the DBService class is created to perform specific aggregations, calculations or other operations on the data objects in persistence layer according to the requirements from the presentation layer. Secondly, the ParserService class is created to make use of Sweble parser to parse the input content to AST. Then the nodes of AST are rendered into HTML format. Thirdly, as mentioned in section 2.3.8, Sweble wiki engine needs to send a confirmation email to the user after registration. Therefore, the EmailService class is created to implement email sending process.

Furthermore, to achieve better flexibility and independency, the service classes in logic layer only expose interfaces to the classes in presentation layer. The interfaces are backed by their concrete implementations.

2.4.4 Implementation of persistence layer

Persistence layer works as a data transporter between the logic layer and the database. Typically, this layer concentrates on the ORM problem, which is mentioned in section 2.2.5. In order to fix this problem, jOOQ is selected, which is an emerging lightweight ORM software library [8]. The concrete introduction of jOOQ will be given in section 3.12. Here, I only introduce its characteristics in brief. JOOQ is characterized by its internal DSL, which is mentioned in section 2.2.6. With jOOQ, developers can write SQL in Java code just as it was natively supported. Besides that, another useful characteristic is that jOOQ can reverse-engineer the database schema to generate the mapping Java classes by some configuration works [9]. The concrete code generation process through configuring the project object model (POM) file in Maven will be introduced in section 3.3.1. As a result, some Java classes, which model tables, records, plain old Java objects (POJOs) and data access objects (DAOs) are generated to assist the data access in database. In order to make use of this characteristic, the database schema should be designed first. Next, the database schema of my project is introduced.

For Sweble wiki engine, wiki articles, revisions, and user information are the three most important data that needs to be stored in an external database. As explained in section 2.3, in the circumstance of wiki, one article may have many revisions, which have the same title but different contents. To map this relationship to the design of table schema, an article table is created to save the title and a revision table is created to save the different version of content. Specifically, besides the title and content, these two tables also save other related metadata respectively. For example, each article record always marks its latest revision while each revision record stores some metadata, such as its creation time, author and previous revision, which can be used to restore the revision history. In particular, to enable the soft deletion feature mentioned in section 2.3.7, a flag which indicates whether this record is softly deleted, is added both to article and revision tables. In addition, for the sake of access control, a user table is created to save the information about users, i.e., the registration information and the assigned role.

Since jOOQ lacks transaction management mechanism, Spring framework is also used in this layer to remedy this problem. The transaction management of Spring will be introduced in section 3.3.2.

2.4.5 Implementation of dependency injection

In three layer architecture, dependencies exist between two connected layers. For instance, the objects in presentation layer depend on the objects in logic layer. DI pattern, which is introduced in section 2.2.4, is selected to manage the dependencies between layers. In DI pattern, objects ask an outsider to create and inject dependencies for them, rather than create dependencies by themselves. In this way, objects in presentation layer don't need to take care of which implementation is used and also how it is obtained.

In the development of Sweble wiki engine, Google Guice is chosen as the DI framework, which will be introduced in detail in section 3.1.3. Google Guice works as a central controller of dependencies, which means it takes charge of which dependency is created and how many dependencies are needed. In Google Guice [30], the object relationships are configured in a Guice module. SwebleWikiModule is the Guice module for the project of Sweble wiki engine, which is configured as figure 2.3. It tells Guice that IDBService is instantiated as DBServiceImpl when it's used. Similarly, IParserService is instantiated as Wom3NodeParser and IEmailService is instantiated as SimpleEmailSender. In the future, if the interface is bound to another implementation, it's only necessary to change the configuration in this module. After SwebleWikiModule is configured, the *@Inject* annotation can be used in the objects in presentation layer, which indicates Guice to inject the dependencies.

```
final class SwebleWikiModule extends ServletModule
{
    @Override
    protected void configureServlets()
    {
        install(new TransactionModule());

        filter("/*").through(WicketFilter.class,
            createWicketFilterInitParams());

        bind(WicketFilter.class).in(Scopes.SINGLETON);

        bind(IDBService.class).to(DBServiceImpl.class).in(Singleton.class);

        bind(IParserService.class).to(Wom3NodeParser.class);

        bind(IEmailService.class).to(SimpleEmailSender.class);
    }
}
```

Figure 2.3: GuiceModule for Sweble wiki engine

2.5 Research Results

As a research result, the software of Sweble wiki engine realizes the requirements listed in section 2.3. The software implementation is structured into three layers, which are presentation layer, logic layer and persistence layer (cf. Figure 2.4).

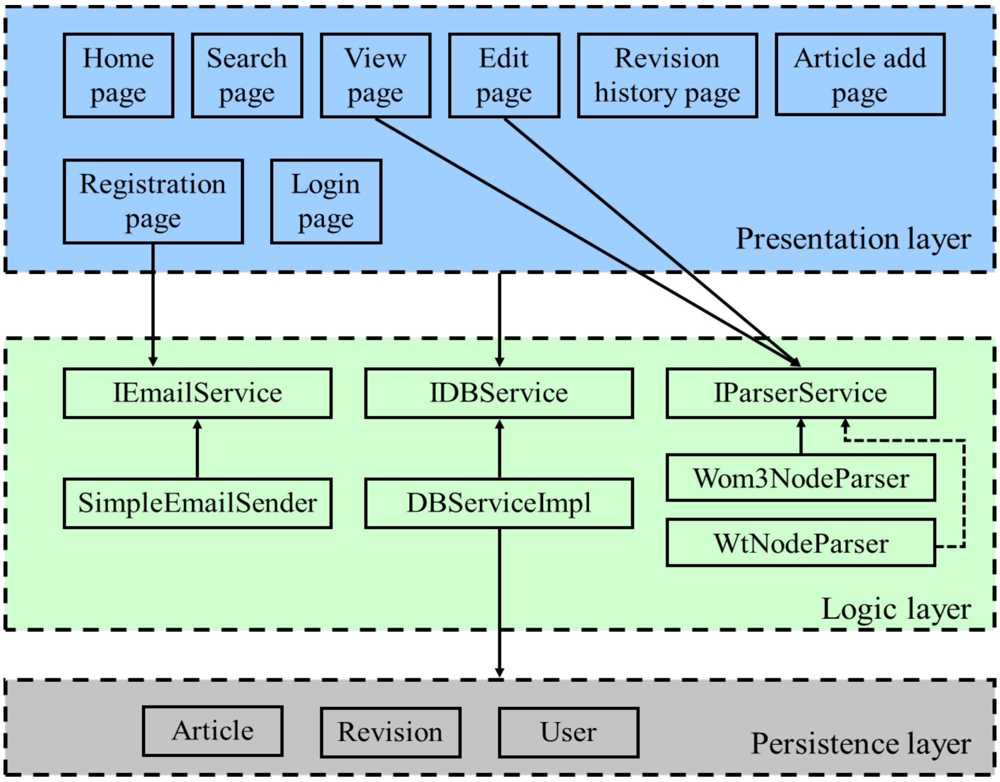
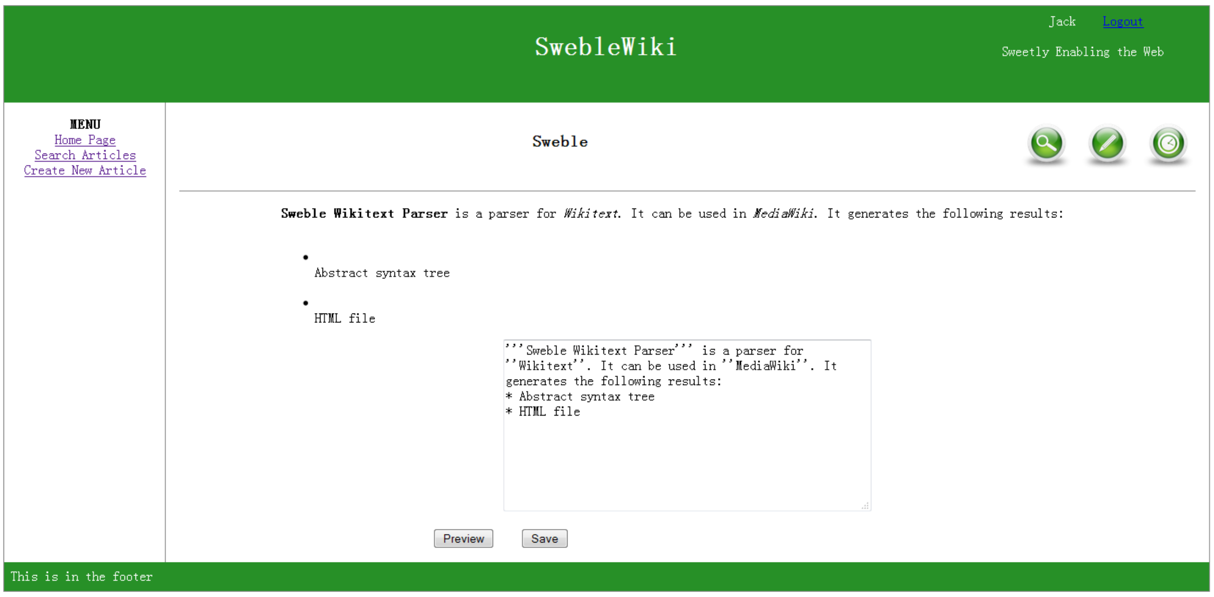


Figure 2.4: Software architecture of Sweble wiki engine



Sweble Wikitext Parser is a parser for *Wikitext*. It can be used in *MediaWiki*. It generates the following results:

- Abstract syntax tree
- HTML file

```
'''Sweble Wikitext Parser''' is a parser for
''Wikitext''. It can be used in ''MediaWiki''. It
generates the following results:
* Abstract syntax tree
* HTML file
```

Figure 2.5: Screenshot of edit page in Sweble wiki engine

In the presentation layer, home page, view page, edit page, revision history page, search page, article add page, registration page, and login page are implemented to achieve the features proposed in section 2.3. For example, the screenshot of edit page is shown in figure 2.5. In this page, the user can edit the content of wiki article online. In particular, the user can choose to preview or save the content via the "Preview" and "Save" buttons. More implementation results will be shown in section 3.3.2.

The dynamic logic of the above pages are supported by service objects in the logic layer. For instance, the registration page requests IEmailService to send an email to the user in order to finish the registration. The view page and the preview function in the edit page request IParserService to translate the content to HTML for display. In particular, since Sweble parser can generate AST with Wikitext nodes or WOM nodes, which are introduced in section 2.2.7 and 2.2.8. Different HTML renderers are needed to match different kinds of nodes. Thus, the interface IParserSevice has two implementations, which are WtNodeParser and Wom3NodeParser. Either of them can be chosen according to the needs. In addition, all the pages request IDBService to query, store or retrieve data in the database. The implementation of IDBService makes use of the data objects from persistence layer to provide the storage service.

There are three kinds of data objects, which are article, revision and user, in persistence layer. All the Java classes, such as tables, records, POJOs and DAOs, are generated by using the source code generation function of jOOQ.

2.6 Results Discussion

The results discussion in this section is performed in two aspects, which are extensibility of the software implementation and the comparison of different solutions of content storage.

2.6.1 Discussion on extensibility

As mentioned before, the implementation of Sweble wiki engine achieves the predefined functional requirements. Besides that, the clear structure and modularization of the implemented software can be regarded as another research result since it creates convenience for the further extension.

A simple example is taken to show the extensibility of the implemented software. If a new feature is raised, the further developer need to add a new page object in the presentation layer firstly. According to the requirements, new service objects are added to the logic layer. If the feature requires some database related functions, new methods should be added in IDBService and DBServiceImpl classes. Furthermore, if the new methods involve extra data objects or extra information of the existing data objects, the further developer have to modify or extend the database schema and then use jOOQ to update the data objects in the persistence layer. We can see that the three layer architecture helps the further developer to perform extension task in a clear way.

In a word, the high extensibility thanks to the appropriate selection of technologies and frameworks. The attempts of SoC are applied throughout the development process. Three layer architecture leads a clear software structure, which facilitates the extension and troubleshooting. Wicket separates the logic and presentation layout in a clean way, which also simplifies further development. DI centrally manages the dependencies which are distributed in the project. Guice brings compilation checking to the dependency configuration process [19]. In virtue of Guice, dependencies can be changed or added easily without influence of the source code. The source code generation feature of jOOQ reduces the workload for the modifications in persistence layer.

2.6.2 Discussion on content storage

The characteristic of Sweble wiki engine is the usage of Sweble parser. Through Sweble parser, the representation of wiki content is converted from Wikitext to AST/WOM and further rendered into HTML. It produces a problem: which representation is the best choice for the content storage? To answer this problem, the strengths and weaknesses of these three representations are compared in this section.

a. evaluation criteria

To make a comprehensive comparison, the performance impact of choices on various common operations are taken into account, such as rendering, software processing, and system migration. Consequently, the evaluation is conducted according to the following four criteria.

1. Restoration of the original information

The retrieved data should have the capability to restore the original user input content. Otherwise, it means that some original information is lost.

2. Complexity of HTML generation.

In order to display wiki content via web browser, the stored data needs to be converted to HTML file. This process should be keep as simple as possible.

3. Machine accessibility

As mentioned in [11], a machine-accessible representation is a prerequisite for efficient software processing on wiki content.

4. Ease of data migration

The database storage of wiki content is independent of the wiki software. It's possible to migrate the storage data to different wiki engines or wiki software. If the representation is not WML dialect-specific, it can be processed by multiple software in a standardized way [12]. The data migration is straightforward and easy under this situation. Otherwise, the data migration needs a lot of extra processes.

b. Comparison of different representations

In this part, the storage solutions of Wikitext, WOM/AST document and HTML are compared based on the above criteria.

Wikitext:

1. It preserves the original information completely.
2. The conversion of Wikitext to HTML needs a complicated parsing process [11].
3. It is a purely textual representation, which is extremely difficult to be processed by computer program.
4. It requires specific wiki engine or at least specific parser. If the data is migrated to a wiki engine that is unaware of its markup, an appropriate parser must be added.

WOM/AST document:

1. The rigorous specification of WOM filters out some invalid Wikitext content, which implies the possibility of losing original information. To remedy this problem, WOM

uses round-trip data to fully preserve WML-specific information.
2. It is convenient to generate HTML through traversing the AST/WOM.
3. It is machine-accessible.
4. WOM/AST is independent of syntactic idiosyncrasies of WML so that it is easy to migrate the data when it is saved as WOM/AST document.

HTML:
1. It's possible that HTML loses the original information heavily if the parsing process is error-prone. In addition, it also misses the syntactic idiosyncrasy of the original WML. To re-engineer HTML back to the original WML is a non-trivial process [11].
2. It can be directly rendered by web browser.
3. It is machine-accessible.
4. The data migration is easy owing to the standardized specification of HTML.

In conclusion, WOM/AST document is the best choice for storage owing to its fully restoration of original information, easiness of HTML generation, machine accessibility and easiness in data migration. However, due to the time constraint I cannot implement the storage solution in WOM/AST any more.

2.7 Conclusions

In this thesis, I have implemented a web application, which integrates Sweble parser, attaches a storage service solution and a web frontend to empowers the basic wiki functions, e.g., search, view, edit, show revision history, etc. At the same time, the proper selection of technologies and frameworks guarantees the software is extensible in the future.

Based on the above discussion, AST/WOM document is the best choice for storage. In the future, the wiki content can be saved as AST/WOM document instead of Wikitext. The change of storage format leads a problem of database selection. The further developers need to consider which kind of database is most suitable to store the tree structure of AST/WOM. With the right choice, it can achieve high performance on data retrieval, query and update. Besides that, Sweble parser is the main asset of Sweble wiki engine. Therefore, the engine can be extended by adding some powerful functions which are supported by Sweble parser, for instance, such as data analysis, transformation and refactoring, visual editor and semantic wikis.

3 Elaboration of Research Chapter

This chapter mainly focuses on the implementation of Sweble wiki engine. In order to make the implementation description easy to understand, section 3.1 introduces the software frameworks by several simple examples. In section 3.2, the concrete implementation of presentation layer is elaborated. The section 3.3 presents the concrete implementation of persistence layer.

3.1 Introduction of Software frameworks

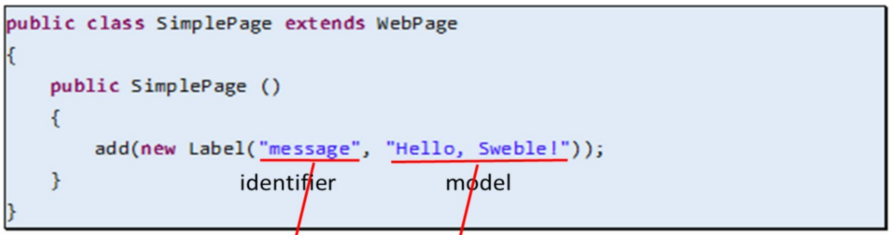
3.1.1 Introduction of Apache Wicket

Apache Wicket is an open source project of Java web application framework started in 2004 by Jonathan Locke [2]. It has merits of simplicity, separation of concerns and ease of development. By means of Wicket framework, web applications can be developed through using regular object oriented (OO) Java programming.

Wicket is a component based framework. Web pages are constituted of web UI components [5]. In order to illustrate how a web page is created by using Wicket, a simple example is shown in figure 3.1. In Wicket, every web page is created by a pair of Java and HTML files, which should have the same name. In the example, SimplePage class (cf. figure 3.1(a)) is the Wicket page component and SimplePage.html (cf. figure 3.1(b)) holds the markups for the SimplePage class. In particular, a Label component, which is provided by Wicket, is added to SimplePage as its child component. The Label component has an identifier “message”. Wicket automatically matches it to the associated markup, *<h1>* fragment, because it has an attribute *wicket:id = “message”*. Furthermore, the Label component attaches a model to yield a string “Hello, Sweble!”. The content of *<h1>* element is replaced by this model. As a result, the browser will receive a HTML file with *<h1>Hello, Sweble!</h1>* as its body (cf. figure 3.1(c)).

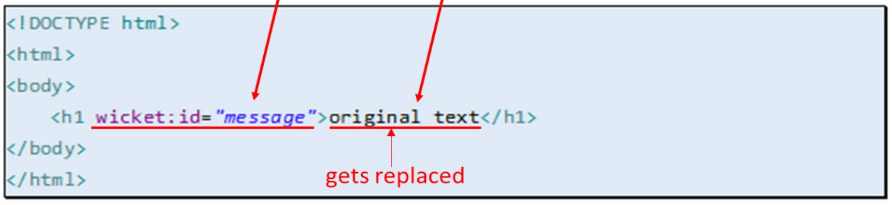
The component-oriented development in Wicket achieves some benefits. Firstly, Wicket provides meaningful abstractions for all the visible widgets (like buttons, text fields, and links, etc.) and the context of web application (like applications, sessions, pages, etc.) so that the workload is reduced greatly [6]. Wicket lets the developers focus on the logic implementation of the components without considering the underlying technology, e.g., hypertext transfer protocol (HTTP). In addition, the object oriented nature of Wicket enables the developers to reuse the components easily [31].

Secondly, the separation of logic and presentation in Wicket takes the separation of concerns to a new level [5]. Generally, Wicket component is responsible for the logic aspect. It describes the behaviors of the web application on the fly, such as form handling, dynamic content processing and so on. On the other hand, the HTML file takes charge of the static presention of components. Wicket requires the HTML file should not contain any script or logic but only the clean markup code and some placeholders for the components.



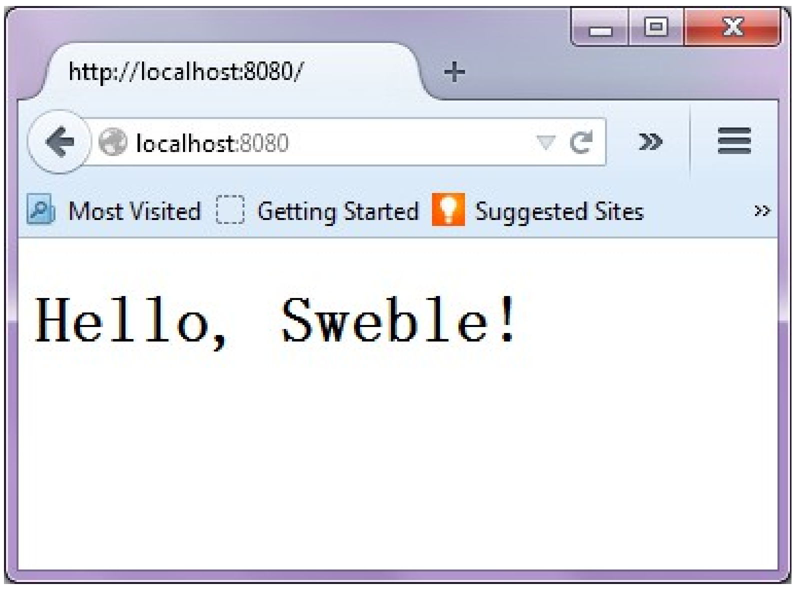
```
public class SimplePage extends WebPage
{
    public SimplePage ()
    {
        add(new Label("message", "Hello, Sweble!"));
    }
}
```

(a) SimplePage



```
<!DOCTYPE html>
<html>
<body>
    <h1 wicket:id="message">original text</h1>
</body>
</html>
```

(b) SimplePage.html



Hello, Sweble!

(c)

Figure 3.1: Simple example of Wicket application: (a) SimplePage; (b)SimplePage.html; (c) final HTML page rendered via browser

3.1.2 Introduction of jOOQ

JOOQ is an open source ORM software library developed by Data Geekery GmbH [8]. It proposed an innovation solution to fill the gap of interaction between Java data types and SQL data types. JOOQ lets the software developers get back in control of the SQL. With the help of jOOQ, it’s easy to build type safe database queries through its fluent API and obtain the Java codes from the database schema. JOOQ provides an internal DSL to simulate the SQL statement in Java. JOOQ API looks very similar to SQL [9]. For instance, if we want to search all authors who are born in 1970 and order them by their last names, the plain SQL statement is showed in figure 3.2 (a) and the version of jOOQ fluent API in Java is showed in figure3.2 (b).

```
SELECT * FROM AUTHOR
WHERE AUTHOR.BORN_IN = 1970
ORDER BY AUTHOR.LASTNAME
```

(a)

```
create.selectFrom(AUTHOR)
      .where(AUTHOR.BORN_IN.eq(1970))
      .orderBy(AUTHOR.LASTNAME)
```

(b)

Figure 3.2: Comparison of plain SQL and jOOQ fluent API: (a) Plain SQL; (b) JOOQ fluent API in Java

We can see that, with jOOQ DSL, the developers can write SQL statement in Java just as it was natively supported. Besides that, jOOQ can also take advantage of the Java compiler to conduct compile-check on SQL codes, including checks for column type, row value expression type and SQL syntax [9]. The closeness of SQL enables jOOQ to support almost every features that database offers. Furthermore, jOOQ also can be used as a standalone typesafe SQL builder and executor. On top of these capabilities, the feature of source code generation is developed and it becomes one of the jOOQ's powerful assets. Using jOOQ code generator, the database schema is reverse-engineered into a set of Java classes. Once the Java compiler detects some changes of database schema, it throws compilation errors [9]. The process of source code generation of jOOQ is presented in section 3.3.1.

In addition, jOOQ still provides features like traditional ORM frameworks, such as standardization of SQL dialects, active records and so on. It simplifies the process of Java application and relational database integration so that the developers can focus on their business logic [8].

3.1.3 Introduction of Google Guice

Google Guice is an open source software framework for supporting DI. It is developed by engineers in Google since 2006 and released under the Apache License [19]. It is the first generic framework which handles DI using annotations to configure Java objects. With Guice, developers don't need to build factory classes or use *new* in Java code to create dependencies, Guice's *@Inject* is able to inject the dependencies instead [30].

In this section, a simple example is taken to illustrate how to realize DI pattern by using Guice framework. Let's assume that an interface IStorageService is created and it is backed by the concrete implementation DatabaseStorageService. Figure 3.3 (a) shows a DI way to implement StorageClient class, which is the client of IStorageService. StorageClient accepts its dependent interface IStorageService in its constructor. This constructor injection is a good option because it makes StorageClient independent, i.e., it doesn't need to take care of which implementation is used and also how it is obtained. However, when a new StorageClient is constructed, its dependency IStorageService should be instantiate first. Guice plays a role in this situation. The *@Inject* annotation is written in front of this method to ask Guice to inject the service variable. If Guice detects the annotated constructor, it takes in charge of the instantiation of its dependencies.

Obviously, we need to tell Guice which implementation should be used to map the interface. In order to configure the mapping, a concrete implementation of Module interface needs to be created. Specifically, if StorageClient is constructed using DatabaseStorageService, the StorageModule is configured as figure 3.3 (b). In many other DI frameworks, the object relations are configured in an XML file, which is without compilation checking. In contrast, Guice module is a Java object using fluent and English-like method calls to describe the bindings, which is easy to master and can be checked in compilation [30].

Finally, an injector is created according to StorageModule in main method (cf. figure 3.3 (c)). The injector is used to get an instance of the bound class. After the injector is created, StorageClient can be constructed. If we want to store data in a file later, we only need to create FileStorageService to implement IStorageService and modify the configuration in StorageModule as figure 3.3 (d). Other codes stay the same. DI framework makes the code more flexible.

```java
public class StorageClient
{
    private final IStorageService storage;

    @Inject
    public StorageClient(IStorageService storage)
    {
        this.storage = storage;
    }

    public void saveData (String id, String data)
    {
        storage.store(id, data);
    }
}
```

(a)

```java
public class StorageModule extends AbstractModule
{
    @Override
    protected void configure()
    {
        /*
         * This binding tells Guice that when IStorageService is used in
         * dependency, it should be instantiated as DatabaseStorageService
         */
        bind(IStorageService.class).to(DatabaseStorageService.class);
    }
}
```

(b)

```java
public class StorageApp
{
    public static void main(String[] args)
    {
        Injector injector = Guice.createInjector(new StorageModule());
        StorageClient storageClient =
            injector.getInstance(StorageClient.class);
        storageClient.saveData("123", "Hello");
    }
}
```

(c)

```java
public class StorageModule extends AbstractModule
{
    @Override
    protected void configure()
    {
        bind(IStorageService.class).to(FileStorageService.class);
    }
}
```

(d)

Figure 3.3: Dependency injection using Google Guice: (a) StorageClient; (b) StorageModule; (c) StorageApp; (d) Modified StorageModule

3.2 Concrete implementation of presentation layer

The concrete implementation of presentation layer is presented in this section. This layer includes all the web pages that fulfill the requirements proposed in section 2.3. Since the dynamic logic of the pages are supported by the service objects from logic layer, the implementation explanation also contains the related methods from the objects in logic layer.

3.2.1 Basic layout of Sweble wiki engine

By using Wicket framework, every feature mentioned in section 2.3 is implemented as a Wicket page components. How to organize these pages together to achieve friendly user experience needs to be considered first. It's easy to find that reading, editing and revision history are the functions focus on one specific article. Users may want to shift among these functions frequently. For instance, the user perhaps wants to edit the content after reading. Or he sometimes wants to trace the revision history after reading or editing the article. To achieve better user experience, the portals to these three pages should be always provided in the title panel. On the other hand, searching and adding functions are not specific on one article. Their underlying processes are searching records and adding a new record in the database. They should be separated from the aforementioned functions. The links of homepage, search article and add article should be provided in the panel of navigation menu. Therefore, the basic layout of Sweble wiki engine is given as figure 3.4.

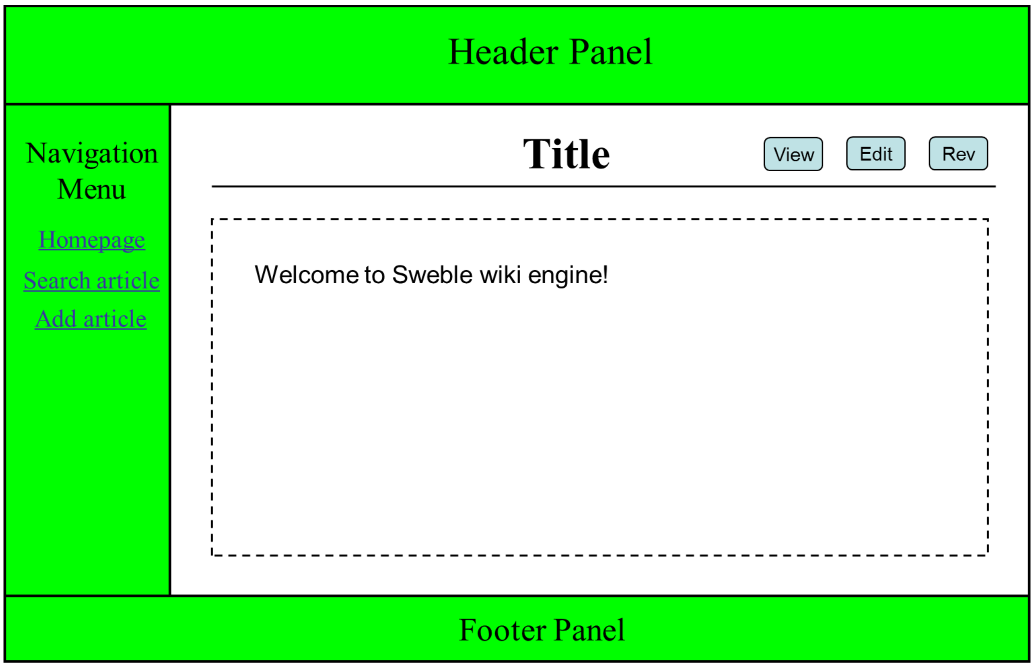


Figure 3.4: Basic layout of Sweble wiki engine

From the analysis of basic layout, we can see that some parts of the layout appear in all pages repeatedly. In order to reduce the duplication, I want to find a way to put the common parts together and let the common parts be reused as a whole. This intention can be realized by using the markup inheritance of Wicket. As show in figure 3.5, on one hand, with the Java *extends* keyword, both Page1 and Page2 classes can share the common codes in BasePage class. On the other hand, Wicket also enables the markup inheritance in HTML by introducing special tag of <*wicket:child*> and <*wicket:extend*> [6]. This feature lets the inheritance in markup be in light of the inheritance hierarchy in the Java classes.

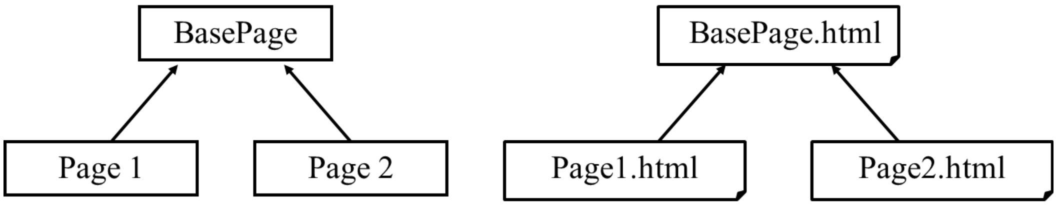

$$\begin{tikzcd}
 & \text{BasePage} & & & \text{BasePage.html} & \\
\text{Page 1} \arrow[ur] & & \text{Page 2} \arrow[ul] & \text{Page1.html} \arrow[ur] & & \text{Page2.html} \arrow[ul]
\end{tikzcd}$$

Figure 3.5: Markup inheritance in Wicket

In my project, BasePage is created firest, which includes all the common portions. Then all the pages in the application extend the BasePage with the help of markup inheritance in Wicket, as shown in figure 3.6. Consequently, every concrete pages shares the common layout of BasePage and adds its particular content in the child area.

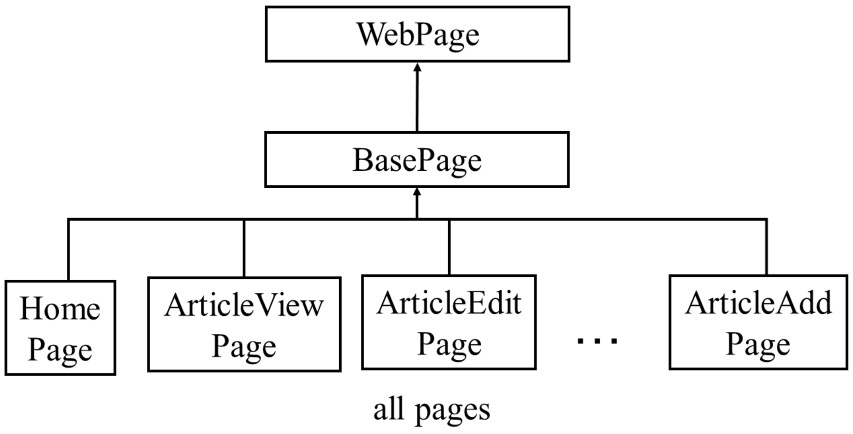


Figure 3.6: Page hierarchy of Sweble wiki engine in the initial stage

In Wicket, The complex page is usually constructed by several panels, which are usually used to group the components together as a unit and this unit can be reused in any pages (cf. figure 3.7) [6]. Therefore, the BasePage and other pages in my project are constructed by panels to achieve a well-organized and flexible structure.

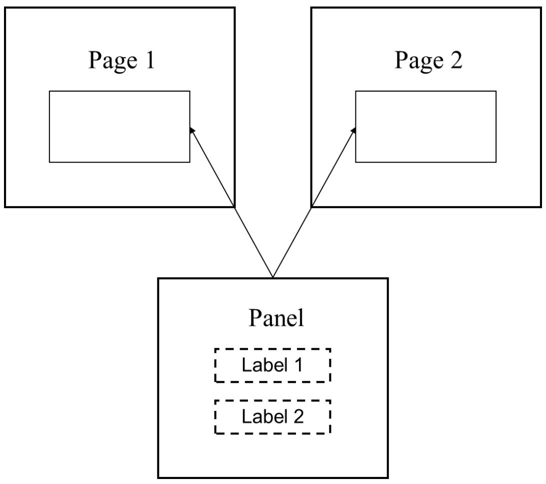


Figure 3.7: Reusability of Wicket panel

Figure 3.8 shows the screenshot of BasePage. It contains HeaderPanel, MenuPanel and FooterPanel. The HeaderPanel shows the title and slogan of a wiki instance. The UserPanel which indicates the user's login status will be later added to this panel. The MenuPanel comprises a group of direct links to homepage, article searching page and article adding page. They are implemented by using BookmarkablePageLink, which is a Wicket component and used to give a direct access to the internal pages of the application [5]. Finally, the relative information of the wiki instance such as license and organization can be displayed in the FooterPanel.

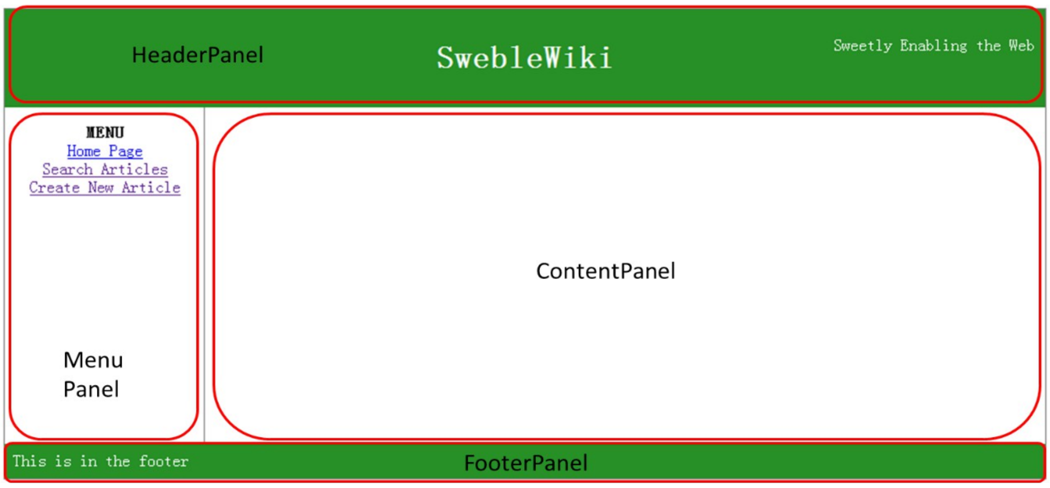


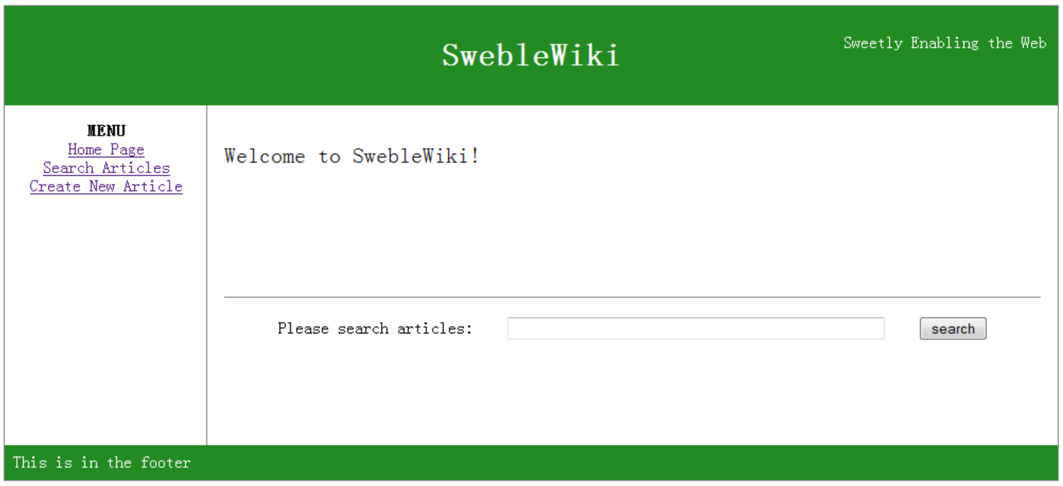
Figure 3.8: Screenshot of BasePage

3.2.2 Homepage

HomePage is the first page come to users. Besides the layout inherited from BasePage, it also contains an introduction of the wiki instance (cf. figure 3.9). Furthermore, a SearchPanel is added here so that the user can start his travel of the website.

a. SearchPanel

It contains one component of SearchForm. It is an instantiation of Wicket Form, which is used receive and handle the user inputs. In particular, SearchForm includes a text field and a button. The input keywords in text field are passed to SearchResultPage (cf. Section 3.2.6) if the "Search" button is pressed. SearchPanel is reused both in the Homepage and SearchResultPage.



Welcome to SwebleWiki!

Please search articles:

Figure 3.9: Screenshot of HomePage

3.2.3 View articles

ArticleViewPage is designed to display articles according to the requirements mentioned in section 2.3.2. It generally contains TitleViewPanel and ContentViewPanel. Sometimes, the NoContentPanel is applied to handle the special case that no content is available for this article.

a. TitleViewPanel

This panel contains the title and another three icons as shown in figure 3.10. The title is the identification of an article and the icons are the portals to the three basic functional pages, i.e., reading, editing and revision history. Since this panel is reused in the basic functional pages, it should always pass the identical page parameters to ensure that the latest revision is always returned if the user clicks the "View" icon.

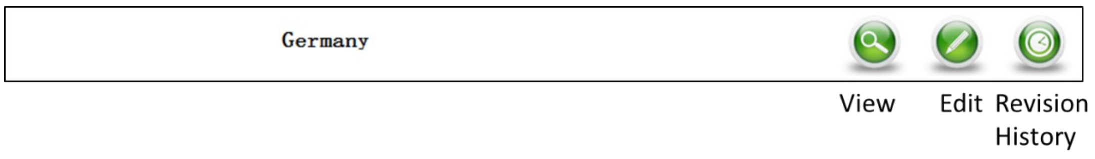


Figure 3.10: Screenshot of TitleViewPanel

b. ContentViewPanel

This panel is used to read the revision contents out from the database and convert them into HTML format. There are three ways to access ArticleViewPage: i) clicking the link in search result page (cf. section 3.2.6); ii) clicking the view icon in the TitleViewPanel (cf. figure 3.10); iii) selecting a specific reversion in the ArticleHistoryPage (cf. figure 3.14). Through the first two ways, the latest revision is shown, whereas a specific reversion (not necessary the lastest one) is displayed through the third way. The adaptability of showing different revisions is supported by passing and receiving different page parameters (cf. Figure 3.11). If the received page parameter contains a specific revision id. It is designated to the getContentByRevisionId method. Otherwise, the getLatestContentByArticleId method is

```
//if the received page parameters contains a specific revision id, assign
it to fetch a revision
if (!parameter.get("revisionId").isNull()
     && !parameter.get("revisionId").isEmpty())
{
    Long revisionId = parameter.get("revisionId").toLong();
    content = dbService.getContentByRevisionId(revisionId, false);
}
// otherwise, if the received page parameters doesn't contain any specific
revision id, assign it to fetch the latest revision
else
{
    Long articleId = parameter.get("articleId").toLong();
    content = dbService.getLatestContentByArticleId(articleId);
}
```

Figure 3.11: Pass different page parameters under different situations

invoked. In addition, as the content retrieved from the database is written in Wikitext. The renderHtml method in IParserService is called to convert the content into HTML format.

c. NoContentPanel

If no content is available for this article, the NoContentPanel is displayed just like figure 3.12. In order to encourage the collaborative editing, it suggests the user to edit the article. A link which connects to the ArticleEditPage of this article is provided. This panel is also used in the ArticleHistoryPage in the same case.

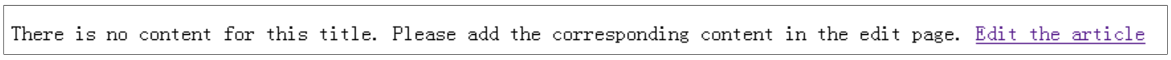

There is no content for this title. Please add the corresponding content in the edit page. Edit the article

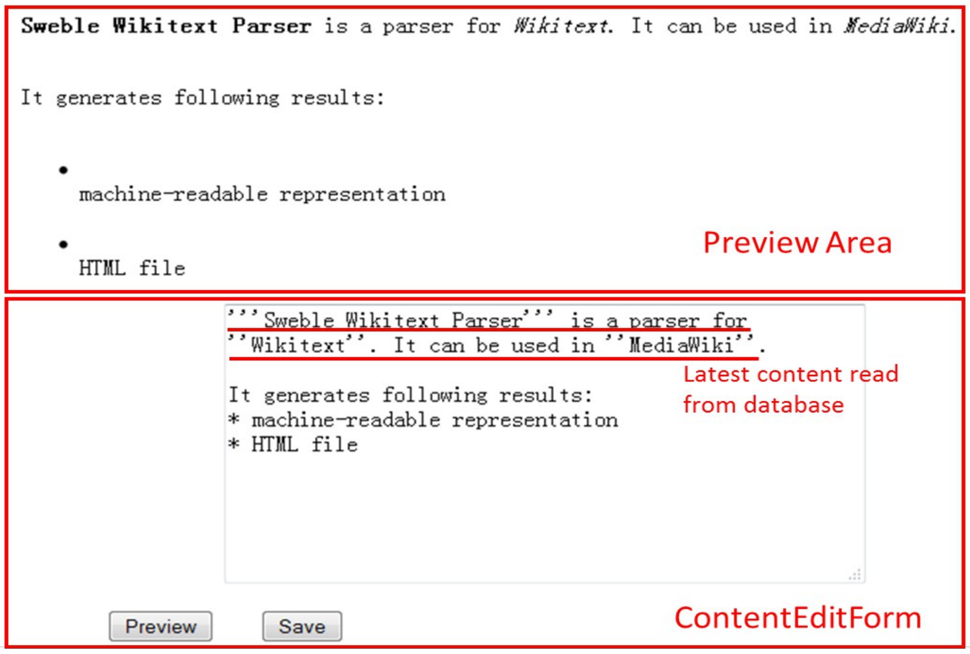
Figure 3.12: Screenshot of NoContentPanel

3.2.4 Edit articles

ArticleEditPage is used to implement the article editing requirements mentioned in section 2.3.3. It consists of the TitleViewPanel (cf. figure 3.10) and the ContentEditPanel.

a. ContentEditPanel

Figure 3.13 shows the screenshot of the ContentEditPanel, which includes a preview area and a ContentEditForm. In order to receive and process the user's input, the ContentEditForm is constituted of a text area and two buttons: preview and save. The text area is used to receive the user's input. At the beginning, it is initialized by the latest content through invoking the getLatestContentByArticleId method. If the preview button is pressed, the renderHtml method is called to translate the input content from Wikitext to HTML format. The result is shown in the preview area. Whereas if the save button is pressed, the createRevision method is invoked to save the input content as a new revision record in the database. Besides the content, the


Sweble Wikitext Parser is a parser for *Wikitext*. It can be used in *MediaWiki*.

It generates following results:

- machine-readable representation
- HTML file

```
'''Sweble Wikitext Parser''' is a parser for
''Wikitext''. It can be used in ''MediaWiki''.

It generates following results:
* machine-readable representation
* HTML file
```

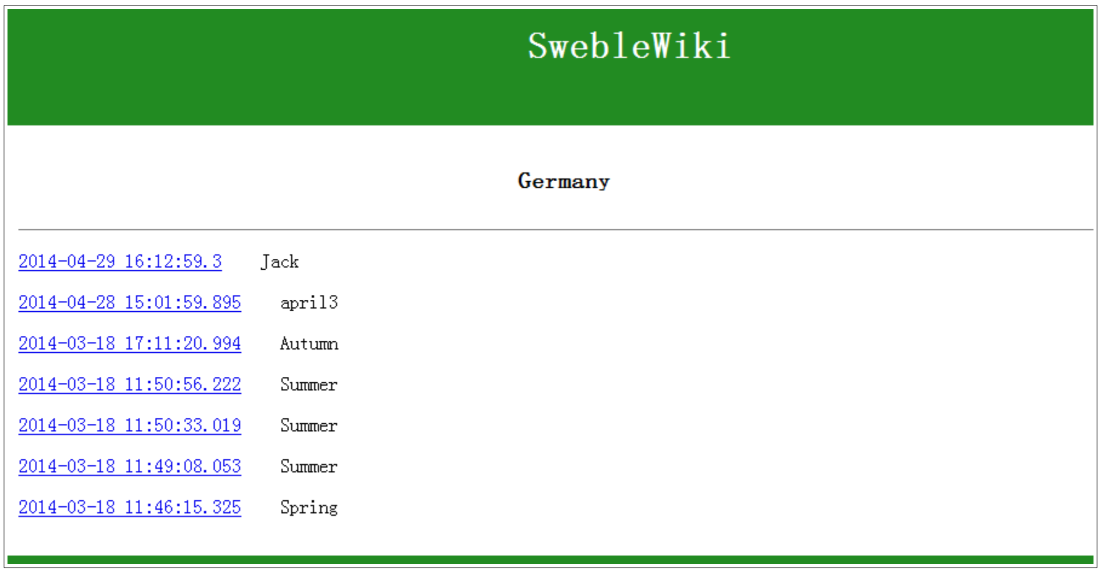
Figure 3.13: Screenshot of ContentEditPanel

information of current article id, user id and current timestamp is also collected to set up a revision record. Once a new revision is created, the latestRevisionId of the corresponding article record is updated. Finally, the user will be redirected to the ArticleViewPage to view their work.

3.2.5 Revision history

The RevisionHistoryPage is used to show the revision history of a specific article. This page is composed of TitleViewPanel, ListView and NoContentPanel. The NoContentPanel is only visible when this article doesn't possess any revision.

The getRevisionTreeOfArticle method is invoked to acquire a tree of all revisions. It takes advantage of the previousRevId of each revision record to connect the revisions in reverse chronological order. Only the revisions that are not softly deleted are added to the tree. Afterwards, a Wicket ListView component is applied to show the result list. Concretely, the revision's creation time and its author are listed. The creation time is rendered as a link connecting to the ArticleViewPage to display the content of the specific revision. Figure 3.14 shows the revision history of the article "Germany". RevisionHistoryPage gives the user an opportunity to view all the previous revisions. While through other portals, the user always accesses the latest revision.


SwebleWiki

Germany

2014-04-29 16:12:59.3	Jack
2014-04-28 15:01:59.895	april3
2014-03-18 17:11:20.994	Autumn
2014-03-18 11:50:56.222	Summer
2014-03-18 11:50:33.019	Summer
2014-03-18 11:49:08.053	Summer
2014-03-18 11:46:15.325	Spring

Figure 3.14: Screenshot of revision history of the article "Germany"

3.2.6 Search articles

To fulfill the requirement of search function mentioned in section 2.3.5. While the SearchPanel (cf. section 3.2.2) receives the user input keywords, SearchResultPage is used to return the search results to the user.

a. SearchResultPage

Once SearchResultPage receives the keywords, the method searchByKeyword is called to find the articles to match the keywords. The search algorithm looks up the article and revision tables. If the keywords appear either in the title or in the most updated content, such an article is regarded as the matching one and returned to the user. If no article satisfies the matching conditions, a feedback information is shown. Whereas if some articles are found, the results are showed through a Wicket ListView. The title of each article is rendered as a link which connects to its corresponding ArticleViewPage.

3.2.7 Create articles

ArticleAddPage is developed to take care of the first step of article creation mentioned in section 2.3.6. It contains two forms: ArticleAddForm and FeedbackForm. At the beginning, only the ArticleAddForm is visible, which is used to get the input title. Once the title is submitted, the checkExistenceOfArticleTitle method is called. The task of this method is to query the database and check whether the input title already exists. According to the check results, four different situations could happen. Then the FeedbackForm is rendered to show the check result and lead the user to the next step.

The first situation indicates the title doesn't exist. If the user confirms the title, a new article record with this title is created and saved in the database. Then the user is redirected to the ArticleEditPage to continue with the content. The second situation indicates that the title exists, but there is no revision belongs to it. The user can choose to add the first revision of this article. The third situation indicates the title exists and at least one revision belongs to it.

The user can choose to view the existing contents. The fourth situation indicates the article with this title is softly deleted for some reasons. It cannot be viewed or edited for the user temporarily.

3.2.8 Access control

a. Registration

RegistrationPage contains a RegistrationForm with four text fields to receive the user information. The normal text fields are used to receive the username and email address. while the password text fields are used to receive the password and confirm password. the password text field is different from the normal one. The input in the password text field is presented as dots instead of the real characters to guard against peep. Moreover, the password text field is reset every time when it is rendered.

To ensure the inputs conform the predefined conditions specified in section 2.3.8, I set up five validators for the input fields. According to its validation rule, each validator can filter out the unsuitable input and report the associated error. Figure 3.15 summarizes the input fields and their corresponding validators. The customized validators, which are the UsernameValidator and the EmailValidator, are used to check whether the input username and email address already exist in the database. EmailAddressValidator is assigned to the email field to check whether the input fits the format of email address. StringValidator is assigned to password field to guarantee the length of password is longer than six characters. EqualPasswordInputValidator is assigned to the confirm password field to check whether the confirm password is the same as the password.

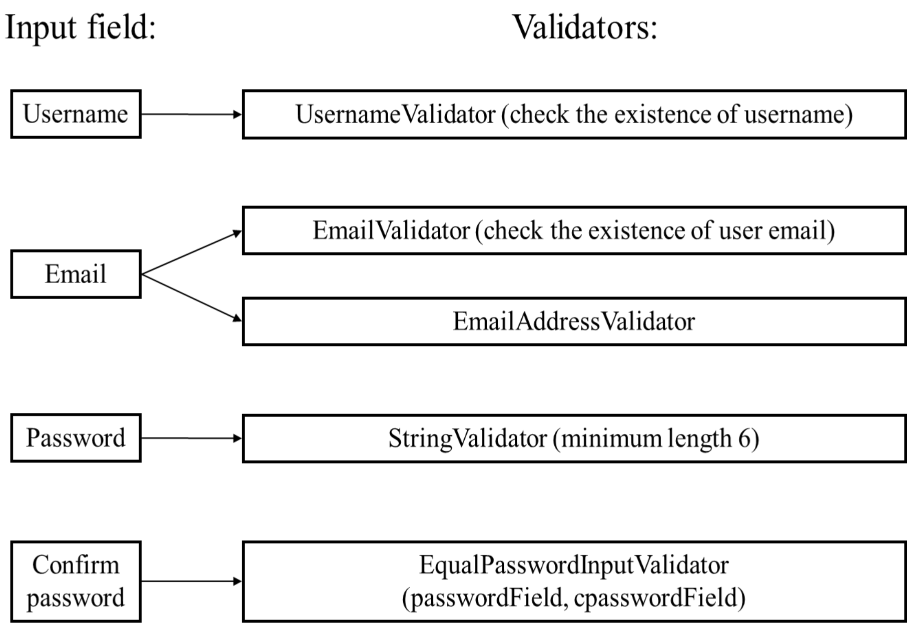

Input field:
Validators:
Username
UsernameValidator (check the existence of username)
Email
EmailValidator (check the existence of user email)
EmailAddressValidator
Password
StringValidator (minimum length 6)
Confirm password
EqualPasswordInputValidator (passwordField, cpasswordField)

Figure 3.15: Summary of the input fields and their corresponding validators

Once the inputs satisfy all the validation rules, they can be submitted. The program first sends an email to the email address offered by the user. If the email is sent successfully, the user data are saved in the database. The sending email function is implemented by using JavaMail (javax.mail) API [20] as the following steps:

1. Instantiate a SenderAuthenticator to provide the authentication information;
2. Configure the parameters of SMTP server;
3. Create a session with the information of SMTP server and authentication information to sign in the sender email account;
4. Instantiate a multipurpose internet mail extensions (MIME) message to set up a concrete email message, including subject, content, date, sender address, sender name, and the email address of recipient;
5. Send email with the MIME message.

b. Authentication

In Wicket, the access control for several pages or components is implemented in two steps: authentication and authorization. In my design, the first step is to authenticate whether the user has registered in the application. The second step is to authorize the user to access the ArticleEditPage and the ArticleAddPage according to his role.

The authentication process is realized through LoginPage. Since LoginPage is the public area of a website that always opens to users. It should be implemented as a Wicket stateless page, which means it is independent of the user session. The statelessness of LoginPage brings about several benefits, such as improving user experience, saving resources and avoiding security weakness [5]. In particular, LoginPage requires the user to input his username and password. Similar to the RegistrationPage, the password field also has a StringValidator to make sure the password is longer than six characters.

The username and password are checked in the authenticate method in LoginSession, which is an instantiation of Wicket AuthenticatedWebSession. In the authenticate method, the getUserByUsername method is called to retrieve a user record against the username. If no record is found or the password is not matched, the LoginPage is sent back with the indication of failure reasons. Otherwise, if a record is found and the password is matched, the user is authenticated successfully. After that, a new user session is created and the user information is stored in it. The user is in the stateful circumstance, which means the user's information and behavior can be stored in the session and the stored information is available in the whole session. If the user wants to access the protected pages, it is not enough to only pass the authentication. He needs to be authorized according to his role.

c. Authorization

In the next step, the authorization process is used to determine whether the user is allowed to access the protected pages. To protect ArticleAddPage and ArticleEditPage from the unauthorized accesses, they need to be distinguished from the unprotected pages. As a result shown in figure 3.16, the ArticleAddPage and ArticleEditPage are inherited from the ProtectedWebPage, while the other pages extend the BasePage directly.

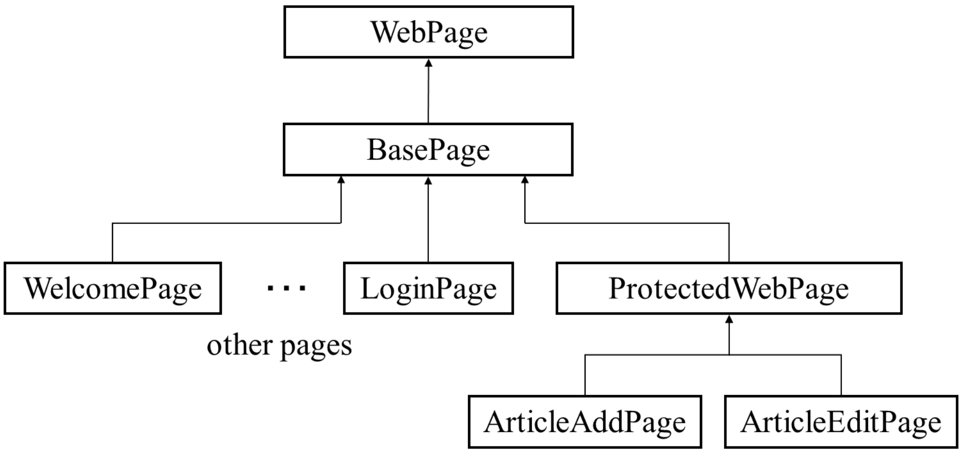


Figure 3.16: Final page hierarchy of Sweble wiki engine

IAuthorizationStrategy interface is provided by Wicket to manage authorization in cross-cutting concern mechanism. In particular, isInstantiationAuthorized method is responsible for authorization on the component creation and isActionAuthorized method takes care of authorization on specific actions on components after creation [6].

Every time the user requests a new page, the application first checks whether this page can be created according to the isInstantiationAuthorized method in AuthorizationStrategy. If the requested page is not the protected pages or the current session is signed in, the requested pages can be instantiated. Otherwise, if the requested page extends ProtectedWebPage and the current session is not signed in yet, the user is redirected to the LoginPage. Further, if the user is signed in successfully, the original requested page is constructed. The concrete instantiation authorization process is shown in figure 3.17.

According to the requirements in section 2.3.8, only the users who own the “editor” role or senior have the privilege to edit or add wiki articles. Thus, after the requested page is instantiated, it is necessary to check whether this page can be enabled to the user according to his role. The role-based action authorization is conducted in isActionAuthorized method. If the logged in user owns the “editor” role or even senior, he can access the ArticleAddPage or ArticleEditPage and undertake the associated add or edit operation. Otherwise, an access denied page is returned to the user.

d. Show authentication status and logout

UserPanel is added to the HeaderPanel to indicate the user’s authentication status (cf. figure 3.18). The UserPanel includes the “Register” and “Login” links, which connect to the RegistrationPage and the LoginPage respectively. The panel also contains a username label and a “Logout” link. The visibility of every component is controlled by overriding its isVisible method. The “Register” and “Login” links are visible when the session is not signed in, whereas the username and “Logout” link are visible after the user is logged in. Logout functionality is realized by clicking the “Logout” link, which invalidates the session and redirects the user to HomePage.

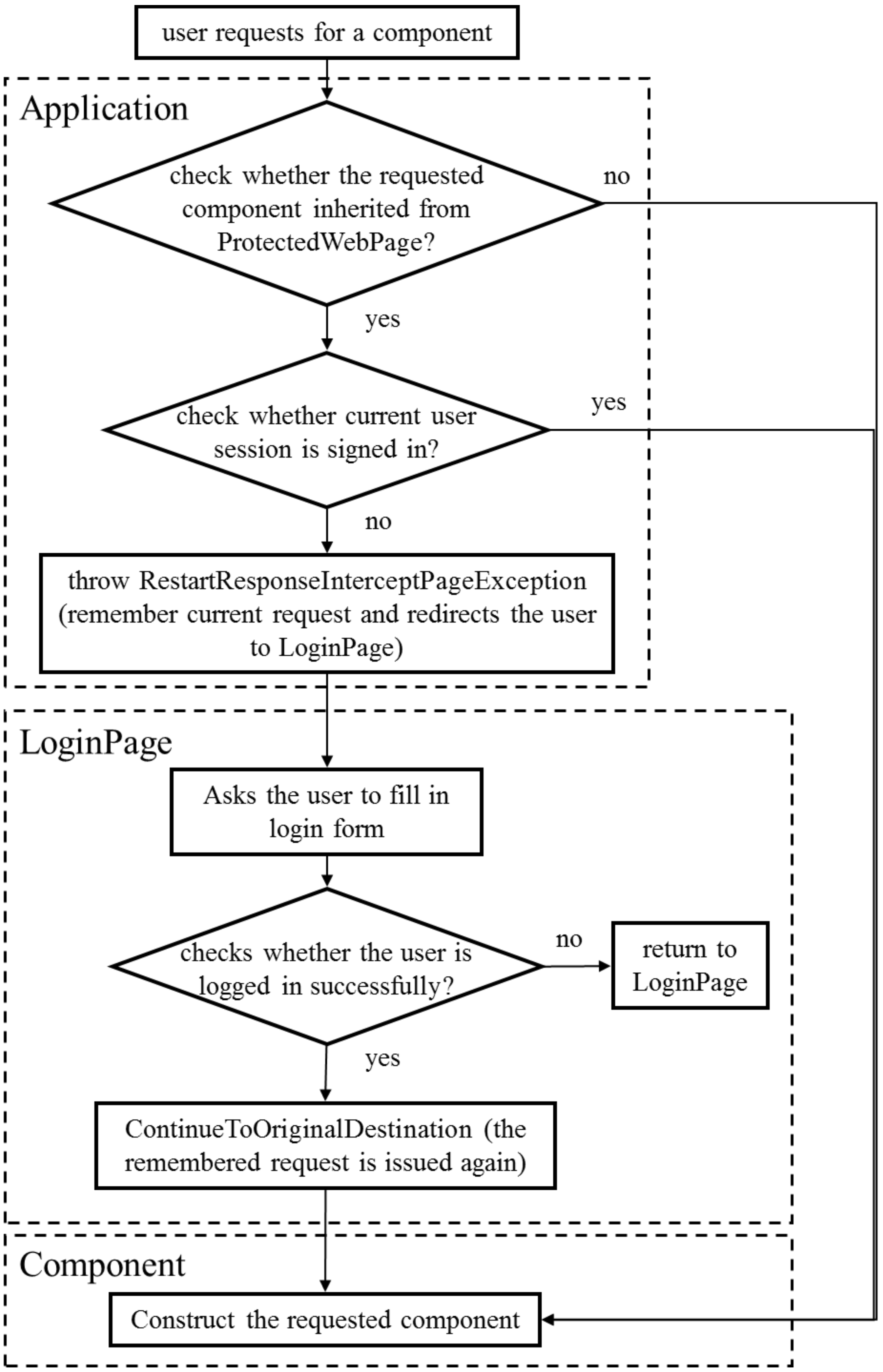
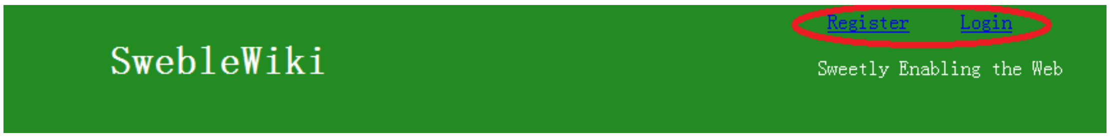
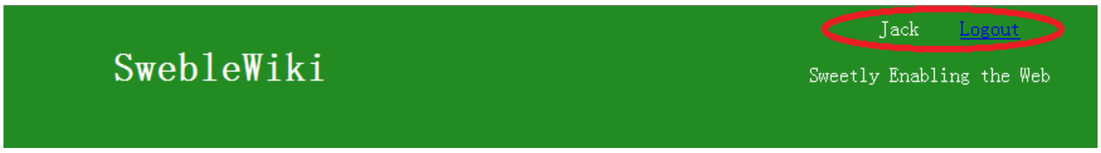

user requests for a component
Application
check whether the requested component inherited from ProtectedWebPage?
no
yes
check whether current user session is signed in?
yes
no
throw RestartResponseInterceptPageException (remember current request and redirects the user to LoginPage)
LoginPage
Asks the user to fill in login form
checks whether the user is logged in successfully?
no
return to LoginPage
yes
ContinueToOriginalDestination (the remembered request is issued again)
Component
Construct the requested component

Figure 3.17: Authorization process of access control



(a)



(b)

Figure 3.18: Screenshot of HeaderPanel with UserPanel: (a) HeaderPanel before the user is logged in; (b) HeaderPanel after the user is logged in

3.3 Concrete implementation of persistence layer

JOOQ is the ORM framework for persistence layer. In figure 2.2, Postgresql database [23] is chosen as the external storage. All the Java classes in this layer are generated by using jOOQ code generation, which process is presented in section 3.3.1. In addition, since jOOQ lacks mechanism for transaction handling, Spring is used as the framework for transaction management, which is introduced in the section 3.3.2.

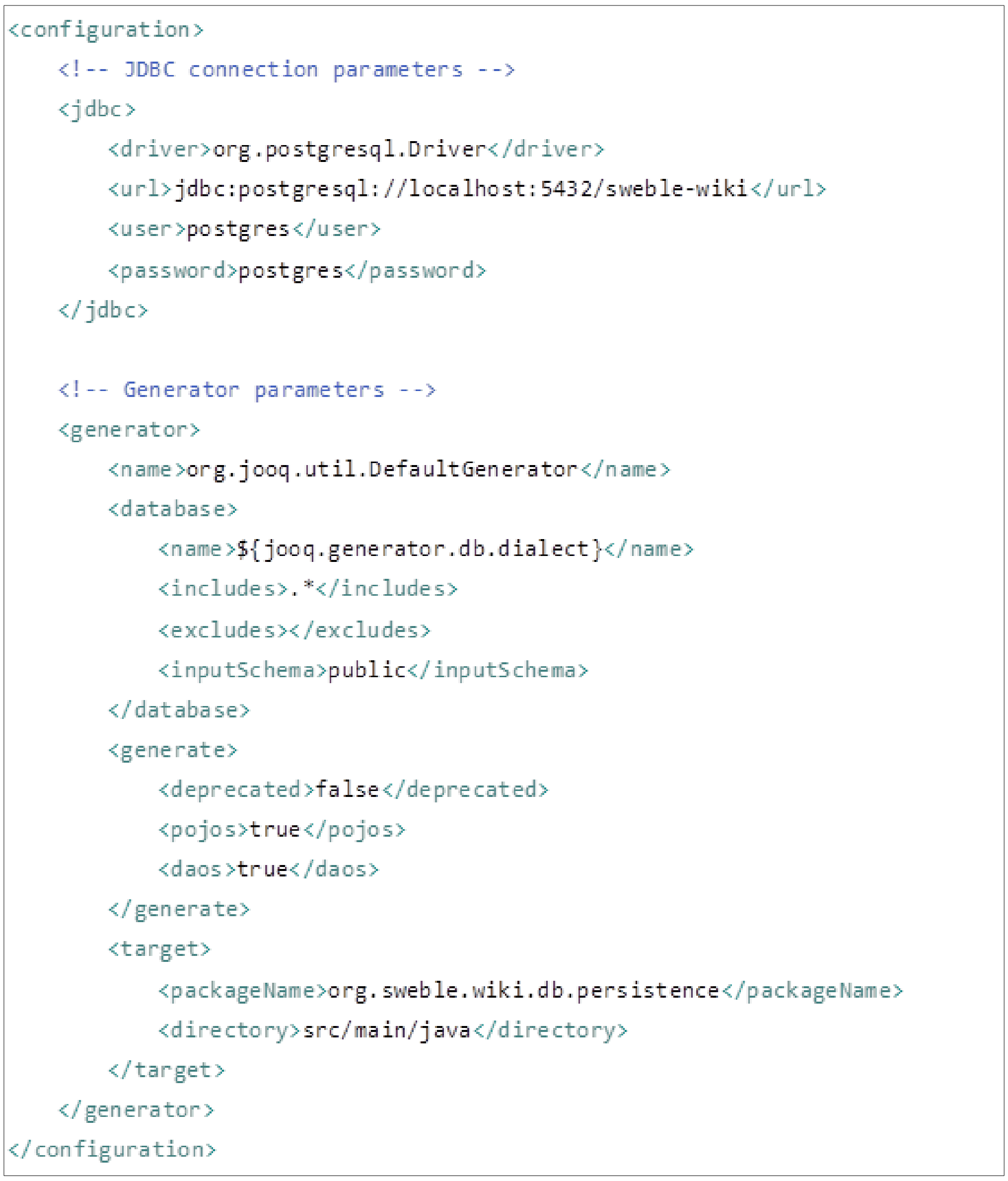
3.3.1 JOOQ code generation with Maven

Firstly, I create a database schema of "sweble-wiki" by SQL script according to the description in section 2.4.2. After it is to be executed by the Postgresql database, the sweble-wiki database with article, revision and user tables are created.

The next step is to generate codes from database. By using the official jooq-codegen-maven plugin, the source code generation is integrated in the Maven build process. It should be noticed that the source code generation is only necessary when something is updated to the database schema. Thus, the plugin should be put into a build profile so that the generation only activates when it's needed.

The configuration of jooq-codegen-maven plugin is shown in figure 3.19. Firstly, the <*jdbc*> fragment are a set of JDBC connection parameters, including the database driver class, url, username, and password. Secondly, the <*generator*> fragment is separated into three parts, which are <*database*>, <*generate*> and <*target*> fragments. The <*database*> fragment tells the code generator which database is used as the source. In my case, the database dialect is Postgresql and the source includes all the tables under the public schema. The <generate> fragment asks the generator to generate classes for database tables and records by default and generate POJOs and DAOs in addition. The <target> fragment specifies the target package and directory for the created classes. Finally, I also add the Postgresql database as the dependency of this plugin although it is not shown in figure 3.19.

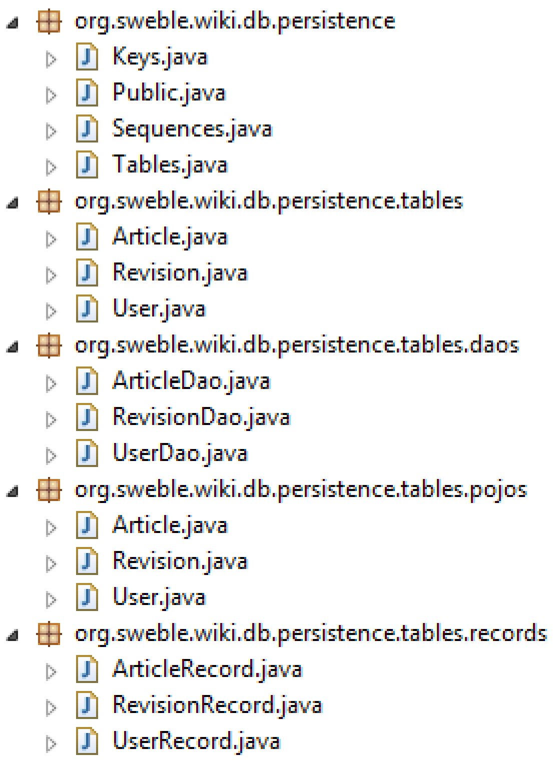
When the profile for this plugin is trigged, jooq analyzes the database schema and generates classes to the target directory and package [21]. Figure 3.20 shows the generated Java classes according to the configuration in figure 3.19. The classes of keys, public, sequences and tables, which are under the org.sweble.wiki.db.persistence package contains the global metadata of the database. Every table in database generates a table classes which are under the tables sub-package. Further, record classes are used to map every record in the table. Like the traditional ORM framework, the generated POJO and DAO classes can be used to facilitate the data access process. All of these Java classes can be used by the DBServiceImpl class in logic layer.



```
<configuration>
    <!-- JDBC connection parameters -->
    <jdbc>
        <driver>org.postgresql.Driver</driver>
        <url>jdbc:postgresql://localhost:5432/sweble-wiki</url>
        <user>postgres</user>
        <password>postgres</password>
    </jdbc>

    <!-- Generator parameters -->
    <generator>
        <name>org.jooq.util.DefaultGenerator</name>
        <database>
            <name>${jooq.generator.db.dialect}</name>
            <includes>.*</includes>
            <excludes></excludes>
            <inputSchema>public</inputSchema>
        </database>
        <generate>
            <deprecated>false</deprecated>
            <pojos>true</pojos>
            <daos>true</daos>
        </generate>
        <target>
            <packageName>org.sweble.wiki.db.persistence</packageName>
            <directory>src/main/java</directory>
        </target>
    </generator>
</configuration>
```

Figure 3.19: Configuration of jooq-codegen-maven plugin



```
org.sweble.wiki.db.persistence
    Keys.java
    Public.java
    Sequences.java
    Tables.java
org.sweble.wiki.db.persistence.tables
    Article.java
    Revision.java
    User.java
org.sweble.wiki.db.persistence.tables.daos
    ArticleDao.java
    RevisionDao.java
    UserDao.java
org.sweble.wiki.db.persistence.tables.pojos
    Article.java
    Revision.java
    User.java
org.sweble.wiki.db.persistence.tables.records
    ArticleRecord.java
    RevisionRecord.java
    UserRecord.java
```

Figure 3.20: The generated Java classes by jOOQ

3.3.2 Transaction management with Spring

JOOQ can only be used for SQL building and SQL execution, which means that jOOQ doesn't take charge of transaction handling, i.e. it doesn't close connection, rollback or commit transactions [9]. However, the transaction management is necessary if multiple data sources are involved. One example of this situation is the connection pooling technology [27], which dynamically opens and maintains database connections for the users of a web application. It is commonly used to improve the database connection efficiency, which is also applicable in Sweble wiki engine. Thus, the transaction management should be taken into account in the situation that connection pooling technology is used.

Spring framework is chosen to manage transactions between Sweble wiki engine and the database. In order to make jOOQ, Google Guice and Spring to work together, some configuration steps are needed [13]. After the configuration, the jOOQ exception and Spring DataAccessException are in the consistent exception hierarchy so that Spring framework can manage the underlying SQL execution. Further, *@Transactional* annotation can be used in the methods in DBServiceImpl to ensure the SQL statements in each method are executed in one transaction, which means that either all of them are committed or all of them are rolled back.

Reference

[1] R. M. Adler. Coordination models for client/server. Computer, volume 28, Issue 4, Apr. 1995, pp.14-22.

[2] Apache Software Foundation. Apache Wicket. Available at: http://wicket.apache.org/, accessed on 20th May 2014.

[3] D. J. Barrent. MediaWiki. O'Reilly, 2008.

[4] D. Barry and T. Stanienda. Solving the Java object storage problem. Computer, Nov. 1998, pp. 33-40.

[5] A. D. Bene, C. Hufe, C. Kroemer, D. Bartl and P. Bor. Apache Wicket user guide – reference documentation. Available at: http://wicket.apache.org/guide/guide/single.pdf, accessed on 20th May 2014.

[6] M. Dashorst and E. Hillenius. Wicket in action. Manning, 2009.

[7] E. W. Dijkstra. A discipline of programming. Prentica Hall, Englewood Cliffs, NJ, 1976.

[8] Data Geekery GmbH. Java object oriented querying (jOOQ). Available at: http://www.jooq.org/, accessed on 10th May 2014.

[9] Data Geekery GmbH. The jOOQ user manual. Available at: http://www.jooq.org/doc/3.3/manual-pdf/jOOQ-manual-3.3.pdf, accessed on 20th May 2014.

[10] H. Dohrn and D. Riehle. Wom: An object model for Wikitext. Technical report CS-2011-05, University of Erlangen, Dept. of Computer Science, July 2011.

[11] H. Dohrn and D. Riehle. Design and implementation of the Sweble Wikitext parser: unlocking the structure within Wikipedia. Proceedings of the 7th International Symposium on Wikis and Open Collaboration, 2011, Mountain View, CA, USA.

[12] H. Dohrn and D. Riehle. Design and implementation of wiki content transformations and refactorings. Proceedings of the 9th International Symposium on Wikis and Open Collaboration, 2013, Hong Kong.

[13] L. Eder. JOOQ-Spring-Guice-Example. Available at: https://github.com/jOOQ/jOOQ/tree/master/jOOQ-examples/jOOQ-spring-guice-example, accessed on 1st June 2014.

[14] T. Elrad, R. E. Filman and A. Bader, Aspect-oriented programming: introduction. Communications of the ACM, Volume 44, No. 10, 2001.

[15] D. F. Ferraiolo, D. R. Kuhn and R. Chandramouli, Role-based access control. Artech House, Inc. Norwood, MA, 2003.

[16] M. Fowler, D. Riche, M. Foemmel, E. Hieatt, R. Mee and R. Stafford. Patterns of enterprise application architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[17] M. Fowler. Inversion of control containers and the dependency injection pattern. Available at: http://www.martinfowler.com/articles/injection.html, Jan. 2004, accessed on 30th May 2014.

[18] M. Fowler. Domain specific languages. Addison-Wesley Professional, 2010.

[19] Google Development Group. Google Guice. Available at: http://code.google.com/p/google-guice/, accessed on 20th May 2014.

[20] JavaMail API documentation. Available at: https://javamail.java.net/docs/api/, accessed on 10th May 2014.

[21] P. Kainulainen. Using jOOQ with Spring: Code generation. Available at: http://www.petrikainulainen.net/programming/jooq/using-jooq-with-spring-code-generation/, 11th, Jan. 2014, accessed on 1st, June 2014.

[22] D. L. Parnas. On the criteria to be used in decomposing systems into modules. Communications of the ACM, Volume 15, No. 12, 1972.

[23] PostgresSQL Global Development Group. PostgreSQL. Available at: http://www.postgresql.org/, accessed on 10th May 2014.

[24] D. R. Prasanna. Dependency injection. Manning Publications Co., 2009.

[25] Red Hat Developers. Hibernate. Available at: http://hibernate.org/, accessed on 20th May 2014.

[26] W. Richardson. Blogs, wikis, podcasts and other powerful Web tools for classrooms. Thousand Oaks, CA: Corwin Press, 2006.

[27] SQL Server Connection Pooling (ADO.NET). Available at: http://msdn.microsoft.com/en-us/library/8xx3tyca(v=vs.110).aspx, accessed on 1st June 2014.

[28] J. Tang. Improving the Wikipedia parser. Diplomarbeit, Friedrich-Alexander University of Erlangen-Nuernberg, 2011.

[29] Transaction management. Available at: http://docs.spring.io/spring/docs/2.0.8/reference/transaction.html, accessed on 1st June 2014.

[30] R. Vanbrabant. Google Guice: agile lightweight dependency injection framework. Springer, 2008.

[31] I. Vaynberg. Apache Wicket cookbook. PACKT Publishing, 2011.

[32] C. Walls and R. Breidenbach. Spring in action. Manning Publications, 2005, pp. 10-150.

[33] J. A. West and M. L. West. Using wiki for online collaboration: the power of the read-write web. San Francisco, CA: Jossey-Bass.

[34] Wikimedia Foundation. Wikipedia. Available at: http://en.wikipedia.org/wiki/Wikipedia, accessed on 20th, May, 2014.