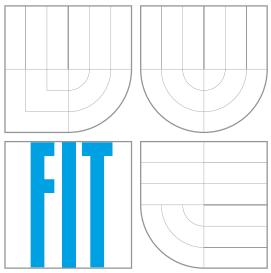
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PLATFORMA PRO VÝVOJ RIA APLIKACÍ

PLATFORM FOR RICH INTERNET APPLICATIONS DEVELOPMENT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN STŘÍŽ

VEDOUCÍ PRÁCE
SUPERVISOR

doc. Ing. ADAM HEROUT, Ph.D.

BRNO 2011

Master Thesis Specification/11111/2010/xstriz02

Brno University of Technology - Faculty of Information Technology

Department of Computer Graphics and Multimedia — Academic year 2010/2011

Master Thesis Specification

For: **Stříž Martin, Bc.**
Branch of study: Information Systems
Title: **Platform for Development of Rich Internet Applications**
Category: Software Engineering

Instructions for project work:

1. Study and describe the possibilities of enterprise development in Java programming language.
2. Assess and compare available technologies for development of RIA (Rich Internet Applications).
3. Design and implement an architecture for development of RIA applications.
4. Implement a sample application on top of the designed platform.
5. Review the achieved results and suggest possibilities of further development of the project; create a poster for presenting the results of the work.

Basic references:

- Martin Fowler: Patterns of Enterprise Application Architecture
- Mike Keith, Merrick Schincariol: Pro Jpa 2: Mastering The Java(tm) Persistence API
- Dhrubojyoti Kayal: Pro Java EE Spring Patterns: Best Practices and Design Strategies; Implementing Java EE Patterns with the Spring Framework

The Term Project discussion items:
Points 1 and 2, partial elaboration of points 3 and 4.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

The Master Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Herout Adam, doc. Ing., Ph.D.**, DCGM FIT BUT
Beginning of work: September 20, 2010
Date of delivery: May 25, 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

Jan Černocký
Associate Professor and Head of Department

Abstrakt

Práce má za cíl navrhnout a implementovat plaftormu pro interaktivní webové aplikace (*Rich Internet Application* – RIA) na základě vhodných technologií pro programovací jazyky Java a JavaScript. Důraz je kladen na výběr odpovídající sady softwarových knihoven, používání principů správného objektově-orientovaného návrhu a programování a možnost dlouhodobé údržby výsledné aplikace.

Abstract

This thesis aims to design and implement the platform for Rich Internet Applications based on technologies suitable for Java and JavaScript programming languages. The emphassis is put on choosing the appropriate software stack from available libraries, using proper principles of object-oriented design and programming and possibility of long term support of the resulting application.

Klíčová slova

Rich Internet Application, RIA, webové aplikace, podnikové aplikace, Java, JavaScript, Spring, JPA, Hibernate

Keywords

interaktivní webové aplikace, RIA, web application, enterprise, Java, JavaScript, Spring, JPA, Hibernate

Citace

Martin Stříž: Platform for Rich Internet Applications development, diplomová práce, Brno, FIT VUT v Brně, 2011

Platform for Rich Internet Applications development

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením doc. Ing. Adama Herouta, Ph.D.

.....
Martin Stříž
25. května 2011

Poděkování

Chtěl bych poděkovat vedoucímu práce doc. Ing. Adamu Heroutovi, Ph.D., Ing. Jaroslavu Bazalovi a kolektivu společnosti RAYNET, s. r. o., za jejich odbornou pomoc při zpracování tohoto tématu.

Contents

Chapter 1

Introduction

Nowadays, *Rich Internet Applications* (RIA) gain a lot of popularity, mostly because of increasing usage of cloud computing. The applications can be accessed through the internet from every place around the globe with only web browser required on the client.

The goal if this thesis is to design and implement the Rich Internet Applications platform based on technologies suitable for Java and JavaScript programming languages. The emphassis is put on choosing the appropriate software stack from available libraries, using proper principles of object-oriented design and programming and possibility of long term support of the resulting application.

In chapter 3 the specification of requirements for the platform, refines the goals and compares classic web application to the one created as RIA and sets the characteristic signs of the resulting enterprise architecture.

Chapter 4 assesses candidate technologies that would contribute to the platform implementation. For each category like enterprise frameworks, persistence or security several possibilities are compared and analysed and then the final choice of technologies is selected.

Architecture and its implementation are handled in chapters 5 and 6. The overall structure and the most important components are presented. The focus is put on the data record handling and inter-layer communication, which are the key ideas introduced into the design and the consecutive development.

With the architecture specified and implemented, the implementation is then verified by creating an example application – the information system for a web hosting company. Concrete implementations of components are presented, further clarifying the concepts of handling metadata and data records and discussing the usability in the real-world deployment and ideas for further improvement and extensions.

Chapter 2

Motivation for the new platform development

Before the specification, design and implementation of the RIA platform is going to be presented, the reasons and events that lead to its development should be discussed.

In the beginning of 2009 I was tasked by the Czech company *RAYNET, s. r. o.* to conduct a research and assess the alternatives for the system that was used as a base for the enterprise web applications development. The system in question was in their portfolio since 2004 and was becoming out-of-date because of the following reasons:

- The web application was based on HTML 4.0 standard and rendered using framesets and frames, which made introduction of the modern standards and interactive elements with asynchronous requests and dynamic updates of the page difficult.
- Data access layer was based on the custom solution based on mapping JDBC to the hash maps. Although sufficient, advanced relations were hard to define and implement.
- The system was built around the *Struts 1.x* framework, which enforces a layer separation into model, view and controller that allow the developer to split presentation and business logic. However, the framework reached its end-of-life in 2008 and stopped being supported.

Where the system really shined, was the central definition of metadata. The centre point of the whole application consisted of an *application tree*, which contained all attributes and views present in the application and served as a source for the security model and the database mapping[1].

The old system was a foundation for the many customer-specific implementations that were deployed in production. One time, the customer required new features that were very complex and would need large changes to the system's architecture. At that time I found the *Spring Framework* (in its version 2.5) and integrated it into the old Struts-based application, first as a plugin, later as a full *Model-View-Controller* (MVC) solution. Spring's features like dependency injection and component based programming were invaluable for the development and greatly increased the maintainability of the product[2], making the requested features possible.

[1] The similar system is reflected into the new architecture. More about the subject is in section 5.2.2.
[2] Spring Framework is discussed in detail in section 4.1.2.

Using the experience with Spring Framework I decided to use it as a foundation of the new product for its flexibility and non-invasive nature. The alternative choice was to use the Enterprise Java Beans (EJB) container from the Java Enterprise Edition API. Both solutions are thoroughly compared later in this thesis.

Spring Framework and EJBs were not the only options that were considered, others include for example *Apache Struts 2.x*, *Apache Tapestry*, *Apache Wicket* and *Apache Click*. All of these frameworks were focused on the web application front-end development (HTML pages, view components) and could not be easily used as a general purpose solution for the complex enterprise applications. Therefore, most of them provide integration with Spring Framework to delegate the back-end business logic.

Non-java alternatives were also considered, notably *Ruby on Rails*, *Grails* and *ASP.NET*, but these options were quickly discarded, because the company wanted to fully utilise the experience of its development team and to have its portfolio technologically consistent to aid the easier maintenance.

Chapter 3

Specification of requirements

The aim of this thesis is to present and implement an architecture that can be easily applied to development of Rich Internet Applications (abbreviated as RIA further in the text). Main requirements for the architecture are as follows:

Client-server with thin client Client-server based architecture means that the whole system is physically split into two parts: the server and the client. The server is responsible for the business logic, security, data access and processing and other computationally intensive operations. One server can facilitate access of many concurrently working users.

The clients can be classified as *thick clients* (sometimes called *fat clients*) and *thin clients* depending on whether they contain business logic or not. This thesis will solely consider thin clients that only take care of presentation layer (user interface).

Communication between server and client is realised through computer network using various client-server protocols.

Extensibility Another equally important requirement is extensibility. The architecture should follow principles of clean object-oriented design, for example by using design patterns [1] [2] (and enterprise design patterns [3]), layered design or loose coupling and high cohesion of components [4]. The reason for caring about extensibility is the fact that product specification, in most cases, change in time. Various features are added, removed or completely re-implemented. The architecture should be robust enough to absorb the impact of major changes in the product.

Maintainability After the application is fully developed and delivered to the customer, the maintenance phase of development lifecycle begins. This phase includes fixing issues, doing minor modifications and other kind of services done by application developers and usually is the longest one in the application lifetime.

Fixing errors in applications can be a very costly operation. It is important to apply methods of reduction of the maintenance costs, for example enforcing good programming principles and code quality [5] and/or supporting automated testing. In relation to extensibility, if the components are loosely coupled, they are also more testable, because the tests can be written to each component individually.

Well-documented Documentation is often underestimated part of the software development process. Usual argument for not having documentation is not enough time for writing

one. While it can be true, the properly written documentation can aid new people to join the development quickly and shorten the delivery time. Therefore, the proposed architecture should have clearly documented interfaces and interaction among its components.

3.1 Classic web application

To fully understand the Rich Internet Applications it is necessary to comprehend the operation of classic web application that are based on *Hypertext Transfer Protocol* (HTTP). HTTP is a protocol designed in 1991 for the purpose of exchanging documents in *Hypertext Markup Language* (HTML). These documents can be linked to each other or to other kind of resources like images, videos.

HTTP soon gained the popularity and became the major protocol for sharing information on the World Wide Web. By its design, the protocol is strictly *stateless* and is based on the request-reply model:

1. Client requests a page with *Uniform Resource Locator* (URL).
2. Server parses URL and sends appropriate resource.
3. Client renders the page.

Web applications usually need to keep some degree of information shared among requests. An example can be authentication information to access the restricted area of the application. The first solution to overcome the stateless nature of the protocol is to send all context-sensitive information with each request. However, it is more bandwidth intensive and can also pose a security vulnerability. More often an alternative is used: the context information (so called *session*) is kept on the server side and user accesses it using the random and unique identifier. During each request the web server pairs this identifier with the appropriate session and presents it to the web application.

To summarise, classic web applications are interconnected dynamically generated web pages with business logic done during page transitions. They support shared state on the server in the HTTP session. On each request the new page is generated on the server, sent to client and rendered inside the web browser. This scheme is quite inefficient: if there is even a minor change, the whole page has to be reloaded. From the user perspective everything disappears and appears again.

3.2 Introduction of AJAX

The need of reloading and re-rendering whole pages lead to emergence of AJAX, which stands for *Asynchronous JavaScript and XML*. AJAX is a technology that allows client side (web page) to be more interactive and responsive [6].

Foundation of AJAX is capability to use JavaScript to make an asynchronous request from the client to the server [7]. In coordination with other JavaScript techniques like DOM[1] manipulation and events it is possible to partially change the displayed page in an event driven manner.

[1] Document Object Model – interface to programmatically access and modify elements in HTML or XML document.

Asynchronous requests require support in web browser. Nowadays, all major browsers supports it. Programmatically, everything is done through the special `XMLHttpRequest` object, that allows to open connection and asynchronously, by registering a callback function, wait for an answer.

Working directly with `XMLHttpRequest` object is not so comfortable and can easily lead to errors. DOM implementations in various browsers do also slightly differ among each other. Many JavaScript frameworks emerged because of that (*jQuery*, *Prototype*, *ExtJS* and more). These frameworks shield the user from browser differences in both AJAX and DOM and provide the means of easy modification of actually displayed page.

3.3 Rich Internet Application (RIA)

Rich Internet Application tries to bring desktop applications to users through the web browser. There are two major approaches: using the browser plug-ins and using JavaScript and AJAX.

Browser plug-ins are software components that integrate into browser infrastructure to provide extended capabilities that browser alone does not have. Examples include Adobe Flash, Adobe Flex, JavaFX or Microsoft Silverlight. All of them require prior installation into the browser. The support of different browser and hardware platforms differ from plug-in to plug-in.

JavaScript's advantage number one is the ubiquity – it can be found in all most commonly used browsers. It is also available on different platforms, even mobile ones.

The architecture proposed further in this thesis will be universal, but in some aspects it will be focused on servicing the requests from JavaScript. However, it could be adapted to support another run-time environment without significant effort.

3.4 Advantages and drawbacks of JavaScript RIA

In the previous section it was mentioned that RIA implemented in JavaScript language are focused to simulate desktop environment to the user through an ordinary web browser. This approach has the advantage that the application is easily accessible from everywhere. There is no need to install special software or run-time environment on the clients' computers, the recent web browser will suffice.

Since the JavaScript code is downloaded from the server on demand, the client side of the application can be updated almost instantly. In typical thick client desktop application more sophisticated methods must be used to ensure that the client is up to date.

Significant drawback of applications run in the browser is the complete isolation of such applications. The original intent of client JavaScript implementation in browsers was to provide better user interface and effects, it was not designed to emulate the complete desktop application. For security reasons the JavaScript interpreter runs in a secure *sandbox* with limited access. The client application cannot interact with other parts of the operating system it runs in. The example consequence of such isolation is the inability to directly access files on the client's computer.

Another minor drawback is the different look and feel of the client. Since the application is ran inside the web browser, it renders all of it's output as a web page. The same look and feel is difficult to accomplish accross various web browsers without using the custom components (due to differences in browsers' rendering engines). Therefore the

native window decorations, controls and widgets cannot be used and the application looks different than the rest of the applications in the client's operating system.

Despite the drawbacks the advantage of JavaScript penetration in all recent browsers is the main reason the architecture is going to be designed towards supporting communication with JavaScript clients.

Chapter 4

Technology assessment

In chapter 3 the main requirements for the enterprise architecture were stated. As a development platform, Java in version 6 is used as a base of all considered technologies throughout this chapter.

Section 4.1 contains the overview of the enterprise applications and frameworks that ease their development in Java run-time environment. Further sections evaluate technologies and libraries in various categories. Results of the evaluations serve as a blueprint for design and implementation phases of this work.

4.1 Enterprise applications in Java

Java Enterprise Edition (JEE) is a development platform built on top of *Java Standard Edition* (Java SE) to provide programming interfaces and services for programming enterprise applications and information systems.

The platform includes several specifications to encapsulate the business logic, access database sources, facilitate e-mail messaging, provide web services or serve and generate web pages.

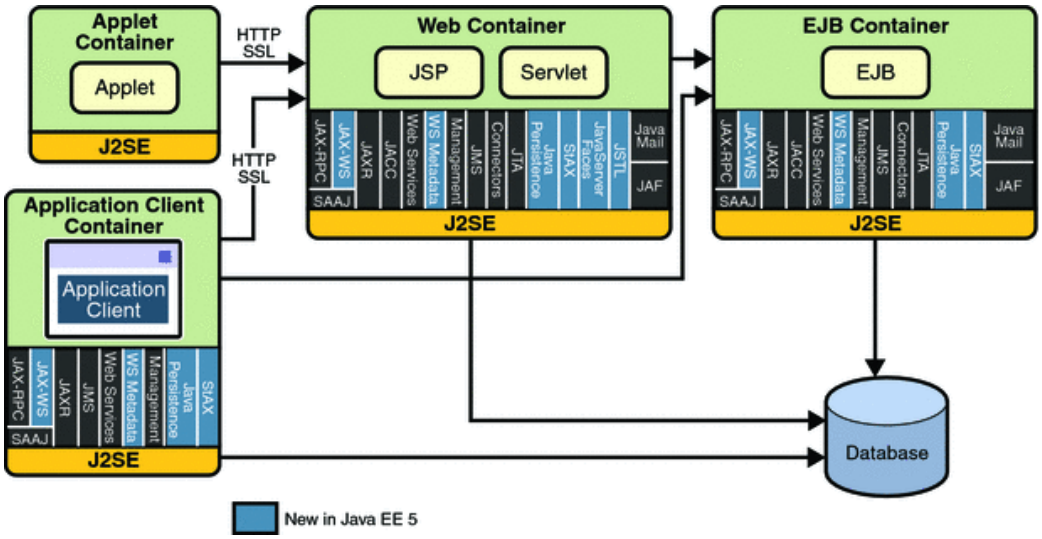


Figure 4.1: Java Enterprise Edition platform API overview (taken from [8])

Java Enterprise Edition applications are developed following these specifications. The

actual run-time environment is provided by an *Application Server* (AS), which contains implementations of all JEE specifications. Nowadays, there are many application servers to choose from, both form open-source and commercial vendors. Application servers are certified to comply with JEE specification after passing compatibility test suite created by Sun Microsystems (now Oracle). Therefore it is theoretically possible to migrate the enterprise application from the server of one vendor to another without problems. In practice, each application server provide a set of additional vendor-specific functionality on top of the original JEE specification. If the application relies on it, the migration is not possible without further effort.

JEE is focused on developing *multitiered* applications. The traditional client-server application, where rich client accesses the database can be classified as two-tiered, whereas JEE application has mostly got three tiers with application server as a mediator between the client and data sources.

Typical web application in this environment consist of four tiers, that can be distributed onto separate machines to improve the scalability of the solution:

1. *Client tier* – web browser.
2. *Web tier* – web components like servlets and Java Server Pages (JSP).
3. *Business tier* – business components like Enterprise JavaBeans (EJB).
4. *Enterprise Information Systems tier* (EIS) – database servers and/or other data sources.

Usual workflow can be shown on the example of saving the order from the ordering system to a database. Client issues the web request by submitting the web form with ordering information. The request is then processed by the web tier, where parameters are parsed into order object and web session is accessed to fill in the information of the currently logged user. Web tier then calls an Enterprise JavaBean in the business tier with the order object. The EJB contains the actual business logic which does price calculations and uses EIS tier to store the order in the database.

4.1.1 Java EE containers and components

Application server provides containers that manage lifecycle of application components. Several types of Java EE components exists, but for the purposes of the web application, two of them are most important: *web components* and *business components* [8].

Web components are either *servlets* or *Java Server Pages* (JSP) [9]. Servlets are application classes that conform to *Java Servlet API* specification and process incoming HTTP requests and return results to the client's web browser. Java Server Pages are text documents that can contain imperative Java code, similar to for example PHP. However, the page code is not interpreted like in the case of PHP, but firstly compiled into servlet and then handling requests like a regular servlet. Servlets are managed by application server in a *servlet container* that takes care of their lifecycle. If the servlet instance does not exists at the time of request, the container automatically creates and initialise it. Also, when the servlet is no longer required, the container disposes it.

Business components are core of the application business logic and contain the code modelling the actual functionality of the application (placing orders, tracking shipments, processing payments, etc.) Components are Enterprise Java Beans (EJB) and are managed

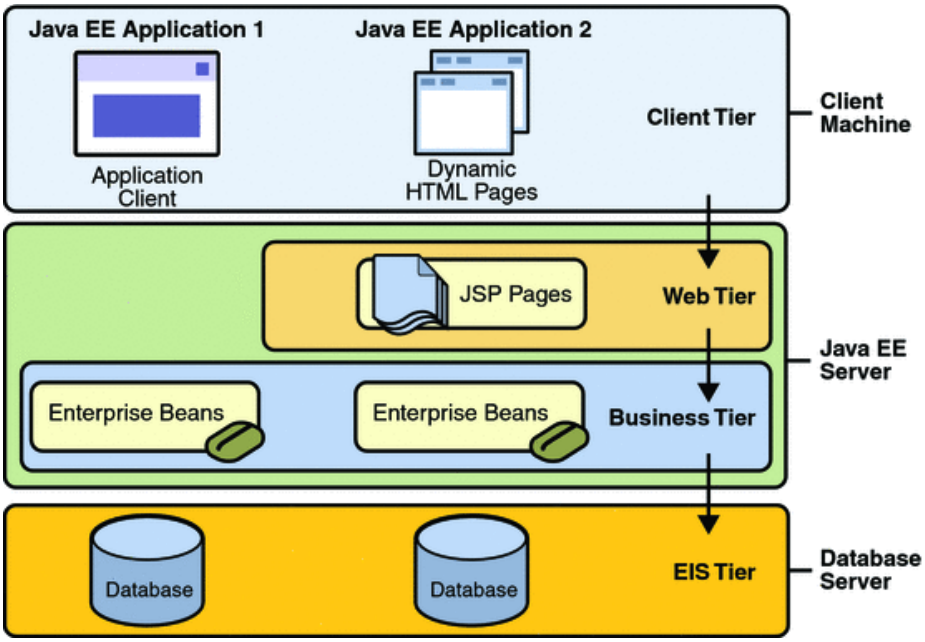


Figure 4.2: Schema of a multitiered application in Java EE environment (taken from [8])

by an EJB container, which manages the lifecycle in the similar way to the web container, but adds several other features like container-managed transactions, security and remoting that ensure that even if the EJBs are scattered to many physical servers, they can be accessed and called like if they were running in the same environment.

Container managed components altogether allow the developers to develop multi-tiered web applications that can be easily deployed to multiple instances of application servers.

4.1.2 Spring Framework

Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications [10] [11]. The main feature of Spring is the ability to compose the application out of intercommunicating POJOs (Plain Old Java Objects[1][12]). Key principles used in the Spring Framework that allow easy composition of applications are *Inversion of Control* (IoC) / *Dependency Injection* (DI) and *Aspect Oriented Programming* (AOP).

Inversion of Control (IoC) / Dependency Injection Conventional applications that are not based on the mentioned principles usually contain central piece of code that controls the flow of the application: it instantiates and disposes components in the right order and with proper dependencies. Disadvantage of this case is that the components are tightly coupled and depend directly on each other. When the developer wants to change the implementation of single component, it may require change to the central piece of integration code and possibly also changes in the dependent components, that increase complexity of such modifications.

[1]Object from Java class, that does not implement any framework-specific interfaces or does not inherit from framework superclass. Mainly it means the object is not the Enterprise JavaBean.

When inversion of control is used, the instantiation and dependency management is handled by the framework. The *control is effectively inverted* from the application code to the framework [13].

Spring provides IoC container to handle the dependency injection. Each component consists from an interface and one or more implementations. With components registered in the container, the framework instantiates all of them together with injecting appropriate dependencies (either via setter method or via constructor parameter). Important difference from the conventional case is that dependencies are based solely on interfaces. The components code does not know anything about the actual implementations hidden behind the interfaces. Dependency injection thus enforces loose coupling and clear separation of concerns.

In Spring Framework, the components are registered into the container in a declarative manner, either from an XML file or by annotating classes with Java annotations. The absence of central implementation that wires everything together in procedural manner allows for reusable and clear implementations.

Aspect Oriented Programming *Aspect Oriented Programming* (AOP) is a paradigm that allows the developer to implement *aspects*, the cross-cutting concerns such as logging, auditing, security or transaction management into the system without modification to the business logic.

Aspects are invoked on declaratively specified events like execution of a method matching certain name. This way it is possible to easily execute code that would otherwise have to be part of all included components. The ideal example of an aspect is the transaction management. Before the execution of a component's method, transaction is created. If the method executes without errors, the transaction is committed, but if an exception is thrown, transaction is rolled back. Everything is handled in a transparent way. Without aspect oriented programming, the transaction handling code would have to be wired into each component requiring transaction management and increase their complexity.

Spring provides the AOP framework that transparently creates dynamic proxy classes. These proxies invoke aspects around components with business logic. Aspects are, same as components, configured by an XML document or annotations. Many predefined aspects exist inside the Spring Framework for various tasks. New aspects can also be introduced should the need arise.

Spring modules Spring Framework is divided into modules that can be categorised into several groups as shown in figure 4.3. Fundamental part of the framework is the *core container*, which is the inversion of control container that provide dependency injection. All application components are registered and instantiated inside it. Due to the flexibility of the container, the application can comprise of loosely coupled components. There is no need to use *singleton design pattern* [1] [2] that would tie the components up to the concrete implementation.

Data access/integration layer contains modules to access and work with various kinds of data sources. JDBC (*Java Database Connectivity*) module contains helper classes to ease the access to the standard JDBC datasources. Its benefits are going to be described in section 4.2.2. In addition from pure JDBC, Spring got modules for the most widely used ORM (*Object-relational mapping*) tools: iBatis/MyBatis, Hibernate, JDO (*Java Data Objects*) and JPA (*Java Persistence API*). Data access layer contains also support for OXM (*Object to XML mapping*) and JMS (*Java Message Service*).

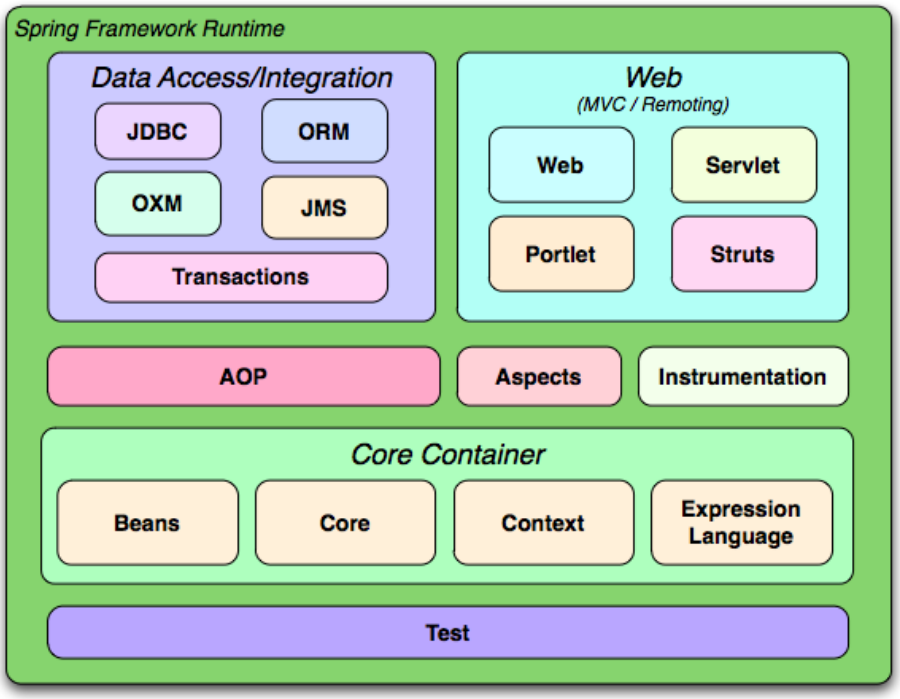


Figure 4.3: Spring Framework runtime modules (taken from [10])

Transactional processing is transparently supported by all modules. Spring provides a powerful abstraction: the *transaction manager*. Many implementations exist to support various data access technologies. From the developers perspective, the usage of the transaction manager is consistent no matter what kind of technology is used. A large benefit is that if the technology should be changed, for example from pure JDBC to Hibernate, only the *implementation* has to be reimplemented. The interfaces stay the same.

Modules of the Spring framework's *web layer* support the web-based part of the application. Web module is based on *Model-View-Controller* architectural pattern [2] [3] to clearly separate web application's business logic (controller), data (model) and presentation logic (view). All web components are, like the business components, loosely coupled and maintained by the Spring core container.

Aspect oriented programming module is another part of the framework. It can be used to put aspects on top of each component included in the container.

4.1.3 Comparison of EJB and Spring containers

EJB and Spring containers have many things in common, both are mainly designed to manage application components. But in order to compare them, the run-time environment has to be included into consideration.

Enterprise JavaBeans container requires a full JavaEE application server to run. The application server is compliant with the JavaEE specification, the developer has the set of defined technologies available. The advantage is that everything is provided by the monolithic application server, the disadvantage is that the specifications itself is the limiting factor for implementation of new technologies.

Spring Framework is much more lightweight solution. Its main feature is noninvasiveness

– all components are simple POJOs wired up together using inversion of control container. The framework can integrate with wide range of technologies out of the box and it is easy to extend it. The following list compares both containers in major aspects:

- *Data persistence* – Data persistence in EJB container is realised by so-called *entity beans* and managed by *Java Persistence API* (JPA). With Spring the developer is not bound only to JPA, but can also choose Hibernate, JDO, MyBatis or even plain JDBC.
- *Transaction management* – Declarative transactions are supported by both containers. EJB relies on *Java Transaction API* (JTA) transaction manager. Spring has an abstraction called `PlatformTransactionManager` with pluggable implementations for JTA, JPA, Hibernate or JDBC.
- *Security* – EJB supports declarative security based on users and roles, its implementation, together with user and role management is specific to the used application server. Spring itself does not contain any security implementation, but a separate project named Spring Security (former Acegi Security) can be used. More information about the Spring Security is written in the section 4.3.2.
- *Dependency Injection and Aspect Oriented Programming* – Both EJB and Spring containers have the ability to inject dependencies in run-time to build-up the complex applications. However, considering AOP, Spring supports the whole set of aspects, whereas EJB supports only method interception (advice before method call).

The lightweight nature of Spring Framework excels during testing. Because components are simple POJOs, they can be tested manually without container. For integration testing Spring provides test context framework to initialise the container and run the tests. Generally, Spring Framework supports *test driven development*.

For the architecture developed as a goal this thesis I am going to use Spring Framework as a development platform because of its flexibility and possibility to use the cutting edge versions of various integrated libraries. This way I can reflect the technological progress during the development.

It is worth to mention, choosing the Spring Framework as a development platform does not close the way to Enterprise JavaBeans. When it is required, for example by means of integration into another enterprise system. Spring has support for EJB – it can be deployed into JavaEE container and provide EJBs and/or use them.

4.2 Data persistence

Information systems are, by their name, based on handling information flows. Information are data with added semantics [14], it is the way how we look at and perceive the raw data. Data itself are only values without further representation. In order to clearly show the difference, here is an example:

- *Data* – 10000 (raw value stored in memory/on disk without further meaning).
- *Information* – account balance in Euro (the way how the value is look at).

This section handles the problems of *data* persistence. Data can be stored, retrieved, modified, deleted, filtered and sorted to achieve the required result. These operations can be done by various technological means, which many of them are going to be described and compared in next sections.

4.2.1 Plain Java Database Connectivity (JDBC)

Java Database Connectivity (JDBC) is a standard Java API for accessing data inside the relational databases. It was designed with simplicity in mind and focuses on three core tasks [15]:

1. Connecting to data source (usually relational database).
2. Sending SQL queries for data retrieval or update.
3. Retrieving and processing data sent as a query response.

JDBC is an abstraction above the raw connection to the database. Most database vendors supply their own *JDBC driver* that allows user to connect to and work with the data source in a uniform way, which is the biggest advantage of this approach.

JDBC code example To demonstrate the features and abilities of JDBC, the following piece of code will be used:

```
List<Employee> results = new ArrayList<Employee>();
Connection conn = dataSource.getConnection();
try {
  PreparedStatement ps = conn.prepareStatement("SELECT name, salary " +
      "FROM employee e " +
      "JOIN department d ON e.department_id = d.id WHERE department.name = ?");
  ps.setString(1, "Sales");
  try {
    ResultSet rs = null;
    try {
      rs = ps.executeQuery();
      while (rs.next()) {
        Employee emp = new Employee();
        emp.setName(rs.getString("name"));
        emp.setSalary(rs.getDouble("salary"));
        results.add(emp);
      }
    } finally {
      if (rs != null) { rs.close(); }
    }
  } finally {
    if (ps != null) { ps.close(); }
  }
} finally {
  if (conn != null) { conn.close(); }
}
return results;
```

The biggest advantage of JDBC is that the Java code is the same no matter what kind of database the developer use. The only difference can be in vendor specific extensions and differences in SQL language implementation. The usual workflow is that `Connection` and related `PreparedStatement` are created. `PreparedStatement` is filled with appropriate parameters and then executed. `ResultSet` is constructed as the result of the query and processed record by record.

By looking at the example code, several disadvantages become apparent. First, the amount of code repetition for each query is very high. Second, as Java language (in version 6) does not automatically close system resources like files or database connections when the variable gets out of scope, the developer has to take care of it properly. It should be done using *try/catch/finally* blocks, otherwise the prepared statement and the connection will not be closed in the case of exception.

Another aspect to consider while developing a complex enterprise system is the maintainability. The domain model change from time to time and it means, the respective queries have to be modified as well. Queries are represented as strings, sometimes dynamically concatenated based on business logic. If the column rename or deletion is desired, it could be difficult to find all queries that reference it and cause errors.

Despite the disadvantages JDBC is widely used in various legacy systems. Also, Java Database Connectivity is also used in many Object-relational mapping libraries as a backend for data querying and retrieval.

4.2.2 Spring JdbcTemplate

Plain JDBC API is the abstraction above database systems, but still it is low-level interface and lot of details have to be taken care about. The common workflow is, as stated by Spring reference documentation [10], summarised in the following list:

1. Define connection parameters.
2. Open the connection.
3. Specify the statement.
4. Prepare and execute the statement.
5. Set up the loop to iterate through the results (if any).
6. Do the work for each iteration.
7. Process any exception.
8. Handle transactions.
9. Close the connection.

There is a large amount of code that is needed to be written in order to successfully retrieve data from the data source, altogether with proper resource management and exception handling. Spring `JdbcTemplate` is a helper class and it is a base of the Spring JDBC support module. It simplifies the use of JDBC API, creates and closes allocated resources (like database connections), handles transactions and errors in a transparent way. Looking at the list above, `JdbcTemplate` leaves only the points 3 and 6 to be implemented by the developer. Here is the code example that handles the same situation like the plain JDBC code presented in section 4.2.1:

```
1  JdbcTemplate template = new JdbcTemplate(dataSource);
2  String sql = "SELECT name, salary FROM employee e " +
3  "JOIN department d ON e.department_id = d.id WHERE department.name = ?";
4
5  RowMapper<Employee> rowMapper = new RowMapper<Employee>() {
6    @Override
7    public Employee mapRow(ResultSet rs, int rowNum) throws SQLException {
8      Employee emp = new Employee();
9      emp.setName(rs.getString("name"));
10     emp.setSalary(rs.getDouble("salary"));
11     return emp;
12   }
13 };
14
15 return template.query(sql, rowMapper, "Sales"); // query, mapper, arguments
```

Compared to the plain JDBC example, there is a lot less code. The common workflow is coded into `JdbcTemplate` and the actual logic is done through callback interfaces. The design pattern used in this case is called *template method pattern* [1] [2], hence the class name `JdbcTemplate`.

Generally, if the application uses Spring Framework (or the Spring Framework is introduced into legacy one), it is far more convenient to use Spring JDBC facilities instead of plain JDBC code, because it leads to more readable code and less probability of error.

The maintainability problem is similar to plain JDBC, the queries are represented by strings and are subject to review, should the domain model changes.

4.2.3 Introduction to object-relational mapping

Data in relational database systems are structured into tables with scalar data composed of data rows and columns. Each table is independent, relations among them are realised using unique identifiers. On the other hand, object-oriented type systems are different, relations among objects are done by object references. Objects usually do not only contain flat and scalar data. They form a graph of interconnected instances.

Both type systems are fundamentally different, but there is a need to convert classic relational schema into objects and back. In sections 4.2.1 and 4.2.2 the manual way of mapping tabular data into objects was described. Doing things manually is feasible for smaller number of entities or for specialised cases. However, if the application contains a lot of simple *CRUD* (Create Read Update Delete) operations, like the rich Internet applications do, it seems more appropriate to automate the process.

During the time several object-relational mapping technologies emerged. In this work I will consider two Java Enterprise standards – *Java Data Objects* (JDO) and *Java Persistence API* (JPA) and their implementations.

4.2.4 Java Data Objects (JDO)

First object-relational mapping technology that is considered in this work is *Java Data Objects* (JDO). JDO is a interface based Java standard. Original version 1.0 was described by the *Java Specification Request* JSR 12 [16], improved version 2.0 by the JSR 243 [17]. Currently, the JDO specification is developed by the community in the *Apache JDO* project.

Java Data Objects are designed to work not only on top of relational database management systems, but also on other types of structured data like XML, Excel, JSON and others.

The application interface is clean and Spring framework contains support for transparent transactions in JDO environment.

Java Data Objects can persist all major relations between objects, such as one-to-one, one-to-many and many-to-many. The relations are well configurable, advanced mapping, operations cascading and collection ordering are supported out of the box.

The tested implementation of JDO was DataNucleus Access Platform in its beta version 2.0 (as in the time of this assessment). All basic CRUD operations were without problems. Batch operations were limited: batch delete is supported, but there is no batch update available.

The problematic part of DataNucleus is the fact, it requires Java bytecode enhancement: after class is compiled from the source to bytecode, new methods are injected to facilitate features like state management, value change interception and on-demand loading of related collections. Three possible solutions were tried:

1. *Ant-based enhancement* – bytecode is enhanced right after compilation by a special task in the Ant build system.

2. *IDE plug-in* – enhancement is done by a plug-in incorporated into Integrated Development Environment. This approach generally works, but it is not feasible for heterogeneous development teams and automated building and testing scenarios.

3. *Load-time weaving* – bytecode modification is done at run-time using a special *instrumentation agent*. Instrumentation at run-time is slow and unreliable, at least in conjunction with lightweight web containers.

Bytecode enhancement and load-time weaving / runtime instrumentation are incompatible with class hot swapping supported by the Java debugger. Hot swap in debugging mode can (if certain conditions are met) replace the running class with the new version, resulting in code change is immediately available without restarting the whole application and leading to much faster development. These and related problems were the main reasons to consider the transition to JPA described in the section 4.2.5.

4.2.5 Java Persistence API (JPA)

Java Persistence API is a standard for object persistence developed as a part of JSR 220 [18] specification (EJB 3.0). JPA was intended to replace the container managed persistence solution present in EJB before version 3.0. JPA was designed to run not only in Enterprise JavaBean containers, but also in standalone Java Standard Edition environment. The standard was soon adopted by most application server vendors.

From 2007 till end of 2009 the JPA version 2.0 was developed in a separate JSR 317 [19]. The aim was to add extended object mapping capabilities, statically typed criteria API and other features present in other persistence frameworks like Hibernate. The JPA 2.0 standard was declared as final in December 2009. Not so long after two major object-relationship mapping frameworks, EclipseLink and Hibernate, became the implementations of the API.

EclipseLink EclipseLink is an open-source fork project that evolved from the Oracle TopLink. After adoption of JPA 2.0 standard, it became its reference implementation. EclipseLink is bundled by default with GlassFish application server and it is widely used nowadays.

While testing on a prototype implementation, EclipseLink behaved according to the specification. The issue was the same like in the case of JDO – bytecode enhancement. Although EclipseLink can be used completely without enhancement or load-time weaving using instrumentation agent, certain advanced features like field change tracking, lazy-loading and fetch groups are unavailable without it.

Hibernate Hibernate is another open-source persistence library originally developed as an alternative solution to the persistence provided by Enterprise JavaBeans version 2 (EJB2). With time it gained new features and became one of the most widely used object-relational mapping persistence solutions. When the JPA standard emerged, Hibernate quickly began to support it. Many features, originally from Hibernate, influenced the creation JPA 2.0 standard, which Hibernate also implements.

DataNucleus and EclipseLink relied on bytecode modification, which proved as a disadvantage together with rapid deployment outside full enterprise application server (see section 4.2.4. Hibernate takes an alternate path: it does not instrument the entity classes directly, but generates synthetic subclasses using *cglib* or *javassist* libraries. These proxy subclasses have got slight memory overhead, but as they do not require processing the class file, Hibernate works without difficulties in various IDEs and build systems.

As an important advantage, Hibernate offers integration with two useful libraries: Hibernate Validator and Hibernate Search. The first transparently ensures the validity of all saved entities before they are saved into database. Validation rules are specified with Java annotations directly into entities' source code. The latter library, Hibernate Search, integrates Hibernate with Apache Lucene fulltext search index capabilities.

The documentation, user guides and manuals are excellent, compared with DataNucleus or EclipseLink the documentation is much more comprehensive. The user community around Hibernate project also seems more active and responsive.

To summarise: no need for bytecode enhancement, seamless integration with validation and fulltext search libraries and very good documentation and support made Hibernate the best choice for the Rich Internet Applications platform developed as a goal of this thesis. And since Hibernate complies with JPA 2.0 standard, there is a possibility to migrate to another JPA provider (EclipseLink, TopLink, OpenJPA or others) without large codebase changes, should the need arise.

4.2.6 Querying in JPA environment

One of the features of JPA is the independency on specific relational database management system. Drivers for many of them exist, including Oracle, PostgreSQL, MySql, H2 and others.

JPA queries are based on *Java Persistence Query Language* (JPQL) language, which is very similar to SQL, just instead of database tables it handles entities. The following piece of code contains the same logic like the SQL example in sections 4.2.1 and 4.2.2:

```
1 TypedQuery<Employee> typedQuery = entityManager.createQuery(
2   "FROM Employee e JOIN e.department d " +
3   "WHERE d.name = :depName", Employee.class);
4
5 typedQuery.setParameter("depName", "Sales");
6 return typedQuery.getResultList();
```

The JPQL query is translated to underlying SQL, executed and processed by the persistence engine. The maintainability issues with queries constructed by string concatenation and references to fields using literal values are present. However, they can be avoided with statically typed criteria API that is described in the next section.

JPA 2.0 Criteria API Data access layer in Rich Internet Applications have often to do complex dynamic queries, for example while filtering and sorting a list. Usually the dynamic queries are built using string concatenation, but this approach has its limits, as it was shown in section 4.2.1.

JPA 2.0 standard comes with a Criteria API that allows the creation of type-safe queries [20]. The only requirement is to create metamodel classes that reflect the representative entity classes. The metamodel does not have to be written by hand, but can be generated automatically by specialised tools that are included in all major JPA implementations.

Type safety is enforced as much as possible: fields that are used in conditions and expressions and query return values are all type-safe. The usage of the criteria API can be seen on the following example:

```
1  CriteriaBuilder cb = entityManager.getCriteriaBuilder();
2
3  CriteriaQuery<Employee> query = cb.createQuery(Employee.class);
4  Root<Employee> employee = query.from(Employee.class);
5  Join<Employee, Department> department = employee.join(Employee_.department);
6  ParameterExpression<String> depNameParam = cb.parameter(String.class);
7  query.where(cb.equal(department.get(Department_.name), depNameParam));
8
9  TypedQuery<Employee> typedQuery = fEntityManager.createQuery(query);
10 typedQuery.setParameter(depNameParam, "Sales");
11 return typedQuery.getResultList();
```

The example is type-safe, all fields are references by metamodel classes' fields (by JPA convention, class names ends with an underscore). The query is constructed incrementally. It means parts of the query can be inserted conditionally without any string concatenations or references.

Type safety with criteria API comes with a price – verbosity. As it is visible from the example, the code is much more verbose. Sometimes the query that would be readable in JPQL could result in hardly readable Java code constructing it. Nevertheless, query type safety pays off in increased maintainability of the data access layer. Each incompatible change of the domain model is clearly visible, because the affected queries cannot be compiled.

Querydsl JPA criteria API has its advantages (type-safety) but also drawbacks (increased verbosity). In ideal state it would be most feasible to have an API that is type-safe, similar to SQL in structure, not excessively verbose and capable of incremental query generation. The product that satisfies all these requirements is *Querydsl*. It provides a *domain specific language* (DSL, hence its name) for a fluent query generation and is based on its own generated metamodel.

```
1  QEmployee employee = QEmployee.employee;
2  QDepartment department = QDepartment.department;
3
4  return new JPAQuery(entityManager)
```

```
5   .from(employee)
6   .join(employee.department, department)
7   .where(department.name.eq("Sales"))
8   .list(employee);
```

The code constructing the query is readable and resembling respective JPQL query string, yet type-safe like while using criteria API. That is why I will use it together with Hibernate as a persistence solution of the proposed architecture.

4.3 Security

Security is a very important aspect of the most multi-user applications. The security concerns all application tiers and layers and has to be done properly. Even a small glitch in design or implementation can cause damage like unauthenticated access to private data or a leak of information to a unauthorised user. Therefore security-related decisions should be taken with great amount of responsibility.

The proposed architecture is going to integrate a well-tested security framework. The list of requirements upon the security framework is as follows:

- *Authentication* – Authentication services should be support at least login form and HTTP basic authentication. They should also be easily extensible to allow others or custom ways of authenticating the user (OpenID, LDAP, Single sign-on, ...).
- *Role-based access control* (RBAC) – Each user belongs to one or more security roles. Permissions to various sections of the application are assigned to roles, not directly to users.
- *Web security* – Rules allowing or denying access to unauthenticated users or selected roles for chosen URLs should be possible.
- *Method-based security* – Selected interfaces should be secured using declarative security aspects in a similar way to declarative transactions. Aspect oriented programming framework would secure the method invocation.
- *Easy integration with Spring Framework.*

Two frameworks were evaluated for the task: *Apache Shiro* and *Spring Security.*

4.3.1 Apache Shiro Application Security Framework

Apache Shiro (former JSecurity and Apache Ki) is a relatively young security framework founded in 2008 and designed with simplicity and extensibility in mind. It is very easy to set up a basic installation. Web based security and method based security are supported. The Apache Shiro comes with its own access control system based on permissions, which could fit simpler security models, but is unusable for complex and dynamic ones.

Spring integration is present as an aspect providing the method level security. Other parts of the security framework are standalone and not registered to the application context.

The evaluation of the Shiro security framework was done in the middle of 2009, when no stable release existed and documentation and further resources were very scarce. Contemporary situation (end of 2010) is better, the project reached its stable release 1.1.0. Lots of issues were fixed and the documentation was also improved.

4.3.2 Spring Security

Spring Security (former Acegi Security) is a mature framework founded in 2003. Nowadays it is maintained directly by the SpringSource corporation. During the time it gained a place as a standard of securing the spring-based applications.

Naturally, integration with Spring Framework is state-of-the-art quality. Every component related to Spring Security is registered into application context. It is possible to add or swap components for custom implementations. The solution is highly customisable.

Both web based and method based security are supported by the framework. Many authentication mechanisms are supported out of the box [21].

There is a lot of documentation and source of information on Spring Security, including a dedicated book [22]. Considering the maturity status, referenced products using the framework and amount of resources, I am going to use Spring Security as a preferred security solution in the proposed application platform.

4.4 Testing

An important part of a software project's lifecycle is testing. Using tests increases the reliability of the product and can significantly decrease servicing costs in the maintenance phase of the project.

Tests can be divided into several categories depending on their scope, where main two of them are:

- *Unit test* – isolated test of a single component. Dependant components are either mocked, stubbed or injected with dummy implementation. The goal is to limit the scope of the test to the minimum, so the component under test is the only thing that can affect the result.
- *Integration test* – test of many components wired-up to form a larger system. It is intended to test complex situations in the application logic.

Testing is valuable not only in the maintenance phase, but also during development. If the tests are in place together with the respective components (or even before), it will ensure that the components are behaving correctly without need to repeatedly launch the whole application to test a single component. Another situation, where unit tests are good to have, is refactoring. The application components evolve over time and tests can enforce that they still function properly after changes.

4.4.1 JUnit

JUnit is de-facto standard library for testing in Java. It originally is meant for unit testing, but together with other tools, such as *Spring TextContext Framework* described in the next section 4.4.2, it can be also used for integration testing. JUnit tests are simple classes with methods containing code that tests components or application logic. Checks are based on *assertions* of the resulting state. The example test class can look like:

```
1 public class ExampleTest {
2   @Test
3   public void shouldReturnResult() throws Exception {
4     TestedComponent tc = new TestedComponent();
```

```
5     Integer result = tc.compute(1, 2);
6     assertNotNull(result);
7     assertEquals(3, result);
8   }
9 }
```

The compiled test can be run from build script, IDE or command line by the JUnit runner that invokes all annotated test methods and evaluates the assertions made inside them. The result of JUnit run is a report, where all successful and failed assertions are presented.

4.4.2 Spring TestContext Framework

Spring Framework contains a module for unit and integration testing, named *TestContext Framework*. The framework contains JUnit custom runner implementation that makes principles of IoC and dependency injection of components available to the unit tests.

The situation, where Spring TestContext Framework really excels, is integration testing. The framework handles the following tasks:

- *Instantiation of Spring application context* – Application context is formed by components present in the application. It is instantiated according to the definitions in XML files supplied to the test case.
- *Transaction support* – Declarative transaction support is available to the test cases. After each test case, the transaction is automatically rolled back (if not stated otherwise), providing consistent data for each test case and also faster execution speed, because the database does not have to be initialised over and over again.
- *Application context caching* – TestContext framework caches all created application contexts based on the XML definition file names used to create them. When more tests share the same application context, TestContext does not create the new context, but re-uses the old one. With larger application containing hundreds of test cases it can lead to speed increase in order of magnitude.

4.5 Monitoring

Once the application is deployed and run, it is often required to monitor that the application runs properly. If problems occur, it is very valuable to have diagnostic data that help to fix the issues quickly.

4.5.1 Logging

First, and very often used mean of monitoring the application is logging. Developer-supplied diagnostic messages are produced while the application is running. The messages are usually stored into plain text log files, but can also be stored to database, sent to another machine over network or by e-mail.

Logging can be implemented, in a naive way, by printing directly to standard-output with Java `System.out.println()` method. While this solution works, it is not flexible like to use a dedicated logging framework. Many logging frameworks are in existence nowadays, for example *java.util.logging*, *commons-logging*, *log4j*, *Logback*, *SLF4J* and others. They differ in usage and capabilities, but the general ideas are the same. The logging framework

consists of two parts, front-end API and logging backend. The API is the interface used by the developer to dispatch a logging message to the framework. The messages that contain a message source, message text and importance level are submitted to the logging backend, where they can be filtered upon various criteria, stored to files etc. The idea is that the logging backend is loosely coupled to the API and can be configured independently on the application.

The choice of the logging framework was based on analysis of Spring Framework, Hibernate and other related libraries. Spring Framework uses *commons logging* as its logging implementation, but as the reference manual states [10], it is only for historical reasons. Commons logging has got class-loading issues due to its dynamic discovery of logging backend [23] and is not recommended to use it for newer projects.

Hibernate and other libraries made a transition (or use from the start) a *Simple Logging Facade for Java* (SLF4J). SLF4J is a lightweight facade that allow various logging frameworks implementations to be plugged in at deployment time. Almost any major logging framework can be used as a logging backend to SLF4J. Apart from logging backends, SLF4J distribution comes also with several wrappers to replace and redirect other logging frameworks to provide unified logging solution.

As a logging backend I have chosen Logback, which is a successor to the popular log4j logging backend and implements SLF4J API natively (thus without need for a wrapper). Logback is very customisable and sufficient solution for many logging and auditing requirements.

4.5.2 Run-time monitoring and management

Logging can be a valuable source of monitoring, auditing and debugging information. However, sometimes it is more appropriate to get the required information in real time or fine-tune the application parameters. Since Java SE version 5.0 the run-time environment contains *Java Management Extensions* (JMX) that allow the developer to monitor and manage the running application. Integration with Spring is easy because of existing support within the framework.

Another aspect of application monitoring and management is performance monitoring. Precisely placed performance monitoring can discover application bottlenecks. *Java Monitoring API* (JAMon) is a very small and fast third party library just for that purpose. It acts as a aggregating database that has got a small memory footprint (saves only statistical information like last value, minimum, maximum, average etc.). JAMon is very fast and can be used even in production environment to search for bottleneck in a long run.

4.5.3 Scripting console

Scripting console is a tool that can compile an issue a script upon the running system. The inspiration comes from *Ruby on Rails* framework. The console is aimed for developers and system administrator, that can aid testing, debugging or disaster recovery. Almost any dynamic language that can be used in scripted mode on JVM can serve as a console's command language, for example Groovy, Scala, BeanShell, JavaScript, JRuby etc. In my work I intend to use BeanShell, because its syntax resembles Java the most and because the runtime libraries needed for the inclusion are the smallest (280 kB in a single jar).

It should be noted that the console poses a great security risk, it creates a backdoor to the system. The possible attacker can use it to invoke a great deal of damage. The measures to make the console secure should be appropriate – either to disable the console in production

mode entirely or enable secondary checks to the username/password authentication, such as IP address restrictions or SSL client certificates.

4.6 View technologies

Rich Internet Applications are essentially web applications that try to look like their desktop counterparts (see section 3.3). In order to present a responsive desktop-like application inside the browser, several view technologies has to be used.

Technically the application contains only two pages: the login page that forms the application entry point and the index page, where the client-side application written entirely in JavaScript is referenced and executed from.

The server part of the architecture is independent of the specific JavaScript framework or library. In fact, any library supporting AJAX and JSON can be used, for example *Sencha Ext JS*, *jQuery*, *Prototype*, *Dojo Toolkit* or others.

JavaServer Pages (JSP) In section 4.1.1 two main web components of Java Enterprise Edition were introduced: *servlets* and *Java Server Pages* (JSP). JSP is going to be used mainly for the login and index pages. The pages are going to be mostly static HTML with little dynamic data injected in them.

JSP be used also for dynamically generated JavaScript code, that is needed on application initialisation: the example could be dictionaries with localisation messages. These are needed only once after a successful login, so it is better to generate and reference them directly to load them together with the application.

JSON view After the application initialises (loads all the referenced JavaScript files), it begins to be available to the user. From that moment on, the web browser stays on the single page and retains the state of the application. All additional user data are loaded on demand by asynchronous (AJAX) calls to the Java web application.

The data are passed in *JavaScript Object Notation* (JSON), which is a lightweight data-interchange format that is easy to write and parse. Spring Framework contains components that can easily de-serialise and serialise JSON into Java objects using *Jackson Java JSON-processor* library, making them good candidates to use for the communication with the client.

Reports and exports Rich Internet applications from time to time need to present data in a format that is not displayable in a standard browser. A frequent example can be reports in *Portable Document Format* PDF or various exports into *Excel worksheets* (XLS). Spring web module is very versatile and not limited to a particular output. Almost any library can be plugged in to serve as a view layer, such as *Jasper Reports* or *Apache POI*.

4.7 Build system and dependency management

Building an application targeted for Java Enterprise Edition can be a very complex task. The application often consists of interdependent modules and has got many dependencies on external third-party libraries.

Integrated Development Environment, such as Eclipse or IntelliJ IDEA can be generally used as a building and development tool. Although this approach is very easy to setup, it

is effective only simple cases and for single developers or very small teams. If the project fulfil ls one of the following requirements, it is not feasible to use just IDE as the build system:

- The build process contains more complex tasks than building sources, such as source code generation or JavaScript obfuscation.
- Dependency management of third-party libraries is required.
- Automated builds done by a build server or continuous integration is required.
- The development team does not use a single IDE on a single operating system.

There are several dedicated build systems available, but in this thesis I will present the major ones: *Apache Maven*, *Apache Ant* and *Gradle*. More about build systems, automated building and testing and continuous integration can be found in [24].

4.7.1 Apache Maven

Apache Maven is a very frequently used build system, especially in the open source world.

The configuration is done by writing XML files. The build logic is based on the idea of fixed *lifecycle*, to which processes can be registered. The actual processes are invoked by special Maven plugins and are configured declaratively inside the XML. Maven has got support for various tasks, such as compilation of sources, automated testing, launching development server and much more. The developers are free to implement their own plugins. Thorough description can be found in [25].

Maven contains extensive support for the dependency management. Dependencies are firstly defined and later fetched from a dedicated *Maven repository*, where many libraries and their sources are stored. The repository can be a public one like *Maven central repository* or a private one built inside the company's firewall with tools like *Sonatype Nexus* or *Artifactory*. The libraries inside the repository are stored as several *artifacts* (library jar, source jar, etc.) and a definition file, where its configuration and dependencies are stated. This system allows the handling of chained transitive dependencies.

The declarative way in the XML file is a major drawback. If the project does not fit into the guidelines and the lifecycle that Maven specifies, it becomes a very tedious job do make a way around it.

Personally I have examined Maven on the prototype project. It worked, but with several problems. The main one was the IDE integration. Although major IDEs contain plugins for Maven integration, the development is slow. To run the application, the whole build has to be done: the application is compiled, packed to its artifacts (jar, war, ear) and then deployed into the application server. With multi-module application the build process gets even slower. Therefore I would not personally recommend Maven as a tool for quick and agile development process.

4.7.2 Apache Ant with Apache Ivy

Apache Ant is a very old and mature tool for building Java based applications [26]. It can be compared to the *make* utility found almost on every Unix platform. The build process is based on *tasks* that can have dependencies among each other. The tasks' content is defined in XML files just like in the case of Maven, but in an imperative way (the XML tags are

evaluated and respective actions executed one by one). The processes connected with the XML tags definitions are defined inside *Ant tasks* (similar to *Maven plugins*).

There is no notion of fixed lifecycle in the Ant builds. The system is based purely on tasks that are invoked in the order of their dependencies.

Apache Ant itself does not contain means of dependency management of libraries. However, another project from the Apache Software Foundation, the *Apache Ivy*, serves as a dependency management solution for the Ant builds. Apache Ivy management of dependencies works in a similar way to Maven's one. As artifacts repository, various sources can be used: for example http, ftp, ssh, or even a maven repository.

The IDE integration is without problems, no special plugin is required. Since Ivy copies all required libraries into the specified „lib“ folder, the only action to make the project working properly, is to include the folder into the application classpath.

The build of the prototype project was for a long time based on the Apache Ant and Apache Ivy. The custom build logic was relatively easy to implement without the need for implementation of specific Ant tasks. However, with increasing complexity of the build, the tasks' definitions became hardly maintainable.

4.7.3 Gradle

Gradle is a relatively modern build system based on a *domain specific language* implemented in *Groovy* programming language. It quickly gained popularity and was adopted by large projects, such as *Spring Security* and *Hibernate*.

The Gradle's domain specific language provides a powerful abstraction of the build process, greatly reducing the amount of code that need to be written.

For the dependency management the API of Apache Ivy is used transparently to the developer. That allows to configure dependencies and repositories like in the case of Ant builds (see section 4.7.2). Apache Ant can also be easily used inside Gradle builds.

The integration with IDEs are realised by project files generator. Currently, Gradle natively supports Eclipse and IDEA. However, the approach of Apache Ivy, putting all libraries into the folder, could also be used.

The major advantage compared to Maven or Ant is, how Gradle handles multi-module projects. Both Maven ant Ant process modules in a completely separate way, only with the knowledge of the order in which the modules are supposed to be built. Gradle constructs the tasks' dependency tree for all modules before the start of the actual build. The feature results in significant speed up of the build, since only relevant tasks are processed.

During the prototyping phase of this thesis, I migrated the Ant based build into Gradle. In the end the definition and configuration files became much more clear and readable. The migration was easy to implement even without prior knowledge of the Groovy language The Gradle's ability to invoke Ant tasks and Ant builds made incremental migration possible. The ease of use and readability of the build scripts are the main reasons why I am going to use Gradle as a build system in the final version of this thesis.

4.8 Deployment and runtime

Web application based on the Spring Framework consists of many POJO components wired-up together. The application is lightweight in nature and does not require full application server according to Java EE specification, but only a servlet container (although the application can be also deployed into the Java EE compliant server).

Two most used lightweight servlet container used nowadays are *Apache Tomcat* and *Eclipse Jetty* (former *Mortbay Jetty*). Both server are comparable in features, but each of them is suitable for a different situation.

Apache Tomcat is more appropriate for the production use. The server is integrated together with an embedded database connection pool (*Commons DBCP* or *The Tomcat JDBC Connection Pool*) and a manager application that helps with deployment operations. Apache Tomcat also serves as a foundation for commercial servlet container *SpringSource tcServer* that contains additional management extensions such as deployment to cluster of servers, detailed performance monitoring or protection against memory leaks.

On the other hand, *Eclipse Jetty* is more suitable for iterated and agile development inside an IDE. The reason is the possibility to embed the server directly into the web application and run it and debug it as a Java Standard Edition application. Jetty does not contain any integrated database connection pool, but almost any pool providing a `DataSource` can be used (such as *C3P0*, *Commons DBCP* or *Proxool*).

However, Apache Tomcat in its recent version (7.0.8 as the time of writing), has introduced helper classes that aid the developer into embedding the server in the similar way as the Eclipse Jetty. Therefore, the Apache Tomcat can be used in embedded mode to ensure the consistent environment.

4.9 Summary

This chapter discussed various technologies related to the web applications in Java Enterprise Edition environment. The following list summarises the libraries and technologies chosen for the final application of this thesis:

- Runtime environment is going to be *Java EE 6*.
- The architecture is going to be based on the *Spring Framework*.
- Persistence solution will be according to the *Java Persistence API* specification with *Hibernate* as its implementation. Queries are going to dynamically constructed by *Querydsl* library.
- Security solution for the web application security will be *Spring Security*.
- *Apache Tomcat* as an embedded development server and also the standalone production server. *Tomcat JDBC pool* will serve as a connection pool to the underlying database.
- The application will be tested with *JUnit* tests, *SLF4J* and *Logback* will serve as a logging platform.
- Build system will be Gradle based with Maven repositories serving as a source of the artifacts.

The choices of technologies were based on one-off prototype projects that were used to extensively test the features of the respective libraries. The final solution was chosen as a best fit for the Rich Internet Applications platform.

Chapter 5

System architecture

In previous chapters 3 and 4 overall requirements for the RIA platform and technologies to make its realisation were stated. In this chapter basic building blocks and components of the architecture and communication among them will be presented. The key points of the platform design are:

- Simple, loosely coupled components. The component should be designed for one specific task only.
- Complex tasks that are beyond single component's scope are modelled using composition of components into larger systems.
- Data formats used for the communication among components should be as uniform as possible. Uniform data structures allow greater degree of separation.

The meaning of the mentioned key points will become more apparent in the following sections, where main components are going to be described.

5.1 Tiers and layers

Enterprise applications written in Java are very often divided into separate tiers (see section 4.1). Tiers divide the application into distinct parts that can be run on one or more machines. While *tiers* represent the organisation of system parts on a *physical level* (machines/servers), *layers*, on the other hand, divide the system on the *logical level*. The separation to tiers and layers in the proposed architecture is shown at figure 5.1.

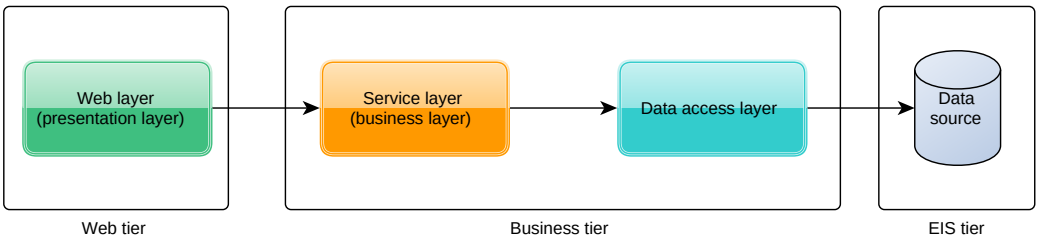


Figure 5.1: System's tiers and layers

The purpose of the system logical layers is:

- *Web layer (Presentation layer)* – The layer provides presentational data and accepts input from/to the client. For example it can provide HTML pages or JSON data based on a set of parameters or parse input from a web form. Validation of data passed from the user happens in this layer.
- *Service layer (Business layer)* – This layer contains the main business functionality of the application: business entities that model the real life business objects (customer, order, order item) and business components that handle business processes and workflow upon these entities. I will refer to this layer as „service layer“, although the name „business“ layer is equivalent.
- *Data access layer* – The data access layer creates a level of abstraction in order to have interfaces to access various underlying data sources or remote services (relational databases, XML files, web services, etc.) The interfaces provided to the service layer do not contain the information, how exactly the data will be fetched. It allows the developer to swap implementations of the data sources by rewriting the data source layer, all components in the service layer stay the same.

5.1.1 Data and inter-layer communication

Business components and service layer form and model the business processes, which are the most crucial part of the whole application. However, processes themselves need some data to operate on. In service/business layer data are represented by business entities (or entity objects) that form the domain model.

Entity objects are used as a mean of communication between the service layer and the data access layer. If we for example consider simple data storage and retrieval, the requests to the data access layer would model the operation of saving and retrieving entity objects to the database (save the customer, get list of orders and so on).

However, the data format used for the communication between the service layer and presentation layer are different. One can use entity objects to pass the data pack and forth, but for the majority of cases it is not applicable. Instead of entity objects a separate set of objects called *data transfer objects* are going to be used. Main reasons to introduce the data transfer objects are:

- The major concern is security. By freely providing the entity objects outside the service layer (meaning to the upper layers), one can unintentionally expose data that exist solely for the internal purposes of the service layer. In enterprise systems, where information can be provided by web services to third parties, it is an issue.
- The application's own security model can serve as an argument for the objects separate from the entity objects. The attributes of the entity objects cannot be filtered, because they are static part of the entity class. Therefore, they can also contain an information/security leak.
- Another concern is from the technological point of view: the independence on the underlying object-relational mapping (ORM) solution. Take Hibernate as the concrete example. To facilitate advanced mapping techniques like lazy loading and dirty checking, the library creates a dynamic proxy class that is different from the actual entity object. It can cause issues with serialisation (if the object is taking part in a remote call) or cause lazy loading exceptions (because the underlying session is no longer available) that are very hard to find.

The data formats that are going to be used in the final architecture are summarised in figure 5.2.

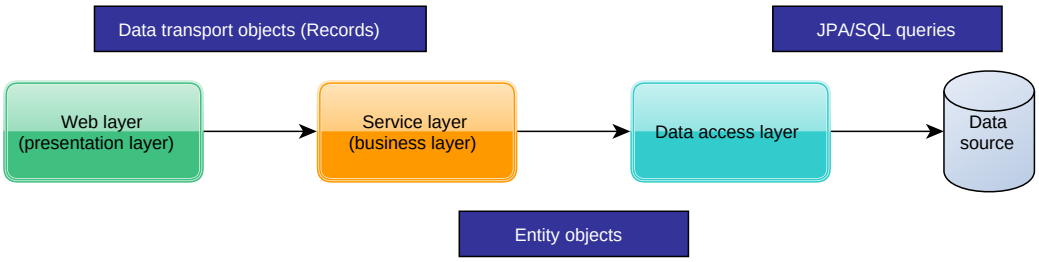


Figure 5.2: Data formats used for inter-layer communication

5.2 Key components and their interaction

Important aspect that needs to be mentioned before the analysis and splitting the system into communicating components is to specify, how exactly will the user interact with the system and what are the usage patterns and use cases. Properly identified use cases can be very helpful during component analysis.

Rich internet applications can be used to model large field of real world situations. The platform proposed in this thesis should cover the applications of broad variety, therefore the key components have to be generic in their nature and cover the most usual tasks in the information systems.

As their name imply, information systems are meant to contain and process information, thus data storage and retrieval operations are very frequent operations. Based on this assumption, the generic components will be mainly targeted to support *CRUD operations* (create, read, update, delete).

Implementation complexity of CRUD operations can significantly increase with growing number of entities needed to be stored in the system. Generically designed components would help to avoid repetition in the resulting code, lowering implementation complexity and also amount of possible mistakes in the code.

Nevertheless, enterprise information systems are very complex and their business logic is almost in every case beyond simple storage and retrieval of data and because of that the components should be very flexible, with possibility for customisation of their behaviour.

Considering all of the arguments mentioned in this chapter, the high-level view over the component architecture is shown in the figure 5.3. Components and their functions and responsibilities will be described in great detail in the following sections, grouped by the logical layers.

5.2.1 Web layer

Web layer is built on top of technologies like *Java Servlets* and *Spring MVC* to provide a thin adapter between the client application in the web browser and the business logic on the server side. The foundation is formed by the *Model-View-Controller (MVC)* architectural pattern [2].

Model is represented by the underlying service layer. Calling the services and business method is the only way to obtain or store the modelled data.

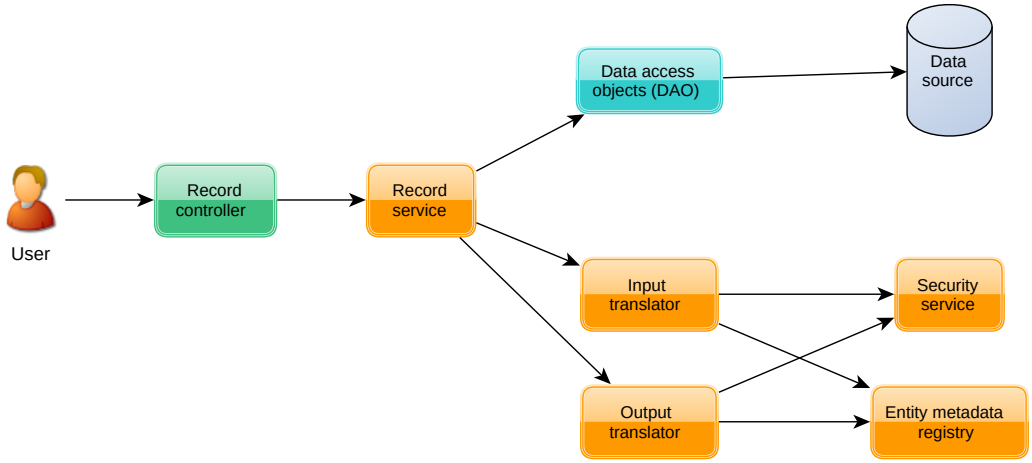


Figure 5.3: CRUD-related logical components. Web layer is shown in green, service layer in yellow and data access layer in blue.

Views contains the actual presentation logic (e.g. how to render a page). There are two common views in this architecture: first, delegated to a JSP page which can be used to provide the index (main) or the login page, and second, JSON view that serialises the model into JSON to be processed in the AJAX request.

Controllers are the entry points to the application. Each time user does an action in the web browser, the browser issues an HTTP request that is processed and dispatched to a specific controller method depending on the used URL address. Generally, the controller is responsible to get or put data to the model (represented by the service layer in this case) and delegate the presentation to the appropriate view.

Record controller *Record controller* is the controller for handling the create, read, update and delete operations (CRUD). Its design is very lightweight: the main task is to gather and then pass the request data to appropriate *record service*. No advanced binding or validation are done at this point, the data are merely passed to the service layer in the form of data transfer objects.

5.2.2 Service layer

Section 5.1 has described service layer as a place where most of the business logic is situated, what the application actually does. Service layer is the centre point: web layer serves as a connector to the user using the web browser and HTTP protocol and data access layer is at full disposal to satisfy the data storage needs.

Next paragraphs depict the nature of the key components of the service layer. They are described in a bottom-up fashion from the most basic to the most complex in order for the dependencies and relations are visible and clear to understand.

Entity metadata registry Entity metadata registry, as its name suggests, is a repository of application's entities metadata. The general idea of the metadata registry is related to the *application tree* in the legacy application that was mentioned in the motivation chapter 2. Each entity registered and present in the application is described by a distinct set of metadata that specify and identify it:

- *Symbolic name* – the unique identifier of the entity that is mostly used as a lookup key for searching in the metadata registry. *Example:* `Person`.
- *Java type* – the Java type of the entity class. Each entity is associated with a single Java class. Java reflection API can be used to introspect the type and scan for additional metadata. *Example:* `com.example.Person`.
- *Attributes* – attributes identify the properties of the entity. They need not to correspond to the attribute of the entity class, some can be left out, others inserted (as derived or computer values that are not present in the object). Attributes are essential to the security model, because permissions are mapped onto their names and checked during the read/write operations. *Examples:* name, surname, address.
- *Permissions* – special permissions that are out of the scope of any attribute. They are purely for use in the business logic for additional security checks on specific operations. *Examples:* generate invoice, upload file.
- *Views* – as attributes are the base of the security model, views are the base for the translation between the entity objects and the data transfer objects. Each entity can be looked upon from different perspectives (views) of how it can be translated. In each view the different subset of attributes can be specified. *Examples:* list view, detail view, cut view.

Security service Security service is a high-level abstraction of a *role based access control (RBAC)* security model used in the application. The RBAC scheme is used together with five permission types: *read*, *insert*, *update*, *remove* and *admin*. The combination proved to be sufficient to cover many use cases, although the model is easily extensible to other permissions should the need arise.

The main purpose of the security service can is to verify, whether the current user has or has not the specific permission type. The actual representation and storage of permission information is left completely on the application's business logic implementation.

Input translator Translating the input data into the entity objects is the responsibility of the *input translator*. Input data are in most cases represented by a hash map, where the key is the attribute name and the value is the attribute value to be bound to the entity object.

Input translator binds the map on the object with the reflection capabilities of Java language. The binding operation has to be quite sophisticated, as conversion of data types can occur and permissions to every written attribute have to be checked through the security model.

Output translator The *output translator* is the exact opposite of the *input translator*: it converts entity objects into the data transfer objects called records. Records are very similar to untyped hash maps with additional metadata (like permission information).

The conversion is controlled by a view definition inside the entity metadata registry. Therefore, the output translator needs to know in front, what view has to be used for the translation. The process can be described in three steps:

1. Name of the attribute is determined from the entity metadata registry.

2. Read permission is checked with the security model. If the user does not have the required permission, the attribute is not written to the output record.

3. Attribute value is read and examined:

   (a) If it is a primitive value (number, string, date, ...), it is written into the output record with optional conversion.

   (b) If it is a reference to another entity, entity metadata registry is looked up for the referenced view and the translation process recursively descends.

Since the translation process is recursive, caution should be taken while designing the view definitions to avoid cycles that would lead to infinite loops at run-time. Other safety measures can be taken, such as limiting the maximum recursion depth or doing cycle detection prior to the translation process. Each of the methods has its drawbacks and should not be mistaken for the proper cycle-free design of the entity views.

Record services The component that integrates all previously mentioned components on the service layer is a *record service*. For each entity, a single instance exists. While this approach is not entirely generic, it allows for much greater degree of customisation than if only one component for all entities would be present. However, with the increasing amount of record service implementations, the coupling between them and the record controller on the web layer also increases, because the controller has to reference all known implementations. To lower the coupling between the web layer and the service layer, a *record service facade* can be introduced. The facade would serve as a lookup table to the record service implementations. That way the controller can reference the record service by name, instead of by reference, and the coupling level would stay low.

The processes inside the record service can be split into two categories: input and output. Input processes handle data that have been received from the web layer, translate them using input translator and do data storage operation through the data access objects. The output processes use data access objects to read the entity objects from the data source and translate them to records with output translator. Every part of the process is checked by the security model.

Detailed workflow of input and output processes is shown in diagrams 5.5 and 5.4, which will serve as a guideline for in the implementation phase.

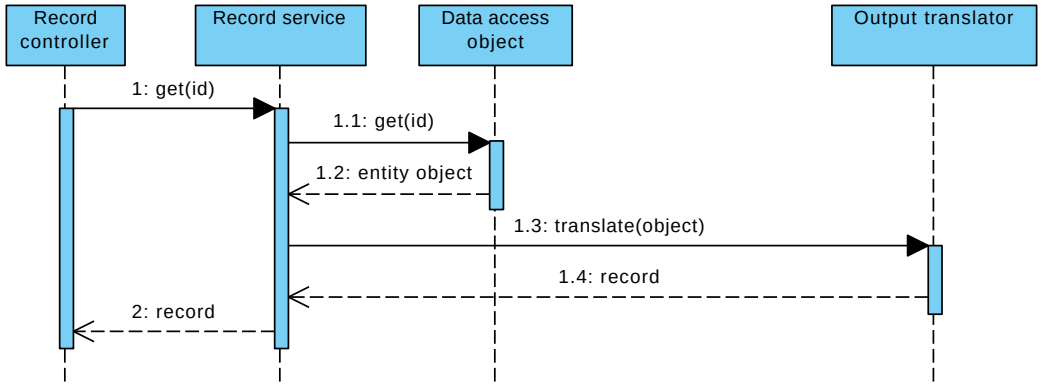


Figure 5.4: The output process (read). Security model is not included in the diagram.

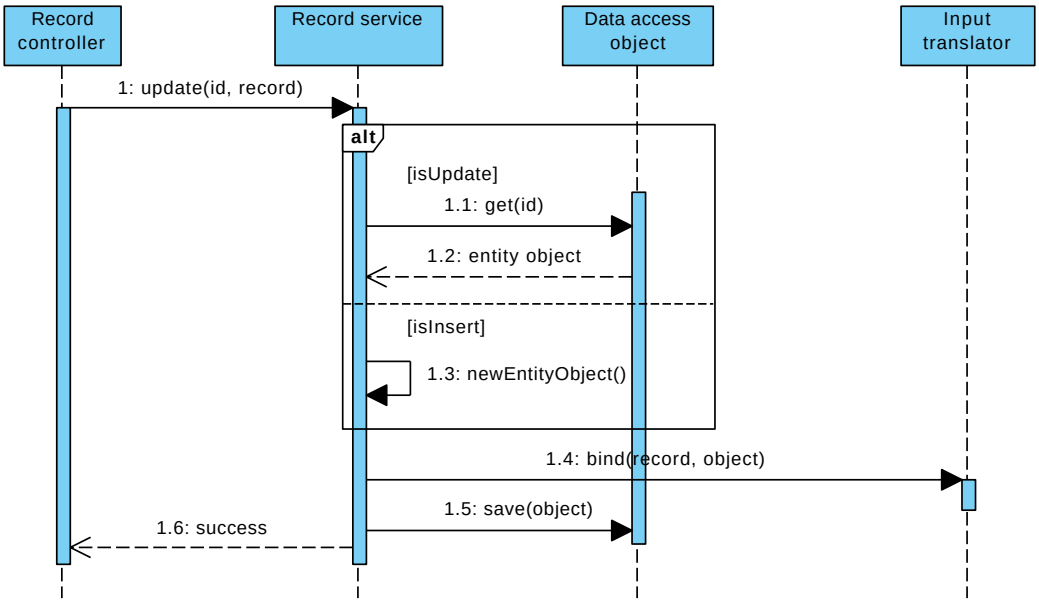


Figure 5.5: The input process (insert, update). Security model is not included in the diagram.

To summarise, declarative security model, views based on metadata and record service are very powerful concepts for the purposes of JavaScript-based Rich Internet Applications. Data can be presented in various views, depending on the amount of information required by the request. Since the output from the output translator is composed out of nested hash maps, it can be easily transformed into JSON format that can be processed by the JavaScript client.

5.2.3 Data access layer

Data access layer is composed out of data access objects and serves as an abstraction of access to the various data sources (section 5.1 contains more detailed explanation).

Data access objects (DAO) *Data access object* (DAO) is a design pattern that shields the business logic from the difficulties of accessing the data source, most likely a relational database. In the architecture, each entity that is itself persistable to the database is assigned one DAO encapsulating the data operations. Each DAO is directly referenced (through an interface) from the respective entity record service. This approach allow the developer of the application to implement custom data operations and queries more easily than in the case of a single data access object for all of the entities.

Chapter 6

Architecture implementation

In the previous chapter 5 the architecture the layers and their core components were introduced. This chapter discusses the implementation of the designed architecture. I am going to focus on the main ideas and issues that I encountered while coding the platform modules. The reference that includes the complete structure and information about the implemented packages and classes is left out and presented separately in the *Doxygen* documentation that is available on the attached CD (see appendix A).

6.1 Splitting the application into modules

The application is physically divided into four modules: *core library*, *business components*, *web application* and *development server*. The point was to split the code base into re-usable pieces that each of them performs the specific task. The modules are modelled as separate projects in the Gradle build system and in the IDE. The contents and responsibilities of the modules are:

- *Core library* – the library consists of components that are either required in other projects or are reusable across various implementations of applications. The examples could be the generic data access objects or default translators implementations. The package structure of the core library is shown in figure 6.1.

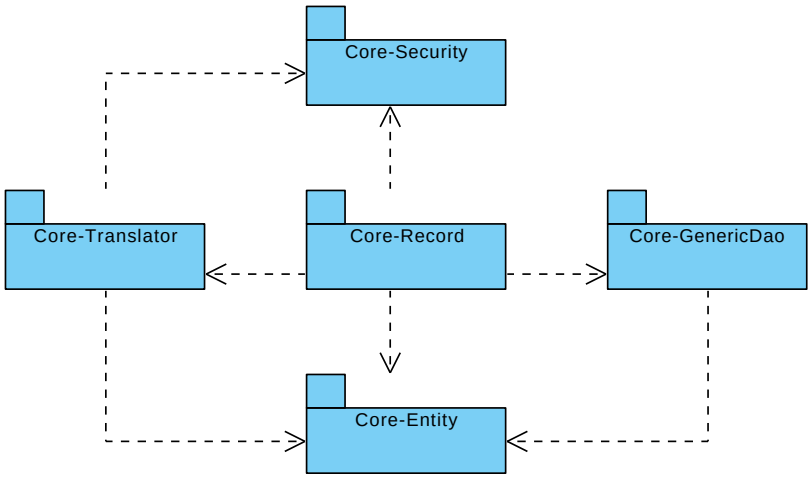


Figure 6.1: Package structure of the core library

- *Business components* – here the entity classes, record services implementations and other business-specific components are located.
- *Web application* – core library and business components are not tied to a concrete technology of presentation. This projects contains components that form a web application connector to the business logic. JavaScript front-end implementation is also part of this module.
- *Development server* – the embedded version of the Apache Tomcat servlet container is hosted here. It is used to run the web application without need to setup or use a separate servlet container or application server.

Figure 6.2: Application modules (projects in the build system).

The dependencies of the mentioned modules are shown on figure 6.2. Build of the modules are maintained by the Gradle build system (see section 4.7.3) with respect to their dependencies on external libraries and dependencies among the modules themselves.

6.2 Implementation of components

The foundation of the application is built with Spring Framework. The components are split into interfaces and implementations and wired together inside the Spring's application context. The application context maintains low coupling of components even when their amount gradually increase.

In the next sections several key implementations are mentioned with focus on important decisions made during the development. Concepts behind the following components were stated in section 5.2.

6.2.1 Entity metadata registry implementation

Implementation of the entity metadata registry is presented to the other system components as an interface `EntityMetadataRegistry`. During the coding process, several issues, that had not been previously anticipated, emerged. First the amount of metadata to be stored was increasing with features added to the core library and second the right format to define the metadata was changed several times to accompany the need for change.

Early implementations used custom *domain specific language (DSL)* implemented in Groovy programming language to define the metadata. The format was very compact and readable, but soon the disadvantages appeared: the DSL itself was hard to maintain and, the more tedious, it was hard to validate the correctness without launching the server. Another concern was from the dependencies' point of view, it was the only place where a code in the groovy language was used. Even when the metadata definitions were compact, it did not justify the disadvantages and increased amount of the dependencies.

Second format for the metadata definition was *JavaScript Object Notation (JSON)*, because I the JavaScript object mapper was in place because of the AJAX nature of the web application and the style of writing seemed compact enough. The issue was the complexity of the mapper and also the missing offline validation of the content.

As the third and final try I decided to use XML for the metadata definitions. Java SE includes a *Java Architecture for XML Binding* (JAXB) standard for easy XML to object binding [27]. The feature of JAXB that contributed for picking it as a metadata parsing technology was the ability to automatically generate the required classes out of the *XML Schema Definitions* (XSD). Two problems were solved at the same time: binding to objects and offline validation that could be done in any XML editor with the support for the standard XML schemes.

Schema and data model of metadata

The metadata and their data model were roughly described in the section 5.2.2, where main elements were declared: *entities*, *attributes*, *permissions* and *views*.

These elements were together combined and used to define an XSD, which was in turn transformed by the JAXB compiler (named XJC) to the set of classes that are depicted in the figure 6.3. `AppConfigMetadata` relates to the root element of the XML document, an *application config*.

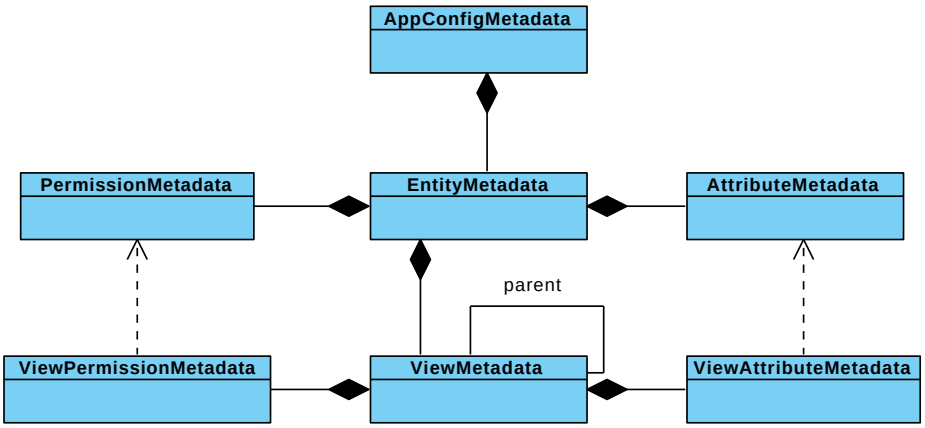


Figure 6.3: Entity metadata definition data model

The application configuration is composed out of `EntityMetadata` definitions that contain symbolic names and Java types of the entities. They also contain collections of `AttributeMetadata`, `PermissionMetadata` and `ViewMetadata` objects that further specify them.

Metadata definition for attributes and permissions are simply holders of data like property names or captions. In contrast to them, view metadata definitions are more sophisticated. They can be inherited from each other (with single inheritance only) and/or declared as abstract. This feature makes the XML files more concise. The view is composed out of `ViewAttributeMetadata` and `ViewPermissionMetadata` items that directly reference attribute or permission of the entity. The XML schema definition enforces that the references are all valid, pointing to the existing attributes.

The capabilities of the metadata definitions in XML files are clarified in the following example:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <app-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3    xsi:noNamespaceSchemaLocation="entities.xsd">
4    <entity id="Customer" class="com.example.entity.Customer">
5      <attribute id="name"/>
6      <attribute id="surname"/>
7      <attribute id="owner" reference="true"/>
8      <view id="common" abstract="true">
9        <attribute ref="name"/>
10       <attribute ref="surname"/>
11     </view>
12     <view id="detail" parent="common">
13       <attribute ref="owner" view-ref="cut"/>
14     </view>
15     <view id="list" parent="common"/>
16     <view id="cut">
17       <attribute ref="surname"/>
18     </view>
19   </entity>
20 </app-config>
```

Line 1 is an XML header with definition of the encoding. The following lines 2–3 define the root element `app-config`, which is linked to the `entities.xsd` schema via the special `xsi` namespace. Entity `Customer` and its Java class name is defined on line 4. The example document contain only one entity definition, but the amount can be arbitrary.

`Customer`'s attributes are defined on lines 5–8 by their IDs, which correspond with the property names of the entity class. Several views with various features are defined on lines 9–20:

- *Common* is the parent view. It is declared as *abstract*, it means it cannot be directly used, but other views can inherit from it. The view contains pointers to two attributes: `name` and `surname`.
- *Detail* view inherits the `common` view and adds another referenced attribute `owner`. The attribute is itself a reference to another entity (analogy to a foreign key used in the database) and must contain the `view-ref` XML attribute that states, what view is going to be used in the referenced entity during the translation process. For more information about the output translation see the section 5.2.2).
- *List* also inherits the `common` view, but does not add any new attribute reference.
- *Cut* view is used for displaying references in lists, therefore it should contain the digest of the information stored in the entity. The view does not inherit from the `common` view, but instead defines its own set of attributes, in this case single attribute `surname`.

Metadata registry

The implementation of the registry is split into two parts (see the class diagram in the figure 6.4):

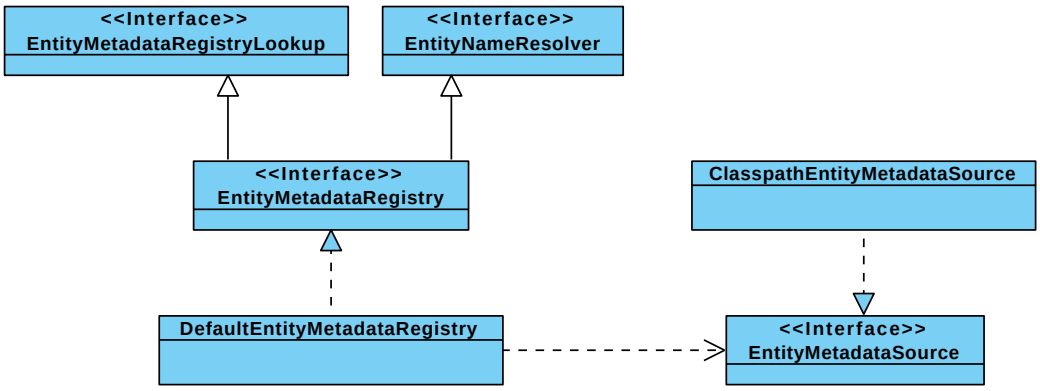


Figure 6.4: Entity metadata registry class diagram

- `EntityMetadataRegistry` – the registry that reads metadata from the metadata source. After reading and post-processing the results, several lookup tables for the quick search are formed and made accessible through the designated interface `EntityMetadataRegistryLookup`. Conversion map between entity names and entity types (class names) is also maintained, under the `EntityNameResolver` interface.
- `EntityMetadataSource` – the strategy interface to read the XML file. The only present implementation, `ClasspathEntityMetadataSource` reads the entities' definition XML files from the *classpath*. It means the files can be bundled to the application code in JAR archive and do not have to be distributed separately.

6.2.2 Security

Security subsystem is an integral part of the core library. Security checks are a cross-cutting that affect many other components in the system. The issue encountered during the implementation was, how to efficiently divide the components that do the actual security checks from the components that host the security model. The split was needed, because the main security components are a part of the core library, whereas entities like the user account or user role are part of the business logic. Introducing dependencies between the two, the circular dependency between the core library and business components would emerge.

The solution that was implemented introduces two new interfaces in the core library that are required to be implemented by the business components to support the authentication and authorisation:

- `UserAdapter` – Through this interface the core library can access various properties of the user object for the currently logged user account.
- `RolePermissionSource` – A service that retrieves the permissions from the data source in a data format independent on the business logic.

Authentication The complexities of user authentication is completely handled by the *Spring Security* framework. There are several authentication back-ends available in the framework (Web form, OpenID, LDAP, CAS, etc.), but for the sake of simplicity only authentication from the web login page is supported. However, the possibility to add more ways to authenticate the user are present, should the need arise.

In order to interconnect the Spring Security framework with the application, the interface `UserDetailsService` has to be implemented. The service has got a single task: to create a `UserDetails` object from the supplied user name that will contain information for password verification. In practice, the object is enriched as adapter (using *adapter design pattern* [1] [2]) with additional fields.

If the framework successfully validates the password, the user is logged into the system and the `UserDetails` object is stored in the *security context*.

Whenever the application need information about the currently logged user, it fetches it from the security context as an instance of `UserAdapter` or `UserDetails`. The class hierarchy is shown in the figure 6.5.

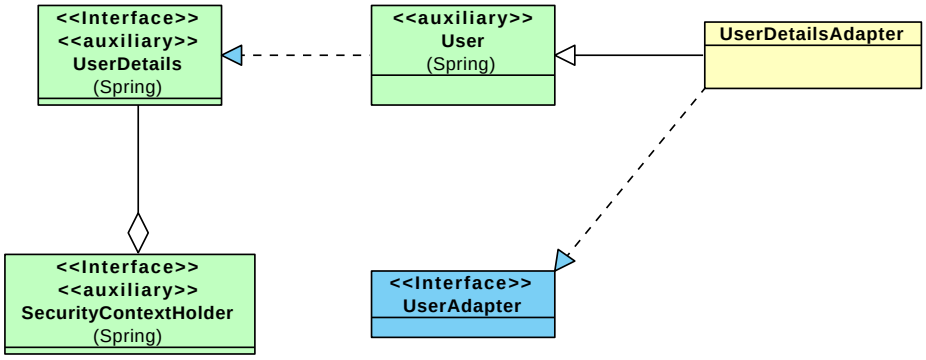


Figure 6.5: Class diagram for the authentication-related security services (business components are depicted in yellow colour)

Authorisation The connections among security-related components are visible in the figure 6.6. The high-level interface used by the other components for permission checks is `RolePermissionDecisionManager`. Its task is to decide, whether the currently logged user has got the requested permission or not (or in the other words, whether the user is *authorised* to do the action). The default implementation queries the `RolePermissionSource` for the permissions of the currently logged user. As it was previously mentioned, the role permission source is implemented inside the business components. There are two classes that serve as data carriers: `RolePermissionKey` that holds the entity name and permission code and `RolePermissionMask` that contains the permissions for the key.

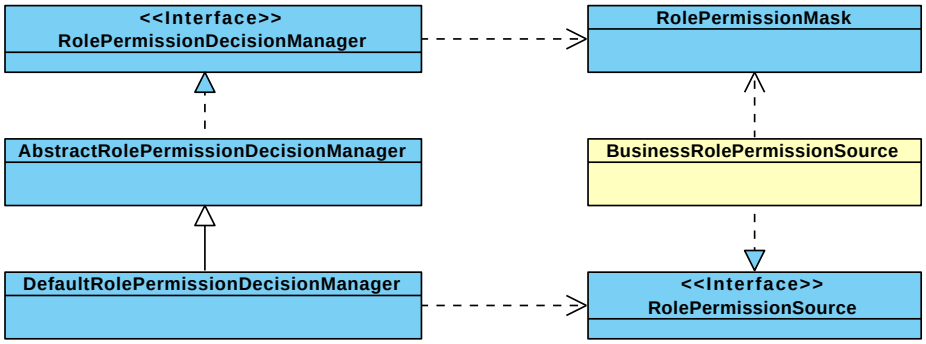


Figure 6.6: Class diagram for the authorisation-related security services (business components are depicted in yellow colour)

6.2.3 Generic DAO

Generic *data access objects (DAO)* form the *data access layer* of the application. The implementation is based on *Java Persistence API (JPA)* (see section 4.2.5), but since the data access objects provide an abstraction from the concrete persistence technology, JPA can be changed for an alternative persistence technology.

Data access objects implementations are in structure very similar to the record services. Each entity has its own interface and implementation of the DAO. Again, in simple cases, no additional code is needed, if the implementation inside the base class `AbstractGenericDao` is sufficient.

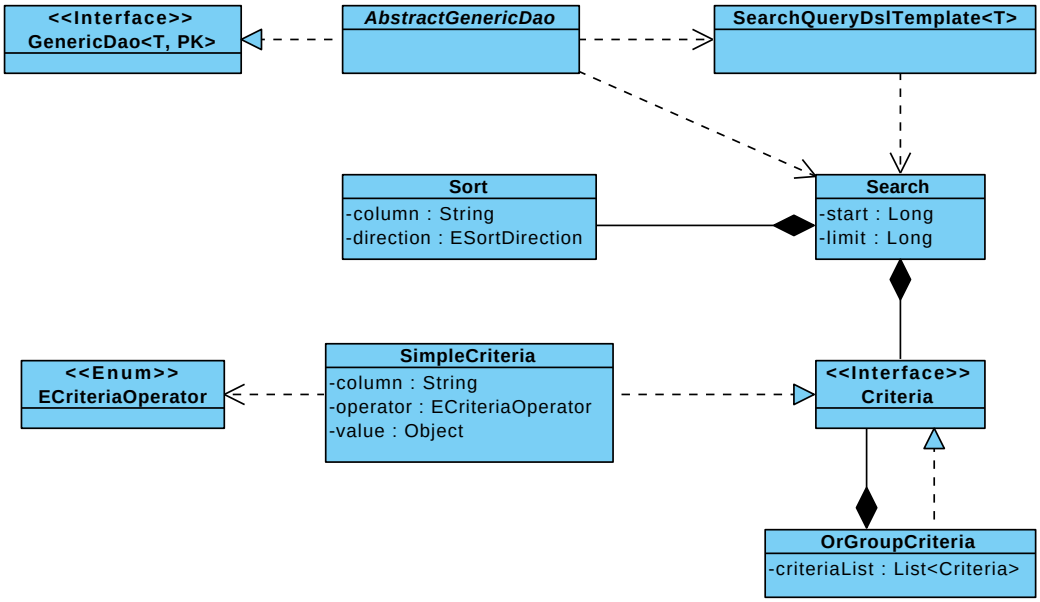


Figure 6.7: Class diagram of base classes for the generic data access objects (DAO)

Insert, update, delete and get by ID operations are simple, they are directly mapped to the methods of JPA `EntityManager`. The operation, that is more sophisticated, is getList. It retrieves list of entities based on dynamic criteria through the `Search` object.

`Search` class represents a flexible way of doing dynamic queries. Its instances serve as a data transfer objects that contain sufficient information to create a database query (see diagram in figure 6.7):

- *Criteria* – criteria are means of filtering the result list. In their basic implementation, called `SimpleCriteria`, the filter can be created using *column*, *operator* and *value*. Operators are grouped in `ECriteriaOperator` enumeration type and includes many operators (equals, greater than, less then, ...). By default, criteria are grouped together with *logical AND*. To achieve grouping with *logical OR*, criteria need to be placed into a container inside the `OrGroupCriteria` class.
- *Sort* – returned list can be sorted by several columns, either in ascending or in descending order. The `Sort` class models this capability.
- *Start and limit* – sometimes it is desirable to limit the number of query results (for example because of paging). Start specifies the number of the first row to be fetched from the data source, limit specifies the maximum count of rows returned.

The construction of the query from the search object is shielded from the business logic and done by the class `SearchQueryDslTemplate`. As its name implies, it uses the *Querydsl* library (see 4.2.6) to do the job. It supports and handles all aspects of the search data type like right grouping of conditions by AND/OR operators, ordering by `Sort` and limiting the number of results. The resulting query is executed (right after the query is constructed) and the list of entities is retrieved.

6.2.4 Record support

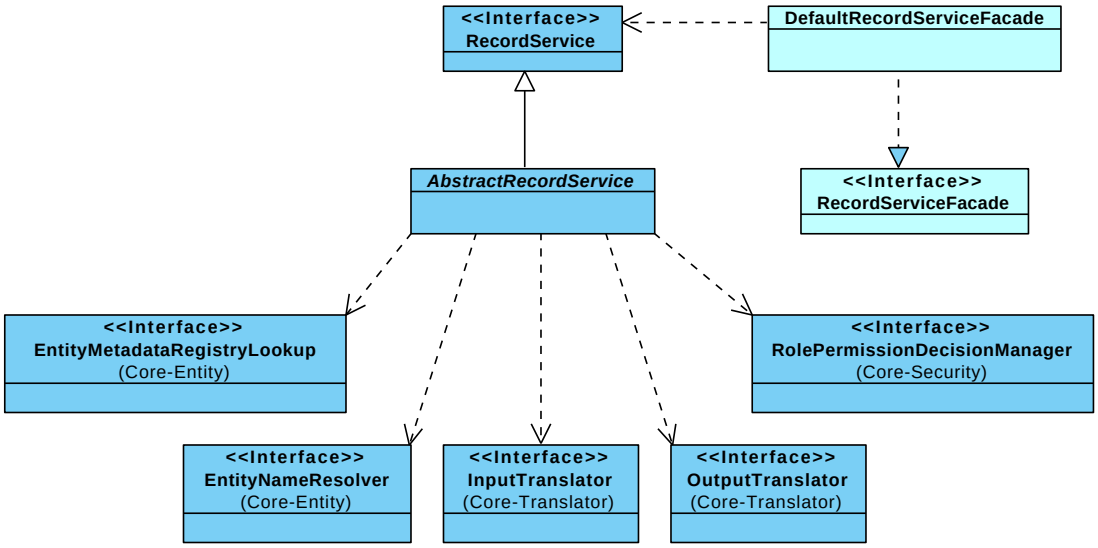


Figure 6.8: Record service and translators (class diagram)

Support for handling records is implemented around the base interface `RecordService`. The implementations are the integral part of the business logic, not the core library, because they depend and handle entity objects that are specific to the application. However, the common functionality is contained in `AbstractRecordService` class, which serves as a template implementation. The common workflow is implemented there, leaving only minor need to override and re-implement methods in the concrete implementations. In simple cases, the concrete implementations can be completely without any additional code.

Record services use `InputTranslator` for binding of input data for the purposes of inserting and updating records. Binding is done through a class `SecureDataBinder`, which is an extension of Spring's `DataBinder` with added security and permissions checks. Any violation of the security model results in an access denied exception.

The interface `OutputTranslator` is used for the conversion between entity objects and data transfer objects during the data retrieval (for example fetching a record or a list of records). After the attribute is found susceptible for the output by the `EntityMetadataRegistry`, it is checked by the security subsystem. Attributes without appropriate read permission are silently filtered out.

The search queries are passed to the record service as `Search` objects. These objects, after checked by the security subsystem (user cannot make queries on attributes without proper access), are passed to data access layer as they are or the further processing.

All record services present in the application are transparently registered into the `RecordServiceRegistry`, which are in turn referenced by the `RecordServiceFacade`. The

facade contain the similar operations as the record services do, only with additional parameter of entity name. That way the higher layers of the application (web layer especially) do not have to contain the logic to select the right record service implementation and only reference it by name while calling the facade.

6.2.5 Web application

The functionality of the *Model-View-Controller* pattern used in the web layer of the architecture was explained in section 5.2.1, this section focuses on the implementation.

The web application is based on Spring MVC library, which allows the developer to expose controller methods under URL locator. The special servlet named `DispatcherServlet` connects the Servlet API onto controller beans of the application context. The controllers are annotated with two annotations: `@Controller` on the class level and `@RequestMapping` on the methods that should be exposed.

After processing by the controller, the model data are passed to the named view. The view is resolved by the Spring's `ViewResolver` mechanism. It is either serialised into JSON by the *Jackson Mapper library* (AJAX calls) or delegated to JSP page for further rendering (index page, login page, ...).

6.2.6 Transaction management

A subject that was not mentioned in the previous chapters is *transaction management*. The implementation completely relies on Spring Framework transactions support (see section 4.1.2 about the Aspect Oriented Programming). In practice, it means that every component that should participate in a transaction, has to be annotated with Java annotation `@Transactional`. Spring's application context automatically introduces a proxy around all calls to the components' public methods and properly handle transactional state.

There are several options for configuration of the `@Transactional` annotation that change the behaviour of the transaction management system. Throughout the implementations, there are only used two of them:

- *Propagation = REQUIRED* – The transaction is required for the method execution of the underlying Spring bean. If there is no active transaction, it is transparently created and registered as active. If the transaction exists, it is reused. This type of transaction propagation is the default and is used on the *service layer*, where the transaction boundary starts.
- *Propagation = MANDATORY* – In this propagation mode, the transaction is also required for the bean method execution, but in case, the new transaction is never created. Instead, the exception is thrown. This behaviour is used in the *data access layer* on DAO implementations. It enforces the strict layer separation: the developer cannot call the data access layer directly without the service layer, because no transaction would be active.

While coding the application, the annotation on top of service implementation can be omitted by mistake. To detect this kind of errors, it is possible to use an own implementation of a `BeanPostProcessor`, which is an interface of Spring Framework that is invoked with every bean in the application context upon start-up of the container. There it can use Java reflection capabilities to check that the annotation is present and throw an exception otherwise.

Considering that there are services that are not related to any data source, thus they need not to be transactional, the post-processor also handles the own `@NonTransactional` annotation for that purpose.

Chapter 7

Evaluation of the development process

Previous two chapters provided a brief insight into the both architecture design and implementation, mostly of its core library. This chapter focuses on the benefits that the architecture brings to the developers building applications on top of it.

In order to demonstrate the power of the architecture, especially in the field of CRUD implementations, the process of creation of the new entity in the system is going to be presented together with explanation of benefits and issues.

7.1 Creating a new entity

Creating new entities and related components that handle the CRUD operations upon them is often a daunting task, where a lot of code repetition take place. This redundancy can reduce the maintainability of the product and can also be a source of errors.

In the architecture proposed in this thesis, the components that are required for the CRUD operations are designed in a way that lowers the amount of redundancy and code repetition. The extent of this generic and template behaviour is carefully balanced to not sacrifice the flexibility of the solution.

7.1.1 Entity class

As an demonstration, the simplistic version of the customer entity will be created. Firstly, the entity class has to be implemented in Java and annotated with JPA annotations to map the class to the persistent data source.

```
1  @Entity
2  @Table(name = "customer")
3  public class Customer {
4      @Column(name = "name")
5      private String name;
6
7      @Column(name = "surname")
8      private String surname;
9
10     @Column(name = "company_name")
```

```
    private String companyName;

    @ManyToOne
    @JoinColumn(name = "owner_id")
    private Person owner; // record owner

    // ... getters and setters were ommitted for brevity
}
```

7.1.2 Metadata

The next step is to create entity metadata needed for the record support, notably for the input and output translation. The metadata are defined as an XML document. For more information about the definition and the metadata related to the example, see the section 6.2.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<app-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="entities.xsd">
  <entity id="Customer" class="com.example.entity.Customer">
    <attribute id="name"/>
    <attribute id="surname"/>
    <attribute id="owner" reference="true"/>
    <view id="common" abstract="true">
      <attribute ref="name"/>
      <attribute ref="surname"/>
    </view>
    <view id="detail" parent="common">
      <attribute ref="owner" view-ref="cut"/>
    </view>
    <view id="list" parent="common"/>
    <view id="cut">
      <attribute ref="surname"/>
    </view>
  </entity>
</app-config>
```

7.1.3 Business components

There are two usual business components present in the system for each entity: the service that contains the business logic related to the entity and data access object (DAO) that handles the persistence of the entity. Both components are clearly defined by their interfaces:

```
public interface CustomerService extends RecordService<Customer> {
}

public interface CustomerDao extends GenericDao<Customer, Long> {
}
```

The CRUD-related methods for the service layer are defined in the inherited `RecordService` interface and in the `GenericDao` interface for the data access layer.

The service implementation inherits from the `AbstractRecordService`. Since it already contains the implementations of the CRUD methods, the customer service implementation is almost empty, except the link to the generic DAO.

```
1  @Service
2  @Transactional
3  public class DefaultCustomerService extends AbstractRecordService<Customer>
4    implements CustomerService {
5      @Inject
6      private CustomerDao fCustomerDao;
7
8      @Override
9      protected GenericDao<Customer, Long> getEntityDao() {
10         return fCustomerDao;
11     }
12 }
```

The DAO implementation inherits from the `AbstractGenericDao` class and its implementation can be empty.

```
1  @Repository
2  @Transactional(propagation=Propagation.MANDATORY)
3  public class DefaultCustomerDao extends AbstractGenericDao<Customer, Long>
4    implements CustomerDao {
5
6  }
```

Although no custom implementation is done in the mentioned components, the entity is now fully accessible from the client side through the *record controller* (see 5.2.1) via several views defined in the metadata. Translation to and from transport objects and serialisation to and deserialisation from JSON is handled transparently by the core library components. The access to the persistent entities is also encapsulated by transactions, as it was discussed in section 6.2.6.

7.2 Extending the components' behaviour

The entity class, entity metadata, record service and data access object are essential part of every entity that should support create-read-update-delete business logic. There is some amount of repetition present that can be further abstracted to provide a theoretically cleaner code, but it would lower the flexibility of the solution. This assumption is based on experience, where most of the business components required not only CRUD, but also the custom business logic. This logic needs to be implemented somewhere, possibly as close as possible to the CRUD-related actions of the entity. Therefore, the small amount of repetition can be justified, especially considering the amount of work that is saved by using the `AbstractRecordService` and `AbstractGenericDao` base classes.

Extending the components can be done in two ways: through overriding the existing methods (for example in the `AbstractRecordService`) or creating the completely new

ones. Overriding is feasible for the logic that is related to the CRUD operations, like „execute an action after the update of the entity“, whereas new methods serve as a point for more specialised actions. Either way, when a method is overridden or a custom method is added to the service or DAO interface, it begins to be automatically available to the entire application throughout the Spring's application context.

7.3 Generating the code

The business components for the entity contain small amount of repetition, which is more apparent during the creation process. Creating the required parts manually is a task susceptible to errors. However, the basic structure for the components can be generated by proper tools. This is similar to the *scaffolding* available in the *Ruby on Rails* or other frameworks.

The input for the generator is the entity class that is properly annotated with JPA annotations. From this starting point, other files can be generated:

- Basic version of the metadata XML. The attribute names can be easily inferred from the field names of the entity class. The views can also be created, but it depends on the specific needs required from the generator.
- Service interface and implementation, as shown in the example.
- DAO interface and (empty) implementation.
- JUnit test case verifying that both the service and DAO can be initialised properly.

The another benefit of business components code generation, apart from avoiding the repetition, is the fact that it enforces the uniform structure of the application code base. The empty implementations also serve as a scaffold for more complex ones.

When the developer familiar with the structure has to modify, inspect or debug code from the other developer in the team, it is easy to find the exact location, because the logic is contained inside the well-specified components. It will become more apparent after the example application is presented in the next section.

Chapter 8

Example application

In order to verify that the architecture for Rich Internet Application development is feasible for the task, an example application will be presented in this chapter. The goal is to develop an application that will serve as an information system for the web hosting company. The example should be close to a project for a small company that is starting its business and is low on budget. The system will not be feature-rich from the beginning, but well prepared for the features to be added as the business will grow.

The main intent is to show that the concepts introduced into the architecture design are working properly and aiding the developer to create a complex Rich Internet Application. Please note, that the example *should not* constitute a complete enterprise application suitable for the production use, but only *demonstrate the capabilities of the architecture.*

8.1 Specification and use cases

The model company *Example, Ltd.* is in the business of web presentation development and web hosting of the created presentations. There is need for an information system that would aid in servicing the projects. The web presentations can be hosted in on one of the dedicated servers that the company rents in the server housing centre. The management needs to keep track of hosted presentations, owned and managed domains associated to them and of payments for various services.

The information system in its first version will be only for the internal purposes of the company. It needs to be accessed only by employees. Each employee will have his/her own account and unlimited number of employees can access the system concurrently. Customer's access is not planned for the first version, but the information system should be designed in an extensible way, in case the feature will become requested in the future.

There are two known roles of users:

- *Employee* – the role that will have to do the evidence of companies, hosting companies, servers and other entities that are defining the information system.
- *Administrator* – the administrator role will have the same capabilities as the employee role, with addition of user account management and other system-level features.

The specified roles are not fixed and can be modified in the later versions of the product, if they become insufficient.

The following entities are required to be present in the system:

- *Customers* – the customers of the Example Ltd. that are supported with hosting services.
- *Presentations* – the web presentations. Each presentation belongs to a customer, is hosted somewhere (and with the specific technology), presented under a domain name (and aliases) and billed periodically. An employee is designated to be responsible for all communication regarding the presentation.
- *Servers* – servers available for presentation hosting. Administrator manages the list of servers and assigns presentations to them.
- *Technologies* – several web hosting technologies are supported, namely PHP, Java, Ruby on Rails and static HTML.
- *Billable services* – services that are billable. Payment for them is required from the customer.
- *Payments* – all received payments that for the services from the customer.
- *User accounts*, *user roles* and *permissions* – list of employees and administrators with their permissions that will access the system. These entities are going to serve as a source for the security model.

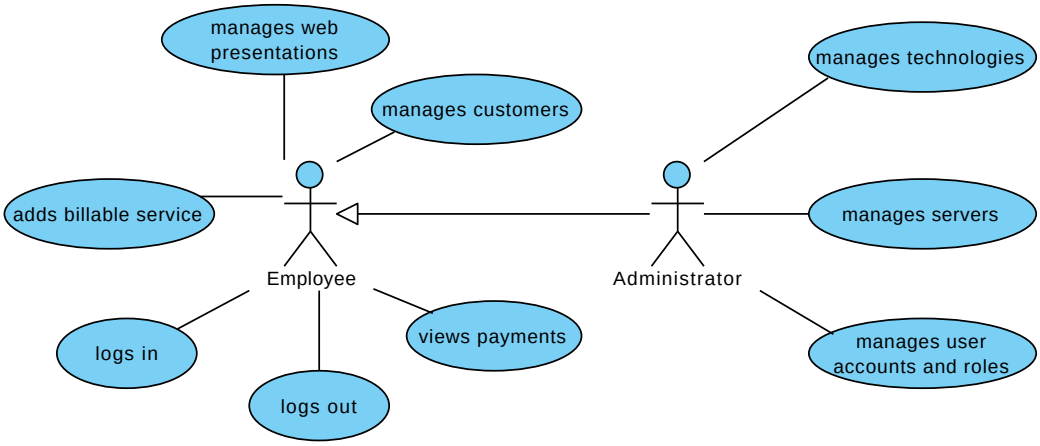


Figure 8.1: Use case diagram of the web hosting management application

8.2 Domain model

Domain model of the application reflects the entities and their relationships mentioned in the specification. The result is a class diagram that can be seen in the figure 8.2. The classes directly map to the database schema via Java Persistence API (JPA) and Hibernate.

Fields are annotated by JPA annotations to indicate relationships between entities, database data types and additional constraints. All entities inherit from the `BaseEntity` class that contain the common fields, for example ID primary key.

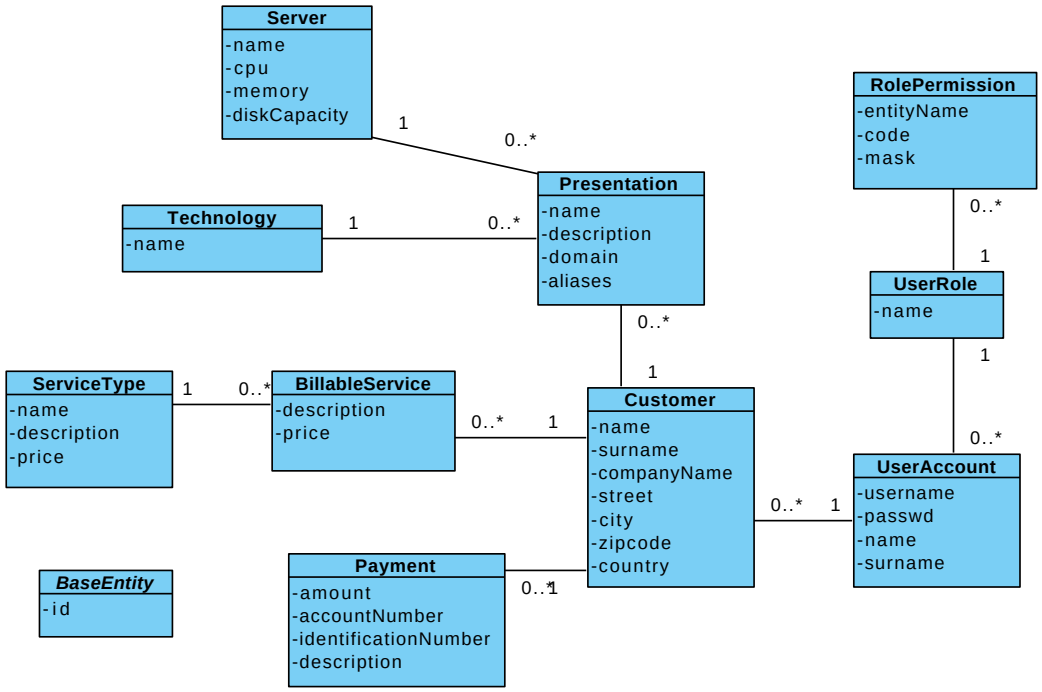


Figure 8.2: Domain model of the web hosting management application. Inheritance from `BaseEntity` is not shown for clarity.

8.3 Business logic

According to the section 5.2.2, for each of the entities in the domain model, several components and supporting files have to be implemented and written:

- *Record service* – record services that inherit from `AbstractRecordService` and implement the interface `RecordService` form the centre of the service layer and will contain all of the business logic.
- *DAO* – data access objects that retrieve and store entities into the database. The classes inherit from `AbstractGenericDao` and implement the interface `GenericDao`.
- *Entity metadata* – XML file to be read by the `EntityMetadataRegistry` that enumerates all available parameters and all views formed by them.

The record service and DAO are split into interface and implementation for the purposes of dependency injection. For example, for the entity class `Company`, `CompanyService`, `DefaultCompanyService`, `CompanyDao` and `DefaultCompanyDao` are all required to be implemented.

The component implementations are put into the XML definition file for the Spring application context. Spring Framework instantiates and wires the components together as the application starts up.

8.3.1 Security model

The security model is formed by the `UserAccount` and `UserRole` classes. Upon login, the own implementation of Spring Security's `UserDetailsService` is used to construct

the authentication object that is in turn used to verify the username and password. All passwords are hashed with salt for the increased level of security.

Permissions are tied to roles represented by the `RolePermission`. Service layer also has to implement the `RolePermissionSource` interface, where the permissions stored in the database are translated into classes readable by the core library.

8.3.2 Services, billing and payments

Customers pay for various services on monthly basis. The billable services, which represent the service that should be paid for can be created periodically by the application or manually by the employee. Periodic creation of billable services is implemented by means of Spring's `@Scheduled` annotation.

Each incoming payment is linked to the customer's account. It can be done manually or automatically by the payment's identification number. The payments have to be entered by hand, the seamless integration with the on-line banking system that would fill the payments automatically is not planned for the first version.

8.4 User interface

JavaScript based RIA applications can be implemented in various ways, but for the purposes of the proposed platform the resulting application will be a *single page application (SPA)*.

The presentation logic of the single page application is done, like its name suggests, in the single (index) page. Once the page is loaded, the complete state, updates, transitions and data requests are done through JavaScript engine and AJAX in the browser.

Designing the user interfaces (UI) can be significantly simplified by using tools for user interface rapid prototyping. *Balsamiq Mockup* is an UI sketching application. The output designs literally look like sketches, which has an advantage that the person evaluating them is solely focused on the overall structure and not on the small detail. The mockups can also be easily modified, making the iterative development of user interfaces very flexible.

8.4.1 Index page

The main, or index, page is a place, where all parts of the presentation logic are concentrated. It is divided into three regions, as shown in figure 8.3:

- *Header* – The information bar with the application name, currently logged user and logout button.
- *Menu* – The menu bar contains link to management of all entities in the application. The menu items are grouped together for clarity. Selecting the menu entry will result in changing and refreshing the content area.
- *Content* – The area for data presentation.

The data will be presented inside the content area. Each entity can be displayed in various ways – in different *views*. The basic two views are:

- *List view* – Contains list of entities. The user can add new or delete them from this view.

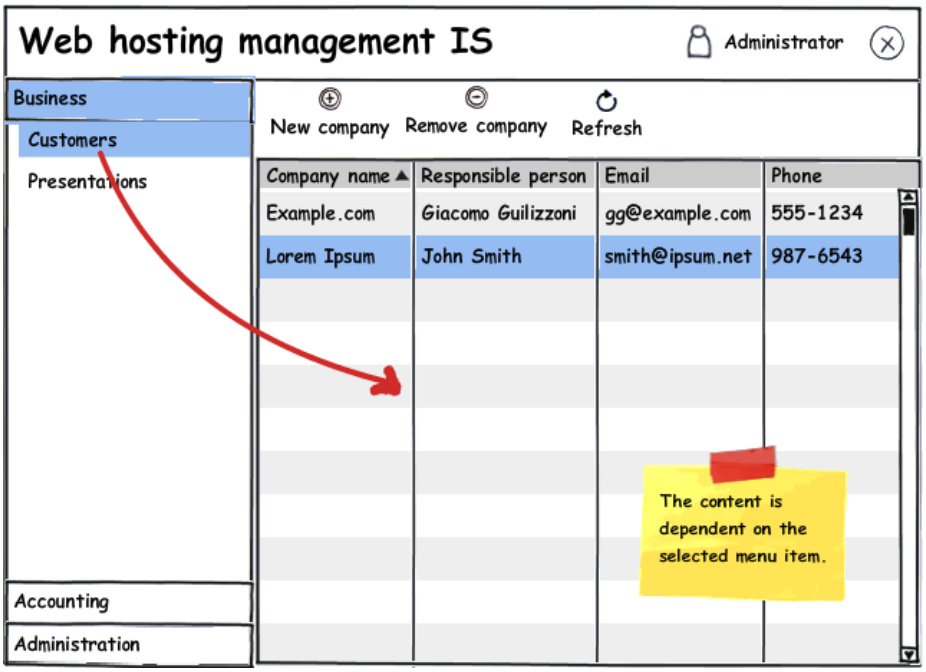


Figure 8.3: Index (main) page interface design

- *Detail view* – Detail of a single entity with possibility to do the full range of *CRUD (create, read. update, delete)* operations.

8.4.2 Login page

The index page is accompanied with a small login page that serves as the entry and exit point to and from the application.

The user is required to enter the user name (login name) and password. Upon clicking the *Login* button, the form is submitted and delegated to the Spring Security filter chain that authenticates the user. If the authentication is successful, the user is redirected onto the index page, otherwise the browser stays on the login page and error message is displayed.

8.5 JavaScript client

The client side of the application, which is mostly implemented in JavaScript, heavily relies on the *Sencha Ext JS* library [28]. It is a cross-browser rich Internet application framework with its own set of components, widgets and API.

The library and the own scripts containing the business logic of the presentation are loaded together from the index page. When all scripts are loaded, the objects that form the application user interface are initialized and displayed.

The application is stateful on the client side in the web browser, the state kept on the server side is kept to the minimum. In the example, only currently logged user is stored in a stateful session storage. The stateless nature of the application leads to better scalability and lower memory footprint, because the server side components can be singletons [1][2] with no shared state and can handle many users accessing the application concurrently.

Communication between the JavaScript client and server part of the application is

achieved through the AJAX HTTP requests. The structure of the data that are send and received is action specific with JSON being used as a serialisation mechanism. For the basic CRUD actions, the data are directly serialised transport objects from the output translator (see the section 6.2.4 for more information).

8.6 Database and run-time environment

In the section 4.8 the technology for the deployment and run-time was discussed and Apache Tomcat was selected as both production and development solution.

There are several differences between the production and the development environments and also issues that need to be resolved, but the general idea is, that the environments are as similar as possible to lower the number of failures in the deployment.

8.6.1 Production environment

The production environment is the one exposed to the customer. The most important aspects are *robustness*, *stability* and *availability*. The range of needed hardware can vary greatly – the application may be situated on anything from a single server to a large cluster with load balancing, caching proxies, separate storage, replication and automatic failover. The complexity of such setup is based on the load that the application has to face and also on the desired availability. The large scale deployment is out of scope of this work and only the basic case is handled here.

A relatively simple setup would consist of an application server (or servlet container) and a database server. They can run on the same machine, virtualised or on the physically independent machines.

The database server used in production can be any that is possible to use with a JDBC driver and Hibernate adapter. Both are supported by large number of vendors, one could for example use *PostgreSQL*, *Oracle*, *MS SQL*, etc. The database connections are managed by the application server and made available to the application through the *Java Naming and Directory Interface (JNDI)*. Often, a connection pool is used to recycle the connections and save the system resources.

The user can access the application directly on the port, where the Apache Tomcat server has been started. When the application goes down, either due to scheduled maintenance or a failure, it will be completely unavailable and seem like non-existent. That is why it is usual that the application servers are put behind a lightweight HTTP and reverse proxy (for example *nginx*). The proxy can delay the requests and wait for the application to become available, if the outage is short or can supply the user with an error page, informing about the service problems. Having the proxy server in the request chain is also perspective, because it can be used as a load balancer in the future, should the more application servers are put into place.

8.6.2 Development environment

As it was mentioned, the development environment should model the production one as closely as possible. However, this goal is difficult to accomplish and usually going opposite the ease of use and development productivity.

As the stability is essential to the production environment, the low *turnaround* and ease of use are the essence of the development environment. The turnaround is the time spent

while the change of the source code is reflected into the running application. Many aspects contribute to it, for example compilation, packaging and deployment of JAR/WAR/EAR archives, reload of the web application or a complete restart of the application server.

The more frequent the reload/restart events are, the more time the developer spends waiting. The development environment to be presented contains several features that aid the faster turnover time:

- The application server is embedded into the application as a library. The start-up process shrinks to an execution of the `main` method.
- The database server is also embedded: *H2* is a relational database management system written in pure Java that supports SQL language standard. Hibernate adapter is also available, therefore the *H2* can be used as an transparent alternative to other database management systems.
- There are components in the application that initialise the database from scratch (on demand, on every start). That way the developer has the most recent testing data and schema. In the early stages of development, when there is no version directly available to the customer, the initialiser can be used in a *drop-create* fashion, where even the schema (structure) of the database is recreated. It allows for painless schema changes and rapid prototyping of the domain model.

Another benefit of the solution is the very short time for introduction of new people involved in the project – developers, testers, management, etc. There is no need to install anything more then Java Development Kit (JDK), because:

1. application server (*Apache Tomcat*) is embedded
2. database server (*H2*) is embedded and database is automatically created and filled with data
3. *Gradle* build system takes care of the rest (download of required libraries, compilation of code, running the application)

The steps for building and executing the application in the development mode are discussed in the appendix B.

8.7 Testing the solution

The importance of testing not only during the development phase of the project was briefly described in the section 4.4. During the implementation of the platform many tests were created, both unit and integration ones. Their importance showed itself in late development stages when refactoring and changes to the core components had to be done. The test cases failed and avoided many mistakes and bugs that would be hard to find.

Writing tests before or together with components also proved to be very efficient. Running a simple test is quick and can be done repeatedly in a short amount of time, opposing the booting of full-fledged application server and manually testing the component's behaviour.

I decided to use JUnit as a testing framework for its simplicity and good integration into both *Gradle build system* and *Eclipse IDE*. There are three types of tests in the implementations. The types differ in the level of isolation from the other parts of the system.

8.7.1 Unit tests

Unit tests is the most isolated kind of tests. The component (or unit, hence the name unit tests) is tested without any dependencies. When the test failure occurs, the cause originates from the component itself. But components without dependencies are rare in the complex system. To be able to unit test the component, the dependencies has to be simulated by the test environment. There are several ways to do so [29]:

- *Using dummy objects* – Dummy objects are used in situations where the dependency is needed for the component to properly initialise, but not used in the actual test. Dummy object can be viewed as a placeholder, its methods are not called from any point of the test (if so, the exception should be raised).
- *Stubbing* – Stubs are testing implementations of dependent components that simulate the functionality of the real dependencies. For example stubs used in the business logic test instead of data access object can simulate data retrieval from a database without the need for a database connection. That way the test is isolated from side effects.
- *Mocking* – Mocks are very similar to stubs. The notable difference is that with mocks the behaviour of side-effects can be observed. Suppose the service calls data access layer to fetch data. The mock object in place can verify, which methods were called with specified parameters. The observation can be later verified to a set of rules and restrictions (method can be called maximum three times, first parameter should be specific value etc.). There are several mocking libraries available for Java, for example *Mockito* or *JMockit*.

The other possibility is to use the *real* dependency. In this case it would not be pure unit test, but rather a component test. Component tests are described in the next paragraph.

8.7.2 Component integration tests

Component test is a compromise between the unit test, which is completely isolated, and a full integration test that involves a large part of the system. The aim is to test a single component together with its dependencies to find errors in the communication among them.

In the implementation the tests are usually done using the *Spring TestContext Framework* (see section 4.4.2). A spring application context, that includes units under test, is created and later used as a base for the test methods. The scope of the application context should be as minimal as possible to alleviate the side effects from non-related components.

8.7.3 Full integration tests

Sometimes it is desirable to test the component behaviour inside the complete system. The full integration tests are just like the component integration tests, only with application context involving all components used in the application, including the database.

In order to the database operations are fast and to be independent on the started database server, the *H2* in memory database was used in the example application.

In the implementation, full integration tests are subclassed from the base class called `AbstractBusinessTest`. The class has got attached Spring application context that all subclasses share if they are run at the same time, leading to much faster test execution.

8.8 Evaluation

The web hosting management application presented as an example application for the platform is only a *functional prototype* that shows that the platform and its architecture is viable to be used for *Rich Internet Applications* development. The real enterprise solution would be much more complicated, with more entities and complex business rules, integration with on-line banking system, integration with accounting system etc. There are vast possibilities of improvement and further extension.

However, that is not the aim of the example application. The goal was to show the possibilities and capabilities of the implemented RIA platform without much features that could be distracting.

Chapter 9

Conclusion

The goal of this thesis was to design and develop a platform for Rich Internet Applications using the recent technologies, frameworks and libraries from the Java architecture stack.

The Spring framework was found as the most suitable foundation for the architecture of the platform. Given the flexibility of the inversion of control (IoC) pattern that is heavily employed by the Spring container, almost any other library can be plugged-into it without much effort.

The libraries helping to create the platform were chosen from many freely available options. In each category like data persistence, security or testing several libraries were tried as candidates and the best-fitting one selected for the final solution. The process of selection was based on several criteria from the activity in the project development, support or quality of the documentation to creating a functional prototype and/or personal experience.

The platform was designed with the knowledge of all the selected technologies, containers, libraries, persistence solutions etc. The architecture of the platform was created with patterns of enterprise application development in mind and the final version was composed of several layers with clear separation of concerns.

The layers (web/presentation, service/business and data access) have clearly defined not only interfaces, but also the objects that are used to communicate across them. The split of entity objects and data transfer objects (records) is one of the important aspects of the design.

Another important aspect was to avoid excessive code repetition needed to develop an application on top of the architecture. Using the carefully designed components, such as entity metadata registry or input and output translators, the developer is required to write less code, but retains the flexibility to override almost any step of the behaviour.

To demonstrate that the design is sound and can successfully be used to develop a rich internet application, the example application, an information system for the web hosting company, was created as a demonstration of the capabilities.

9.1 Real-world use

When the platform and the example application were finished, it was presented to the management of the *RAYNET, s. r. o.* The ideas included in the design were accepted and soon after, the architecture (although heavily extended) started to serve as a foundation for the first commercial information system. Nowadays, the development team of approximately

twenty people develop it (and with it) on a daily basis.

9.2 Possibilities of improvement

There are many ways how the platform can be improved, both from the developers perspective and the user experience. The area, in which the work has the reserves. is the production deployment. The presented one is sufficient for smaller deployment, but if the Rich Internet Application has to serve hundreds or thousands of concurrent users, the approach to the production environment has to be completely changed, possibly towards the deployment to the cluster of nodes.

Bibliography

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[2] Rudolf Pecinovský. *Návrhové vzory (in Czech)*. Computer Press, 2007.

[3] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[4] Rod Johnson. *Expert One-on-One J2EE Design & Development*. Wrox Press Ltd., Birmingham, UK, 2002.

[5] Robert C. Martin. *Clean Code: A handbook of agile software craftsmanship*. Prentice Hall, 2009.

[6] Jesse James Garrett. *Ajax: A New Approach to Web Applications*. http://www.adaptivepath.com/ideas/essays/archives/000385.php. [Online, available 2010/10].

[7] Web pages. *XML HttpRequest specification*. http://www.w3.org/TR/XMLHttpRequest/. [Online, available 2010/12].

[8] Web pages. *The Java EE 5 Tutorial*. http://download.oracle.com/javaee/5/tutorial/doc/. [Online, available 2010/09].

[9] Damodar Chetty. *Tomcat 6 Developer's Guide*. Packt Publishing, 2009.

[10] Web pages. *Spring Framework Reference Documentation*. http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/. [Online, available 2010/09].

[11] Craig Walls and Ryan Breidenbach. *Spring in action*. Manning Publications Co., Greenwich, CT, USA, 2007.

[12] Martin Fowler. *Plain Old Java Object*. http://www.martinfowler.com/bliki/POJO.html. [Online, available 2010/10].

[13] Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. http://www.martinfowler.com/articles/injection.html. [Online, available 2010/12].

[14] Tomáš Hruška. *Studijní opora předmětu Informační systémy (in Czech)*. FIT VUT v Brně, 2006.

[15] Web pages. *The Java Database Connectivity (JDBC) API.* http://download.oracle.com/javase/6/docs/technotes/guides/jdbc/. [Online, available 2010/12].

[16] Craig Russell. *JSR 12: Java Data Objects (JDO) Specification*, April 2002.

[17] Craig Russell. *JSR 243: Java Data Objects 2.0 - An Extension to the JDO specification*, May 2006.

[18] Linda DeMichiel and Michael Keith. *JSR 220: Enterprise JavaBeans 3.0*, May 2006.

[19] Linda DeMichiel. *JSR 317: Java Persistence 2.0*, December 2009.

[20] Mike Keith and Merrick Schincariol. *Pro JPA 2: Mastering the Java Persistence API.* Apress, Berkely, CA, USA, 2009.

[21] Web pages. *Spring Security Reference Documentation.* http://static.springsource.org/spring-security/site/docs/3.0.x/reference/springsecurity.html. [Online, available 2010/09].

[22] Peter Mularien. *Spring Security 3.* Packt Publishing, 2010.

[23] Ceki Gülcü. *Taxonomy of class loader problems encountered when using Jakarta Commons Logging.* http://articles.qos.ch/classloader.html. [Online, available 2010/12].

[24] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley, 2010.

[25] Tim O'Brien, John Casey, Brian Fox, Jason Van Zyl, Manfred Moser, Eric Redmond, and Larry Shatzer. *Maven: The Complete Reference.* http://www.sonatype.com/books/mvnref-book/reference/public-book.html. [Online, available 2010/12].

[26] Steve Loughran and Erik Hatcher. *Ant in action: java development with ant, second edition.* Manning Publications Co., Greenwich, CT, USA, 2007.

[27] Web pages. *Java Architecture for XML Binding (JAXB).* http://www.oracle.com/technetwork/articles/javase/index-140168.html. [Online, available 2011/03].

[28] Web pages. *Learn About the Ext JavaScript Library.* http://www.sencha.com/learn/Learn_About_the_Ext_JavaScript_Library. [Online, available 2011/04].

[29] Web pages. *xUnit Patterns: Mocks, Fakes, Stubs and Dummies.* http://xunitpatterns.com/Mocks,%20Fakes,%20Stubs%20and%20Dummies.html. [Online, available 2011/03].

Appendices

Appendix A

Contents of the enclosed CD

The CD enclosed inside this thesis contains the following files and directories:

- File `thesis.pdf` – term project report.
- File `thesis-print.pdf` – printable version of the term project report (without the coloured links).
- File `cover.pdf` – cover of the printed thesis.
- Directory `thesis` – source code of the term project report (LaTeX).
- File `app/application-bin.tar.gz` – compiled version of the platform and the example application packed together with libraries.
- File `app/application-src.tar.gz` – source code of the platform and the example application.
- File `doc/core/index.html` – Doxygen documentation of the core library.
- File `doc/refman-core.pdf` – Doxygen documentation of the core library (PDF format).
- File `doc/app/index.html` – Doxygen documentation of the entire application.
- File `doc/refman-app.pdf` – Doxygen documentation of the entire application (PDF format).

Appendix B

Running the application

B.1 Running the compiled version

The easiest way, how to quickly run the application, is to use the compiled version. The version is packed together with all required libraries into the archive file named `app/application-bin.tar.gz` that is available on the enclosed CD. The only runtime requirement is the *Java Runtime Environment* (JRE).

1. Untar the archive into any directory.
2. Change current directory to the new directory.
3. Run: `sh runner.sh`.

The application should start the boot and intialisation process. In the end, if the start is without an error, you should see the line:

```
1 INFO  [o.a.c.h.Http11Protocol] - Starting Coyote HTTP/1.1 on http-8080
```

Now you can navigate your browser to the URL http://localhost:8080 and log in as user *admin* with password *admin*.

The compiled application was successfully tested with *Sun JRE 1.6.0_24* under Linux. Running the binary application under Windows operating system was not tested, although it should be runnable, if the `runner.sh` script is rewritten into the windows batch file or unix-like environment like Cygwin is installed. However, building and running from the source code *is* supported on the Windows platform.

B.2 Building from source

Building the application from the source code is achieved using the Gradle build system. The archive `app/application-src.tar.gz` containing source code is location on the enclosed CD.

After unpacking the archive, there is a script called `gradlew` (or `gradlew.bat` in the case of Windows) which serves as a wrapper to the Gradle build system. By running it the build process starts: dependencies are downloaded and source code is compiled. The development server can be run by issuing the command:

```
./gradlew runServer
```

The command will run the server in the same way like in the case of the binary version. It is also possible to generate project files for the Eclipse IDE with the command:

```
./gradlew cleanEclipse eclipse
```

B.3 Running the test suite

Test suite is also executed through the Gradle build system, only the command is different:

```
./gradlew test
```

The test target will compile and run all JUnit tests in the source directories.