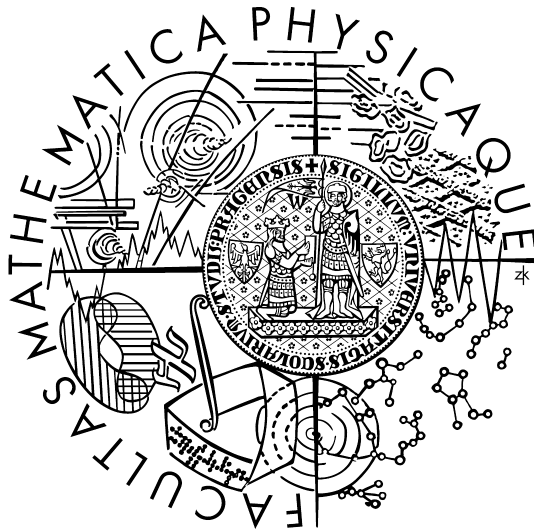
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Tibor Blénessy

Support for Enterprise Applications in SOFA 2

Department of Distributed and Dependable Systems

Supervisor: RNDr. Tomáš Bureš, Ph.D.
Study programme: Informatics
Specialization: Software Systems

Prague 2011

I would like to thank Zuzka for her help and support during writing of this thesis.

I hereby declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

Prague, 5th of August 2011 Tibor Blénessy

Název práce: Support for Enterprise Applications in SOFA 2

Autor: Tibor Blénessy

Katedra: Katedra distribuovaných a spolehlivých systémù

Vedoucí diplomové práce: RNDr. Tomáš Bureš, Ph.D.

Abstrakt:

Na zefektívnenie vývoja a na zvýšenie interoperability enterprise systémov sa vyvinuly štandardy definijúce rôzne aspekty týchto systémov. Pre systémy postavené na platforme Java sú tieto štandardy združené pod Java Enterprise Edition.

Komponentový systém SOFA 2 ponúka dobrý základ na vývoj rozsiahlych komponentovo orientovaných systémov. Cieľom tejto práce je navrhnúť a experimentálne overiť možnosť integrácie existujúcich štandardov pre enterprise aplikácie v SOFA 2.

Navrhnuté riešenie rozširuje možnosti SOFA 2 o tvorbu komponent pre webové rozhranie na základe Java Servlet API a o komponenty na ukladanie dát do relačných databázových systémov pomocou štandardu Java Persistence API. Ďalej navrhnuté riešenie integruje kľúčové technológie z platformy Java Enterprise Edition, čo uľahčí budúcu integráciu ďalších štandardov.

Klíčová slova: enterprise aplikace, komponentový systém, SOFA 2, Java EE

Title: Support for Enterprise Applications in SOFA 2

Author: Tibor Blénessy

Department: Department of Distributed and Dependable Systems

Supervisor of the master thesis: RNDr. Tomáš Bureš, Ph.D.

Abstract:

Industry standards defining various aspects of enterprise systems were developed in order to make development of such systems faster and to increase interoperability. For systems created with Java platform, these standards are contained under Java Enterprise Edition.

SOFA 2 component system provides a solid base for development of extensive component oriented systems. Goal of this work is to propose and empirically verify the possibility of integration of existing industry standards for enterprise applications in SOFA 2.

Proposed solution extends SOFA 2 with components for web user interfaces based on Java Servlet API standard and with components for persisting data into relational databases based on Java Peristence API. In addition, it integrates core technologies from the Java Enterprise Edition platform, which should make integration of further standards easier.

Keywords: enteprise applications, component system, SOFA 2, Java EE

Contents

1 Introduction **3**

1.1 Goal of the thesis 5

1.2 Structure of the text 5

2 Background **6**

2.1 SOFA 2 6

2.2 Java Enterprise Edition 12

2.3 Java Classloaders 18

3 Enterprise Applications in SOFA 2 **21**

3.1 Architecture of SOFA Enterprise Application 21

3.2 Analysis 22

4 Implementation of SOFA EE **25**

4.1 Integration of Servlet Container 25

4.2 SOFA EE bootstrap 26

4.3 Components with web interface 26

4.4 Components with data persistence 29

5 Sample Applications **33**

5.1 Numberguess 33

5.2 Accounts 36

6 Conclusion and future work **41**

6.1 Future work 42

Content of the CD **44**

References **45**

Chapter 1

Introduction

Robustness, scalability and performance are key requirements of enterprise applications. High demands are also put on development of these systems – to be effective, cheap and to prevent errors. Enterprise systems are often very extensive and have to be maintained for a long period of time.

Software standards defining various aspects of enterprise systems were developed in order to make development of such systems easier, faster and to increase interoperability. These standards define many areas of software design like network communication protocols (HTTP, FTP), database storage (SQL), user interface design (HTML, XUL, XAML), authentication (LDAP), and many others.

Component based development

In order to cope with the complexity, enterprise systems are decomposed into smaller, simpler parts called components. This approach of building software systems is referred to as a component based development (CBD). A *Component* can be viewed as some black-box entity, with its inner workings hidden from other components. A component provides a clearly defined communication interface for other components. A *Component model* is a set of rules defining the component's creation, life-cycle and other features. An implementation of a component model is called a *component system* [1].

There are many component systems used both in industry and academia. While industry component systems (like EJB [2] or Spring [3]) provide a simpler component model and do not offer advanced features, they are well tested, have a good tool and runtime support and are broadly used for the implementation of enterprise applications. They also provide excellent support for various software standards as standards are often derived from implementations that are already established.

On the other hand, component systems created in academia (like SOFA [4]

or Fractal [5]) have many advanced features, for example micro-architecture, multiple communication styles, behavior description, etc. However, they lack tooling and they do not offer stable and reliable runtime environment [1]. Usually, they also have poor support for common industry standards.

Java Enterprise Edition

Java proved itself as a stable and reliable platform for development of enterprise applications. Features like platform independence, backward compatibility and a very good tooling support made it a popular choice for development of large scale enterprise systems.

Java has a rich library and provides support for many industry standards. Support for standards is developed under Java Specification Requests (JSRs). JSR is a community driven effort to create specifications for Java platform.

Specifications for enterprise applications are aggregated under the Java Enterprise Edition (Java EE). Java EE is a set of interfaces (APIs) for numerous standards. Java EE however does not provide implementation of these standards, implementations are provided by many software vendors. This creates competition between different implementations, but allows stability of the platform. Application code can leverage API provided by Java EE and when the need arises it can exchange only the implementation of the API.

Multilayer architecture

As we already mentioned, large systems are decomposed into simpler parts. Multilayer architecture represents coarse-grained decomposition of the system into individual layers (sometimes called tiers). Each layer has a distinct responsibility and often this reflects into physical infrastructure. The most common is the three-layer architecture [9]. In the three-layer architecture applications are split into a presentation layer, a business logic layer and a database layer.

The presentation layer is responsible for displaying user interface and controlling user interaction. With the rise of the Web in the last decade most of the enterprise applications provide a web based user interface. Web UI brings many benefits both for the user and the developer – no software is needed to install on the client side, the UI has a common approach for all applications, and many others.

The business logic layer, as its name suggests, is responsible for executing business rules of the application. It involves calculations based on input data, validation of the data and figuring out what data needs to be displayed by the presentation layer.

The database layer is solely responsible for the persistence of the data into

the database and for data integrity. This includes interactive queries to the database but also batch loads and similar database operations.

1.1 Goal of the thesis

The goal of the thesis is to design and implement a support for development of enterprise applications in the SOFA 2 component system. In particular, the thesis should add support for web based user interfaces and for data management and persistence.

To implement the support we will leverage already existing standards provided by Java Enterprise Edition specifications.

1.2 Structure of the text

Chapter one overviews the requirements of enterprise applications and contemporary solutions to these requirements from various aspects.

Chapter two describes in detail the SOFA 2 component system and the Java Enterprise Edition platform.

Chapter three presents an architecture and an analysis of the proposed solution for support of enterprise applications in SOFA 2.

Chapter four shows details of implementation of new SOFA EE profile.

Chapter five demonstrates two proof-of-concept applications leveraging the SOFA 2 component system and the implemented support for web user interfaces and data persistence.

Chapter six discusses achieved results, compares them with related works and proposes future enhancements.

Chapter 2

Background

In this chapter we are going to describe the SOFA 2 component system and Java Enterprise Edition platform.

2.1 SOFA 2

The SOFA 2 component system allows building software systems from reusable parts called components. SOFA 2 provides many advanced features for modelling the components such as hierarchical component model, multiple communication styles, modelling of control parts of the component, etc.

The content of this section is mostly based on SOFA User's Guide [6] and SOFA Programmer's Guide [7]. Detailed information on particular parts can be found in the guides.

Component Model

Components in SOFA 2 are defined using a component meta-model. Meta-model describes all concepts of the component and its behavior within the component system. Meta-model is implemented using the Eclipse Modelling Framework [8]. Concepts of the component model are represented as EMF classes. Actual components are then defined using SOFA ADL language based on XML. This description of the component is referred to as the *component metadata*. *Component data* then consists of actual business code of the components.

Frame

The black box description of the component is represented as a component *frame*. The frame defines view of the component for other components, i.e. its

communication interfaces. Interfaces can be split into two basic groups:

Business interfaces relate to business logic of the component and solve problems of the application domain. Business interfaces are further split into *required interfaces* and *provided interfaces*. Required interfaces, as the name suggests, represent interfaces which are needed by the component to operate. Provided interfaces are interfaces offered to other components for consumption.

Control interfaces are mostly used by runtime environment to manipulate components and request runtime information from them. Control interfaces are typically interfaces controlling a component's lifecycle (e.g. *start*, *stop*, *deploy*) or they acquire various information from the component such as memory usage, CPU consumption, etc.

Interfaces can define communication style. Usually it is a simple method invocation, however, there can also be other communication styles defined. Cardinality of the interface specifies, whether the interface participates in a single binding or in a number of bindings. The interface can be annotated with annotations. Factory annotation defines that some method returns new components implementing specified interface. SOFA 2 has also the notion of utility interfaces, which can be bound and unbound at runtime and reference of them can be freely passed between components.

Frame can be marked as being top-level, which means that the frame represents a top-level frame of application. Top-level frames do not have provided nor required interfaces.

Architecture

Architecture provides a gray box description of the component. That means it provides a more detailed view of component's inner workings. There is an n-to-n relationship between frames and architectures. Architecture can implement multiple frames and a frame can have different implementations realized with different architectures.

As we already mentioned, SOFA 2 is a hierarchical component model. Hence, a component can be composed of simpler components. Therefore there are two basic types of architectures in SOFA 2. *Composite* architectures represent components composed of other components. On the other hand *primitive* architectures represent an atomic component implemented directly, so it can be executed on a target platform.

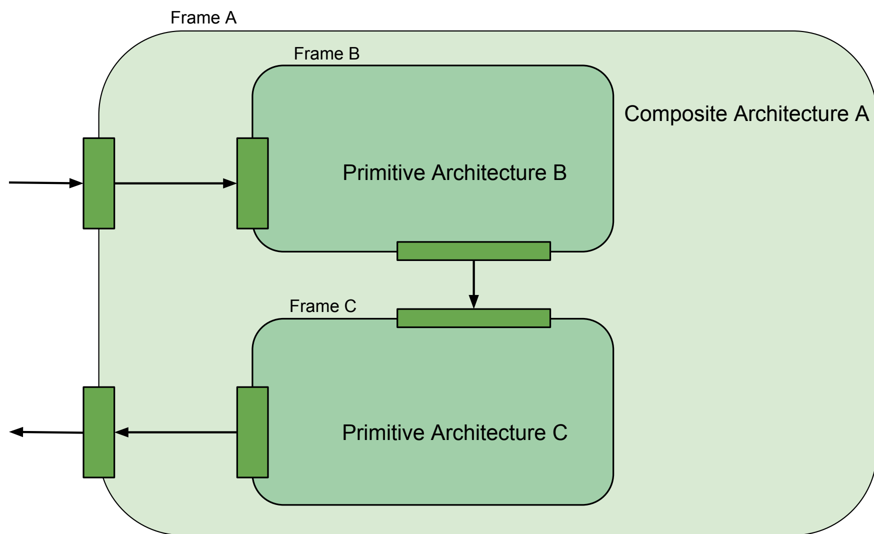


Figure 2.1: Example of a composite architecture with two primitive components.

Microcomponent model

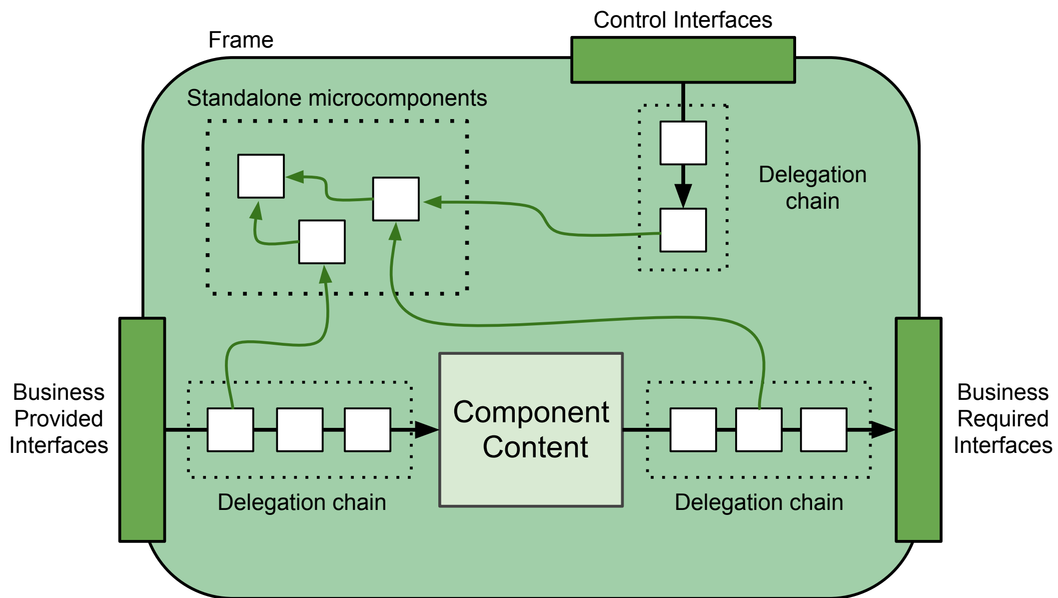
An interesting feature of the SOFA component system is the realization of control parts of the components. SOFA component model defines *microcomponents* which represent the control part of the component. To the outer world of the component this control part is available as a set of control interfaces available on the frame of the component. Internally in the component, the control part is modeled using the *microcomponent model*.

The microcomponent model is however a lot simpler than the model of the business components – it is flat (means no nesting of microcomponents is possible), it does not provide advanced features for interfaces and microcomponents do not have a control part.

The microcomponent model enables us not only to create new control interfaces of the component, but it also allows us to intercept existing interfaces – both control and business ones. Microcomponents are applied to components during component instantiation using *aspects*. Aspects contain simple queries to qualify components to which an aspect should be applied. When a component is selected with the query, the aspect modifies its control part.

This provides an elegant way how to extend capabilities of SOFA 2 component runtime and the implementation of this thesis is heavily based on this feature.

Aspects and microcomponents, which are necessary for basic component functionality are called *bootstrap*. Bootstrap is uploaded to the repository the during compilation of SOFA 2 component system.



Codebundles

Codebundle is an object representing packaged code of the application. Codebundles are created with the SOFA tools and they are uploaded into repository. Each entity with implementation (e.g. Interface, Architecture, Microinterface-type, etc.) has its codebundle. There can be also standalone codebundles bundling third-party libraries or shared application code. Shared codebundles can be referenced as dependencies.

Deployment

When a component's frames and architectures are defined they are assembled together in an *assembly descriptor*. The assembly descriptor assigns concrete architectures to frames. Based on the assembly descriptor a *deployment plan* is created. A deployment plan describes deployment of the application to the runtime environment. With the deployment plan ready, we can finally execute our application.

SOFA Runtime

SOFA 2 component system has a distributed runtime environment. This means, that application's components can be deployed to different hosts connected with a network. Connectors implementing networked communication are generated automatically and they are fully transparent to the application developer.

The whole runtime part of the SOFA 2 component system is encapsulated in *SOFAnode* or shortly node. A SOFA 2 application can run only in one node and this node has all the information for running the application. SOFAnode consists of:

Deployment Dock is the part of the SOFAnode responsible for execution of the actual application code. The dock provides interface to launch the application. When the application is launched, the dock downloads the application's code and component metadata from the Repository, reads the information and begins to instantiate components. There can be multiple docks running on different hosts in a single SOFAnode running individual components of the application.

Deployment Dock Registry keeps a record of running deployment docks within a SOFAnode. It is needed for communication between the docks. When an application is launched and its components are supposed to be running on different docks (according to the deployment plan), the dock running master architecture of the application asks the registry for a reference to docks, which should run other components of the application.

Repository is used to store a description of application's components and its business code. The repository is also responsible for versioning and team support. The repository is implemented as simple HTTP server and communication between docks and repository is based on HTTP protocol.

Global Connector Manager is responsible for connecting required and provided interfaces of components together.

Configuration management

SOFA Repository also provides configuration management facilities for SOFA 2 applications. Each entity (frame, architecture, interface, etc.) stored in the Repository is assigned a unique version identifier (represented as string hash code). Deployment docks support running of different entities concurrently and different versions of the same entity are isolated in the runtime environment.

Tooling support

SOFA 2 provides two tools for creation of applications with the SOFA 2 component system. A command line based tool named *cushion* and Eclipse plug-in *SOFA IDE*. Both tools are based on a same API and they can be used together. These tools are able to create skeletons of entities for components and communicate with the SOFA Repository and Deployment docks. With the tools one can checkout and checkin different versions of the same entity.

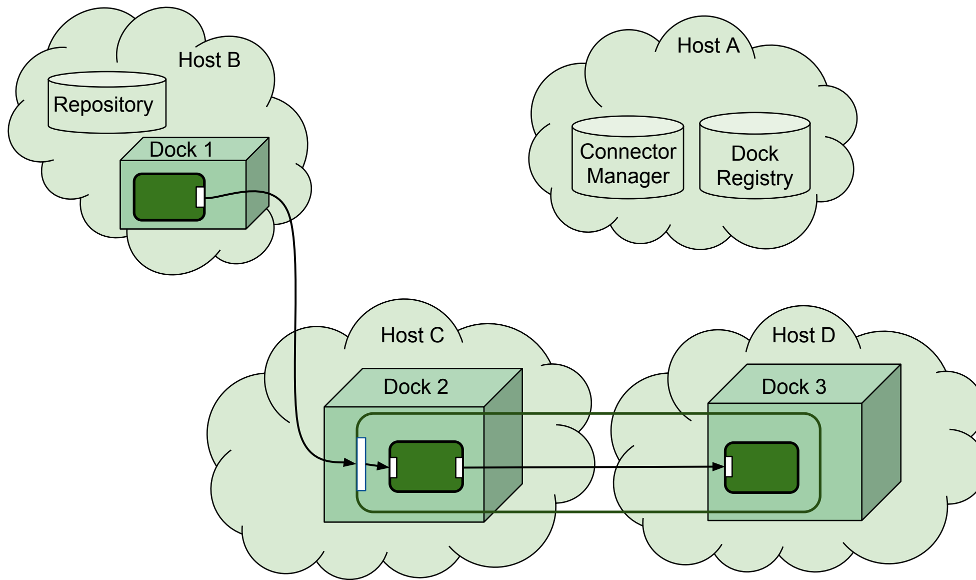


Figure 2.2: SOFA 2 runtime architecture [6] with four hosts (A,B,C,D) running parts of runtime and three deployment docks. Components can communicate across the host boundary and composite components can have its parts spanned in different docks.

SOFA Profiles

SOFA 2 component system can be split into two parts. The first part consisting of a component meta-model, ADL language (used for component metadata) and SOFA Repository is platform agnostic. It can be used to describe components for any target platform. The second part consisting mainly of Deployment Docks and related parts of the runtime is platform specific. A dock has to be able to execute business code of the component, which is specific to the targeted platform.

SOFA 2 has several options for supporting different target platforms. Support for a concrete platform is referred to as *SOFA Profile*. For the time being several profiles were developed:

SOFA J is the oldest profile. It supports executions of components written in Java language.

SOFA HI is a profile targeted at high-integrity real-time embedded systems.

SOFA J2ME is a profile based on SOFA/J, but supports development of component applications for embedded devices utilizing J2ME platform.

2.2 Java Enterprise Edition

As previously mentioned in chapter 1, Java Enterprise Edition is a collection of specifications and APIs for various industry standards. The first version of Java EE was introduced in 1999. The current version of Java EE is Java EE 6.

Java EE 6 introduces the concept of *profiles* as a way to reduce the size of the Java EE platform and focusing it on specific audiences. Profiles are configurations of the Java EE platform that are designed for specific classes of applications. A profile may include a subset of Java EE platform technologies, additional technologies that have gone through the Java Community Process, but that are not part of the Java EE platform, or both. [10]

The first profile defined by Java EE 6 is called *Web Profile*. This profile provides a subset of Java EE platform targeted for web application development.

Further in this chapter we will describe JEE standards related to web applications and data persistence.

Individual Java Specification Requests from Java EE can have multiple implementations. However, each JSR has exactly one *reference implementation*. The reference implementation serves as a complement to the specification. When the behavior according to the specification is ambiguous, the behavior of the reference implementation is taken as standard.

Application server

Runtime environment for Java EE applications is called an *application server*. The application server provides a complete implementation of Java EE specifications. It can also provide additional features not specified in JEE, but when an application relies solely on JEE APIs, it should be portable between application servers from different vendors. Table 2.1 lists some of the most popular JEE application servers.

Table 2.1: Overview of Java application servers

Application server	**Vendor**	**license type**
IBM WebSphere Application Server v8	IBM	commercial
Oracle GlassFish Server 3	Oracle	commercial
Glassfish server Open Source Edition 3		open source
Apache Geronimo 3.0		open source

```
public abstract class HttpServlet extends GenericServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp);
    protected void doPost(HttpServletRequest req, HttpServletResponse resp);
    protected void doPut(HttpServletRequest req, HttpServletResponse resp);
    protected void doDelete(HttpServletRequest req, HttpServletResponse resp);
}
```

Figure 2.3: Excerpt from Servlet API

Web applications

Support for web applications in Java EE is available since the introduction of the platform. Under the hood of any web application lies the HTTP protocol [11]. HTTP protocol is text based, request - response oriented and stateless. Java EE provides an abstraction above HTTP protocol in the form of *Java Servlet API* [12].

Java Servlet API

Java Servlet API defines a set of classes and interfaces for creating web based applications on the Java EE platform. API is contained in the `javax.servlet` package. The core of the API is a notion of *servlet* represented by the `javax.servlet.Servlet` class. Servlet is run inside a *servlet container*, which is usually an integral part of any Java based web server. Servlet can implement any request-response protocol, specialization for the HTTP protocol is implemented in `HttpServlet` class. The HttpServlet class provides abstract methods for every HTTP method (GET, POST, PUT, DELETE). An excerpt from the HttpServlet API is shown in figure 2.3.

Packaging

Java EE applications are packaged into standard archives, which can be deployed into application servers. Each archive contains, besides the actual application code, also the deployment descriptor describing the content of the archive.

The two most common types of the archive are *WAR* archive and *EAR* archive. WAR archive is the packaging of a web application. It contains resources for creating web interfaces (e.g. HTML files, CSS files, images, etc.) and actual Java code, containing the logic of the interface. It can also contain third-party libraries. The structure of the WAR archive is shown in figure 2.4.

EAR archive is used to package multiple WAR archives and other artefacts as JAR files with EJB components, third party libraries, etc.

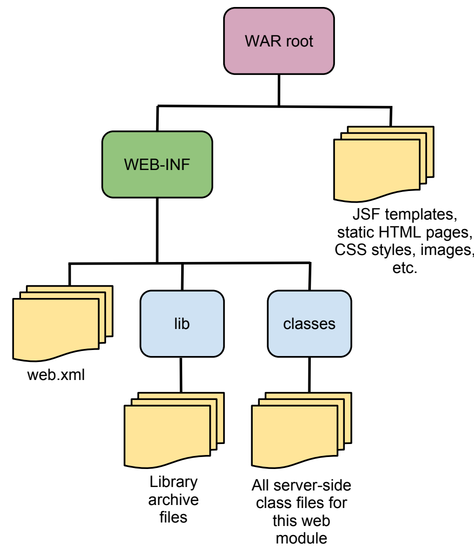


Figure 2.4: Structure of the WAR archive.

Java Server Faces

Java Server Faces framework [13] provides further abstraction above the HTTP protocol. In order to make development of complex web applications faster, JSF allow us to create web interfaces with premade reusable components. JSF also defines rigid contract of request processing and mapping of request values into applications objects.

A web application based on JSF is composed of views and beans. Views are defined as XML templates containing tags, which reference JSF components. Beans are ordinary Java classes used to provide application logic for the views.

The JSF component is used to represent part of the user interface. It can render itself in HTML code (or in other kind of presentation, rendering mechanism is pluggable), process user events, handle validation and conversion of the inputed data. Examples of components are table, text input field, menu. Components refer to application logic through the *Expression language* (EL). EL is used in XML templates to insert values and create bindings from beans containing application logic.

A big advantage of the JSF framework is its extensibility. JSF specification was designed with many extension points, therefore it is easy to create new kinds of components and to modify the behavior of the current ones.

Persistence

Almost all applications require persistence data. Relational databases became de facto standard for storing data in enterprise applications. Relational databases represent data using entities and relationships between them. Entities are mapped to tables and attributes of an entity are represented as columns. Columns in principle contain only scalar values. To express the relationship between entities and to ensure referential integrity, one uses foreign keys, which reference another table's primary key.

JDBC API

SQL language is a standard way to access data from RDBMS system and Java provides Java Database Connectivity API (JDBC) to query database using SQL.

However, this raw approach to access data is very verbose (in lines of code) and error prone. SQL queries are expressed as string literals, therefore there is no way for the compiler to validate them. Also, it is very difficult to refactor the code when the application code or the database structure changes.

Enterprise applications are build with object oriented paradigm and algorithms in enterprise applications are expressed with objects and methods. Objects can contain not only scalar values, but also references to other objects and collections of objects. Objects representing the business logic of the application are usually referred to as a *domain model*.

To cross this gap between the object oriented approach and the relational database approach, one has to convert the data fetched from the database into the object graph of the domain model.

To circumvent these deficiencies, Object-to-Relational (ORM) techniques were developed. ORM solutions provide API to store and retrieve objects of the domain model from RDBMS in an automated way.

Java Persistence API

Java Persistence API [15] is a Java Enterprise Edition standard for Object-to-Relational mapping. Using JPA brings many advantages over the raw JDBC approach. With JPA we can store a whole object graph into the database with a single method call. JPA also provides its own query language called JPQL, which is more object oriented than SQL and JPA runtime can convert and fine tune JPQL queries into a database specific version of SQL by using dialects. But JPQL is still written in Java Strings, so it is vulnerable to refactoring problems. JPA also provides Criteria Queries, which are build using metamodel and therefore they can be statically analyzed by the compiler.

The main interface of the API is the *EntityManager*. EntityManager is used to retrieve and modify data in the database. It is also used to create custom database queries either using JPQL, SQL or Criteria Queries.

Mapping of the domain model objects and relational database tables can be defined by XML files or by annotations on classes. Basic JPA mapping annotations are shown in table 2.2.

Table 2.2: Some of Java Persistence API annotations

Annotation	**Meaning**
@Entity	Class to be stored in the database
@Table	Configuration of table to which entity should be stored
@Id	Primary key of the entity. Id can be configured to be automatically generated and generation mechanism can be customized
@Column	Field or property should be mapped to the database. Name of the column can be configured
@OneToMany	define relationship between entities

Object-to-relational mapping and Java Persistence API are relatively complex topics. Further details can be found in [15].

Context and Dependency Injection

Dependency Injection is a design pattern for improving the structure of the application code. The main idea of the pattern is, that a component should not request its dependencies explicitly, but instead only declare required dependencies. Dependencies are then *injected* into the component automatically, when the component is created.

Context of a component means, that the component is attached to the particular *scope* and its lifecycle is tight with its scope. Therefore, if the scope of the component is created and the object is requested for the first time within the new scope, a new instance of the object is provided. If the scope is destroyed, all objects within the scope are destroyed. Some examples of scopes are:

Singleton scope is created with the application and exists through the lifetime of the application. Therefore, there can be only one instance of an object within the application.

Session scope is tied to the session. Session is usually related to the user working with the application. So when the object is tied to the session scope, there exists an instance of an object for each user.

Context and Dependency Injection is Java EE specification (JSR 299) defining set of services implementing lifecycle contexts and dependency injection for

Java EE applications. CDI is a new standard introduced in Java EE 6, however, dependency injection and notion of object scopes was pioneered long ago with Spring [3] and other frameworks.

According to [16], the JSR 299 specification (CDI) defines a set of complementary services that help improve the structure of the application code. CDI layers an enhanced lifecycle and interaction model over existing Java component types, including managed beans and Enterprise Java Beans. The CDI services provide:

- an improved lifecycle for stateful objects, bound to well-defined contexts
- a typesafe approach to dependency injection,
- object interaction via an event notification facility
- a better approach to binding interceptors to objects, along with a new kind of interceptor, called a decorator, that is more appropriate for solving business problems
- and an SPI for developing portable extensions to the container. The CDI services are a core aspect of the Java EE platform and include full support for Java EE modularity and the Java EE component architecture.

Despite being a relatively new standard, CDI is quickly gaining a broad support within other Java EE specifications. It provides important glue code to connect various parts of the Java EE world.

A component managed by CDI is called a *bean*. The main entry point to the CDI API is the BeanManager class. Programming model of the CDI is based on annotations. An overview of basic CDI annotations is provided in table 2.3.

Table 2.3: Overview of CDI annotations

Annotation	**Meaning**
@Named	Component is managed by CDI and it is identified by name. If no name is provided as annotation attribute, default name is derived from class name of the bean
@SessionScoped	Annotation defining scope of the component
@Inject	Marks a dependency to be injected
@PostContruct	Method to be invoked, when component is constructed
@Produces	Factory of CDI managed beans.

CDI is well designed with regards to its extensibility. There are SPI interfaces to extend various functionalities of the framework.

2.3 Java Classloaders

Classloading is a mechanism for loading class definitions into the Java Virtual Machine. Class definitions are stored in binary class file format. When the class is requested for the first time in an application's code, it is loaded into memory using a *classloader* object. Usually, the classloader loads the class from a file on filesystem, but there can be other ways to construct the class.

Class definition is known only to the instance of the classloader which loaded the class. Therefore, if there are two instances of classloaders and they happen to load the same class file, for the JVM, loaded classes appear as two distinct types.

We can say, that class type is defined by a class name and the instance of the classloader which loaded the class.

A delegation is used to share types between different instances of classloaders. Each classloader has a parent classloader to which requests for classes unknown to the classloader are delegated – classloaders create a hierarchy. The first three classloaders in the hierarchy are:

Bootstrap classloader is the first classloader created by the JVM machine. It is used to load all `java.*` packages and main class of executed application

Extension classloader is used to load classes from the JVM extension directory

System classloader is used to load classes from the system classpath.

Custom classloaders usually delegate requests to the system classloader.

Classloading in Java EE

Classloading hierarchy is heavily utilized in Java application servers. This has two main reasons – to provide security and isolation for applications. Security means, that different applications deployed to the application server can not interfere with each others classes. Isolation means, that each application can have its own version of the libraries and the types from the libraries are not shared among applications. Therefore, applications can use the same library (with the same class names) in different versions.

Usually, there is a complex hierarchy of classloaders in the application server. Example of such a hierarchy is shown in figure 2.5.

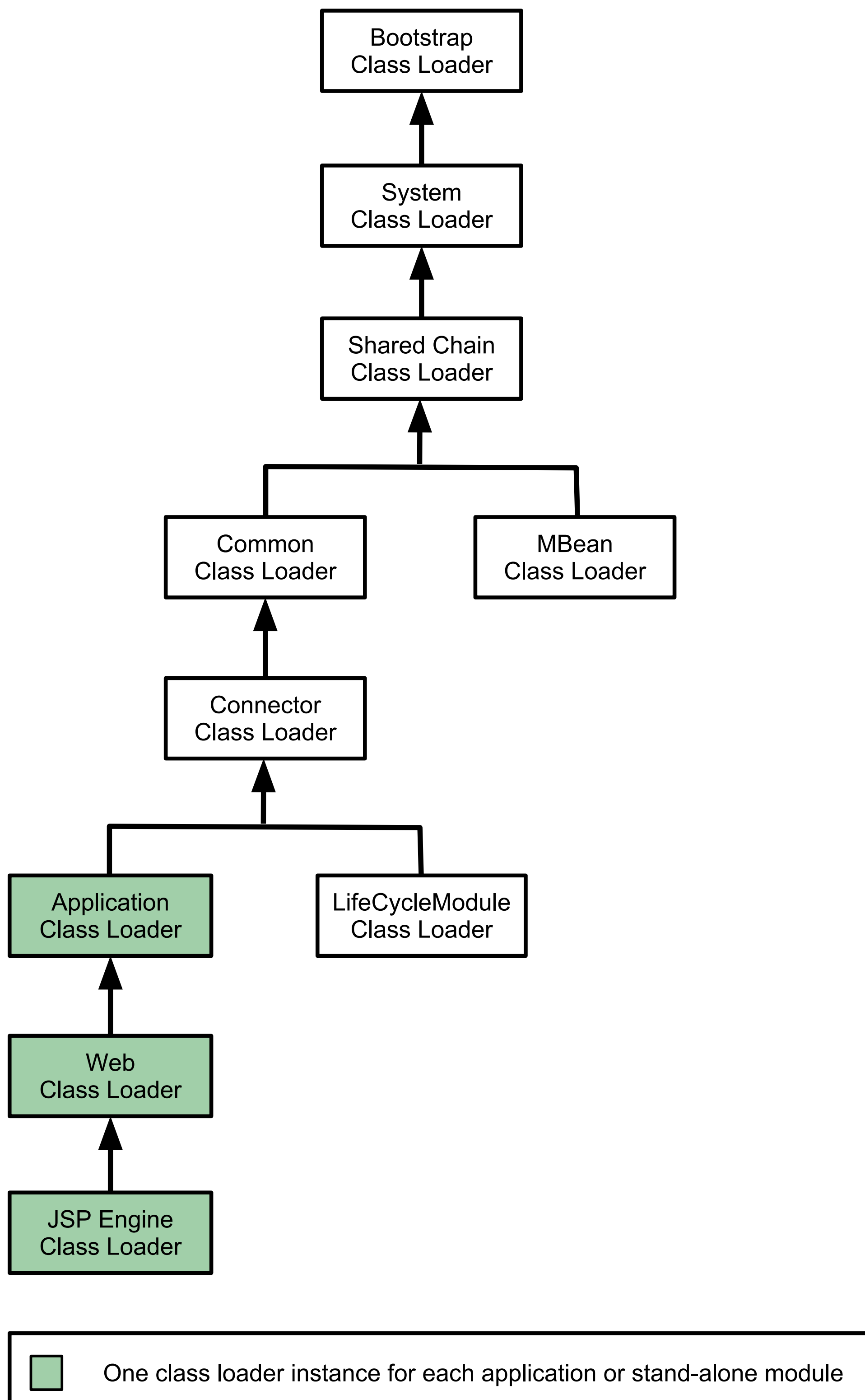


Figure 2.5: Class loader runtime hierarchy in Sun Java System Application Server. [17]

Classloading in SOFA 2

SOFA 2 takes a different approach to provide isolation for applications, called name mangling. As we already mentioned, every revision of an entity in the SOFA 2 model is assigned a unique version number when it is stored in the repository. This version number is used to modify actual class file in which the primitive architecture is implemented. Also, all dependencies of the entity are modified in order to use a concrete version of the class.

There is a specialized classloader in the SOFA 2 system, which is able to load class definitions at runtime from the repository. To allow sharing of entities between applications, the classloader is implemented with singleton pattern. So there is always only one instance of the classloader and each class was loaded using this instance. Therefore, specific version of the class is available to all applications, but multiple revisions of the same class do not interfere.

Chapter 3

Enterprise Applications in SOFA 2

As mentioned in section 1.1 the goal of this thesis is to implement support for enterprise applications in SOFA 2 while leveraging contemporary Java EE standards. In this chapter we are going to design enterprise application support and analyze necessary changes to the SOFA 2 component system.

3.1 Architecture of SOFA Enterprise Application

First we have to design the overall architecture of enterprise application and how this architecture will be realized with the SOFA 2 component system. As we already described in the introduction chapter, a common structure for enterprise applications is the three-layered architecture. We will also adopt this architecture.

Presentation layer of the application will be realized by a component marked with *WarArchive* annotation. This component is able to provide a web based user interface, which is made accessible via HTTP protocol to the user's web browser.

Business logic layer of the application can be implemented using normal SOFA 2 components as it does not have any specific needs and the current infrastructure provided by the SOFA 2 component model is fully sufficient.

Database layer of the application needs to be able to store application data to the RDBM system. SOFA EE runtime will provide this functionality to the components marked with *PersistenceUnit* annotation.

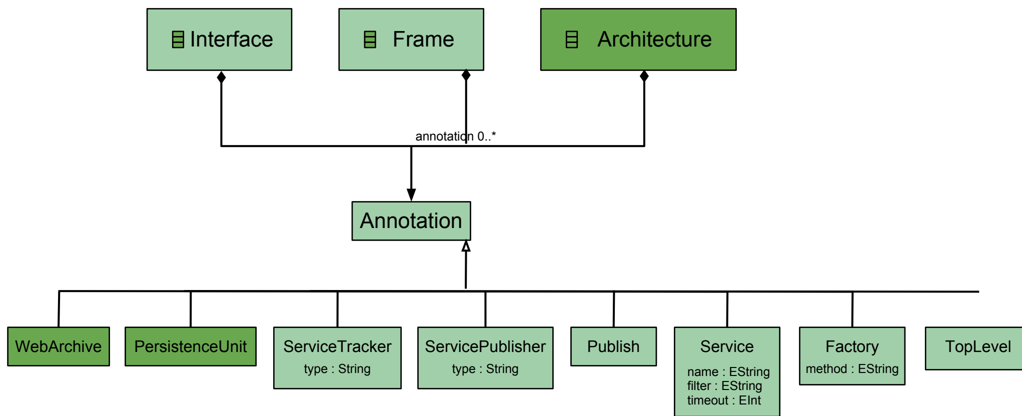


Figure 3.1: Annotations in SOFA 2 component meta-model. Modifications of the meta-model are highlighted with darker color.

3.2 Analysis

To provide support for a web based application we have to modify runtime environment of the SOFA 2 component system. Since applications for SOFA EE are still based on Java platform, we will not provide new implementation of the deployment dock, but only modify current SOFA/J profile. Some modification to the component meta-model would also be necessary.

Extending component meta-model

Currently SOFA 2 only supports annotations for Frames and Interfaces. To differentiate new kinds of architectures for presentation layer and database layer components we will add support for annotations on Architectures. We will also introduce two new annotations – *War Archive* annotation and *PersistenceUnit* annotation. Support for these annotations will also be necessary in ADL language in order to be able to use them.

A diagram of the modified component model is shown in figure 3.1.

Classloading

As we previously stated in 2.3 Java EE and SOFA 2 employ different philosophy with regards to classloading. The advantage of SOFA 2 approach is, that it is more portable, because the classloading mechanism is restricted on some Java platforms. On the other hand, the renaming mechanism is not perfect. When the class is accessed using Java reflection, which means that its name is represented as String literal, the renaming mechanism can not be applied and the code will fail during runtime.

Unfortunately, Java EE platform is heavily based on reflection, so SOFA 2 classloader will have to be modified.

Another shortcoming of the SOFA 2 classloader is that it does not offer API for classpath scanning, more specifically, it does not implement API for resource access using `java.net.URL` class. This API is required by implementations of Java EE specifications, so we will have to implement it.

Web applications tend to use many additional third party libraries to ease creation of web interfaces (for example JSF component libraries, etc.). These resources are usually bundled together within an application archive. Repackaging of these libraries into the SOFA codebundle is laborious and impractical. Also, it will pollute the repository with codebundles related only to one component of the application. We will have to modify the SOFA 2 classloader model to employ another classloader to load classes from these third party libraries packaged in WAR archives.

Packaging

Web applications require a lot of other resources beside Java classes. The SOFA 2 tools API supports processing of Java code only, however support for other types of architecture content can be added using `CodeProcessor` SPI interface. We will need to create a new codeprocessor for SOFA EE applications.

Web server

In order to provide support for the presentation layer we have to employ a web server. The web server should be a facility provided by the deployment dock to the components of the presentation layer. The most suitable way to implement this facility is to extend the deployment dock with a Servlet container.

Java EE components

Context and Dependency Injection is an emerging standard in Java EE platform. Many contemporary frameworks for web based applications like Java Server Faces, Wicket, etc., provide support for CDI. In order to utilize this support, inner workings of the presentation layer component will be implemented using CDI.

For the presentation component to be able to communicate with other SOFA 2 components (i.e. components from business layer), there has to be a way to inject its required interfaces into the CDI world. Since the only task of the presentation component is to interface with the user, it does not have any business provided interfaces.

Persistence

Database layer components need to be able to store the application data into the database. Application data are represented by Java classes of domain model. Mapping between Java classes and database tables is defined by JPA annotations. SOFA EE dock will provide JPA API to store annotated classes in the database with a configurable database connection.

Chapter 4

Implementation of SOFA EE

In the previous chapter we have analyzed what changes have to be introduced in order to provide support for web interfaces and database persistence. In this chapter we will show how these changes were implemented to create the SOFA EE profile.

4.1 Integration of Servlet Container

In order to provide a web server facility we will embed a servlet container to the deployment dock. There are several open source implementations of Servlet Containers. We have chosen the Jetty Servlet Container [18], because it is easily configurable, has a small footprint and it is already used within the SOFA 2 system for implementation of SOFA Repository. Integration of the servlet container is implemented with *dock capabilities*.

Dock capabilities

The Deployment dock can be extended by dock capabilities. The dock is launched with a main method in the `RunDeploymentDock` class, which in turn reads the system property `sofa.dock.capabilities` containing a colon deliminated string. This string is split into an array and passed to the `DeploymentDockFactory`. Subsequently the array is scanned for well-known constants representing some dock capability and the desired capability is initialized.

We have created a new capability with constant `JEE`. Unfortunately, scanning mechanism of the string array with capability constants was not extensible. We have introduced the Service Provider Interface `DockCapabilityHandler` to make capability initialization mechanism pluggable. `DockCapabilityHandler` interface is shown in figure 4.1.

Implementations of dock capability handlers are automatically discovered on the Dock's classpath with SPI mechanism. Afterwards each implementation is

```
public interface DockCapabilityHandler {

    boolean handles(String capability);

    void handleDockCapability(String capability) throws SOFAException;
}
```

Figure 4.1: Interface for extension of the Deployment Dock with capabilities.

asked by the `handles` method, whether it wants to handle the capability and if so, the `handleDockCapability` method is called.

There are two implementations of the `DockCapabilityHandler` interface. One is the refactoring of the old code for support of the OSGi technology and the other one, provided in the `JEEDockCapabilityHandler` class, handles the new `JEE` capability.

Jetty wrapper

When the `JEEDockCapabilityHandler` is invoked, it creates a `JettyWrapper` instance. The `JettyWrapper` is a singleton representing the Jetty Servlet Container. After creation, Jetty is initialized and launched. The reference to the running Jetty Server is available with public static method.

4.2 SOFA EE bootstrap

As we mentioned in section 2.1, bootstrap is a set of SOFA 2 aspects and microcomponents, which provide basic functionality for the components. We have also created a bootstrap for the SOFA EE profile to create new control interfaces for SOFA EE components. An overview of the introduced microcomponent entities is shown in table 4.2. Usage of the bootstrap is explained in detail in following sections.

Bootstrap is uploaded to the repository during the build of SOFA EE.

4.3 Components with web interface

Now we will describe the implementation of the component with the web interface. As any other SOFA 2 component, the web interface component has also a frame and an architecture implementing this frame. It is a primitive architecture, however, it is not implemented as a Java class contained in a codebundle, but as the architecture's codebundle *is* WAR archive of a web application. In the following text we will refer to the component with a web interface as a *WarArchive* component.

Table 4.1: Overview of SOFA EE boostrap

Name	Type
Presentation layer components	
Bootstrap.JEE.WebAppAspect	aspect
Bootstrap.JEE.WebAppDeployer	control interface
Bootstrap.JEE.MWebAppDeployer	microcomponent
...jee.microarchitecture.MIWebAppDeployer	microinterface type
Database layer components	
Bootstrap.JEE.PersistenceUnitAspect	aspect
Bootstrap.JEE.Persistence	control interface
Bootstrap.JEE.MPersistence	microcomponent
...jee.microarchitecture.MIPersistence	microinterface type

Web application classloader

In order to add support for an architecture packaged as WAR archive, we have created a new `SOFAWebAppClassloader` classloader. This classloader is based on Jetty's `WebAppClassloader`, so it understands the WAR archive structure and is able to load classes from libraries in the `WEB-INF/lib` and class files in `WEB-INF/classes`. When the `SOFAWebAppClassloader` can not found the requested class, the call is delegated to the `SOFAClassloader` singleton instance.

Integration with CDI

As mentioned in analysis section (3.2), to leverage contemporary frameworks for creating web based user interfaces, we have integrated support for Context and Dependency Injection specification (JSR 299). We have used the Weld Framework [19] to provide CDI support. Weld is a reference implementation for JSR 299 and it is often used as a CDI implementation in application servers. For example, both Glassfish and JBoss application server use Weld as a CDI provider.

CDI provides a set of SPI interfaces to create portable extensions for integration with other technologies. To integrate CDI into the SOFA 2 programming model we have created `SOFACDIExtension`.

During initialization of the WarArchive component, the CDI container is bootstrapped. CDI scans classes on web application's classpath using `SOFAWebAppClassloader` and discovers CDI beans annotated with `@Named` annotation. After initialization, `SOFACDIExtension` is summoned. `SOFACDIExtension` registers WarArchive component's required business interfaces as CDI beans, so they became available for injection into the beans of the web interface.

Component packaging

In order to be able to create a primitive architecture for the WarArchive component we have extended SOFA tools API to be able to package WAR archive and upload it to the repository.

We have created the `JEECodeProcessor`, which implements the `CodeProcessor` SPI interface and is used with `JEE` lang setting of the working copy. `JEECodeProcessor` compiles classes of the architecture into `WEB-INF/classes` directory and adds `WEB-INF/lib` classes on the compilation classpath.

Deployment

During instantiation of the WarArchive component `WebAppAspect` is applied to the component. The aspect creates the `MWebAppDeployer` microcomponent, which implements the `MIWebAppDeployer` microinterface type. This interface is then provided as the `WebAppDeployer` control interface. The interface is shown in figure 4.2.

```
public interface MIWebAppDeployer {

    void deploy() throws SOFAException;

    void undeploy() throws SOFAException;

}
```

Figure 4.2: Microinterface type of WebAppDeployer control interface

After component instantiation, when the component is starting, the deployment dock uses this interface to deploy the component into running Jetty server. The deployment dock calls `deploy` on `WebAppDeployer`, which uses `SOFAAppProvider` to create a web application.

`SOFAAppProvider` is a specialization of the Jetty `AppProvider` interface, which allows us to customize the source of applications for Jetty. Standard Jetty `AppProvider` implementation watches changes in the specified folder on the filesystem and when a new WAR archive appears, it is provided to Jetty. `MWebAppDeployer` downloads the architecture's bundle to a temporary file and through the `SOFAAppProvider` this temporary file is inserted into Jetty.

`SOFAAppProvider` extends the configuration of the web application in order to automatically insert the initialization of CDI container. The `SOFAWebAppContextConfiguratio` is used to implemented this feature.

The whole deployment process is visualized with a UML sequence diagram in figure 4.3.

Each deployed application has *context*. Context is a set of servlet container

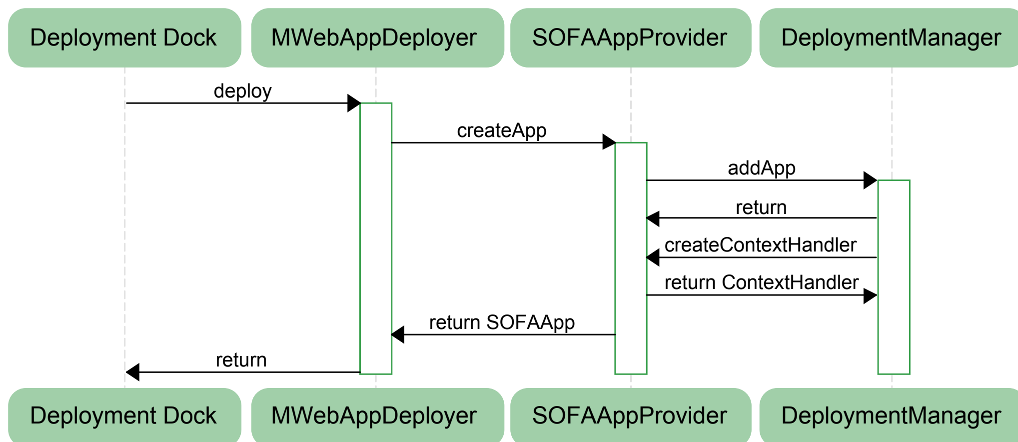


Figure 4.3: Call sequence during deploying of web interface component

resources associated with the application. It is represented with address in form of a string constant and this constant is used to create a URL address of the application in order to provide access to the application via HTTP. In order to configure, a context address architecture has to be configured by `context` parameter in the deployment plan. More elaborate ways can be utilized to create a mapping of the external URL address space to internal addresses on which web applications react, but this simple approach is sufficient for our purposes.

Undeploy process is implemented similarly, `undeploy` call is delegated to the `SOFAAppProvider`, which removes a deployed application from Jetty.

4.4 Components with data persistence

Implementation of a data persistence component follows the same pattern as the WarArchive component implementation, however, it is a bit simpler, because a component does not have to be deployed to some third party runtime (Jetty). The component uses a primitive architecture implemented by Java class, same way as a normal SOFA 2 component. Java classes of the domain model, which should be persisted, are annotated with JPA annotations. The domain model can be in the same codebundle as architecture's implementing class or they can be bundled in dependent codebundles.

We will refer to the component with a support for data persistence as *PersistenceUnit* component.

Again, during instantiation of the component, `PersistenceUnit` annotation on the architecture is recognized and the SOFA EE bootstrap aspect `PersistenceUnitAspect` is applied. The aspect creates `Persistence` control interface with `MIPersistence` microinterface type, which is implemented by `MPersistence` microcomponent.

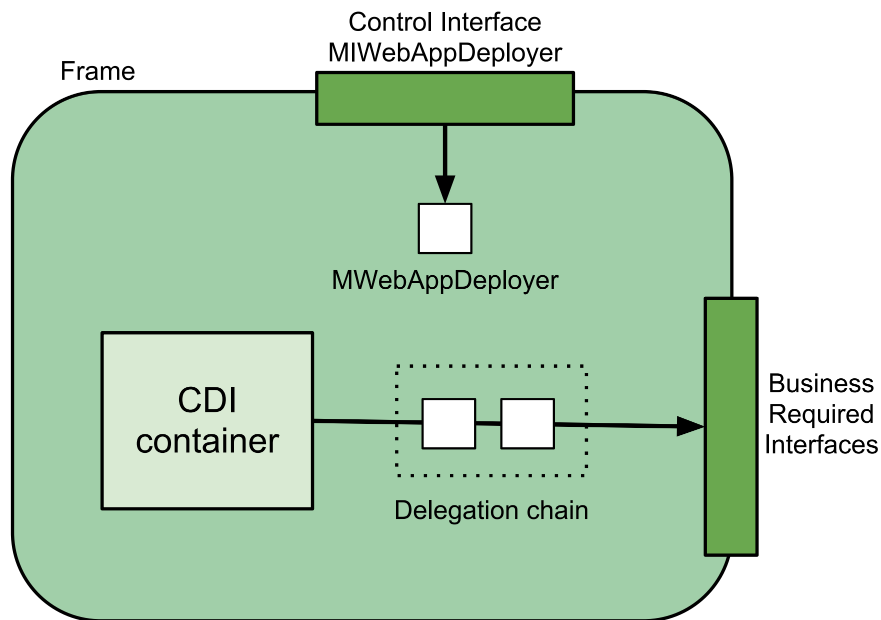


Figure 4.4: Architecture for component implementing web user interface.

```
public interface MIPersistence {
    EntityManagerFactory getEntityManagerFactory();

    EntityManager createEntityManager();
}
```

Figure 4.5: MIPersistence microinterface type

`Persistence` control interface provides access to the JPA implementation provided by the deployment dock. The interface is shown in figure 4.5.

SOFA 2 provides the `SOFAThreadHelper` for a component to access its control interfaces from business code, however, this approach is not very convenient. Therefore, we provided support for automatic injection of the `EntityManagerFactory` instance, if there is a field of this type in the implementation of the PersistenceUnit architecture.

Integrating Hibernate EntityManager

Hibernate [14] was a popular choice for object-to-relational mapping long before Java Persistence API was defined. JPA was greatly inspired by Hibernate. Hibernate EntityManager is a Java Persistence API provider build on top of a Hibernate Core. We chose Hibernate as JPA provider for SOFA 2 Persistence Unit components, because it is often used in industry and it provides useful features beyond JPA specification (e.g. automatic creation of database scheme, finer control on query generation, etc.). Further details on JPA and

Hibernate can be found in [15].

JPA implementation for SOFA 2 is wrapped in `SOFAPersistence` class. Microcomponent `MPersistence` holds instance of this class to provide JPA functionality for the component. This instance is created during component initialization.

`SOFAPersistence` loads standard `META-INF/persistence.xml` resource from the architecture's codebundle and parses configuration for database connection. Then it reads all class names contained in the codebundle and its dependencies. Classes are queried for JPA annotations. If they are annotated with any of the JPA annotations, then they are added to the configuration of JPA. The configuration is encapsulated in the `SOFAPersistenceUnit` class implementing the `PersistenceUnit` interface from JPA. This interface gives all the information necessary for the JPA provider.

In the end, the `SOFAPersistenceUnit` object is passed to the `HibernatePersistence` instance. The `EntityManagerFactory` object is instantiated and the reference is stored in the `MWebAppDeployer` microcomponent.

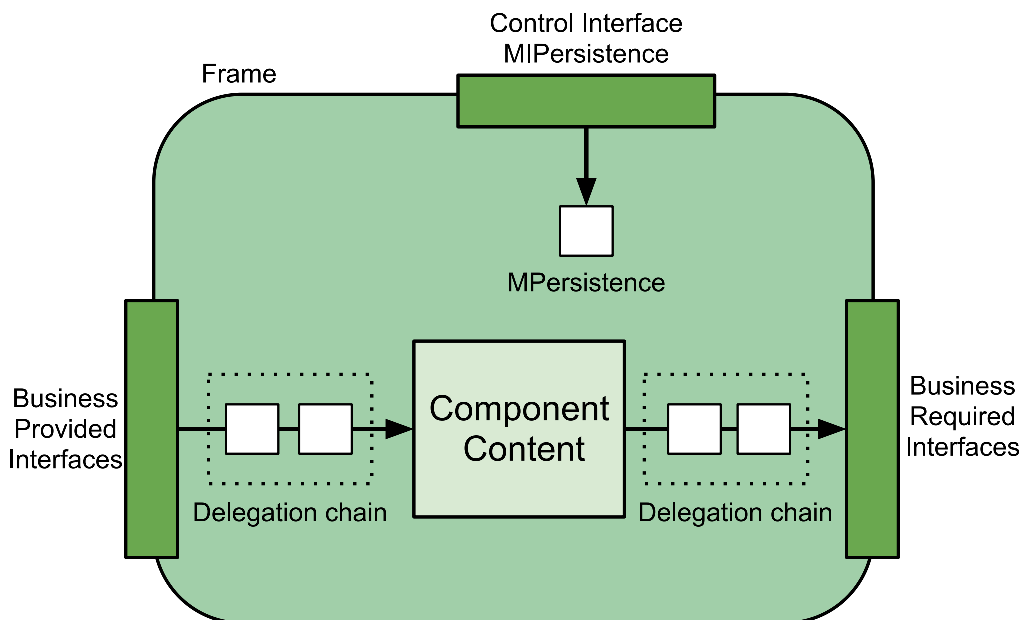


Figure 4.6: Architecture for component implementing persistence unit.

Classpath resources

As we mentioned, Java Persistence API expects to find resources on the classpath and refers to them using `java.net.URL`. However, the `SOFAClassloader` does not support this API. We had to implement support for addressing resources in codebundles by `java.net.URL`.

java.net.URL provides mechanism for accessing resources using protocols. Support for protocols is pluggable and one can provide a custom URL scheme and implementation of its protocol. We have created a new sofa protocol to access resources stored in the codebundles. The protocol is implemented by the SOFAUrlHandler and the SOFAUrlConnection. When the resource in codebundle has to be referred by URL, we create a new java.net.URL object and pass the instance of SOFAUrlHandler. The SOFAUrlHandler has a reference to the codebundle, in which the resource should be searched for. When the URL object with the SOFAUrlHandler is resolved, a call is delegated through the SOFAConnection to the SOFA classloader.

This mechanism is utilized by the SOFAPersistenceUnitInfo, because it has to provide URL for the root of the codebundle containing JPA entities.

Chapter 5

Sample Applications

5.1 Numberguess

Numberguess is a proof-of-concept application for SOFA EE. It is a web application for playing the “guess the number” game. The application is split into two components, which are connected by the `IGuess` interface. A scheme of the application is shown in figure 5.1.

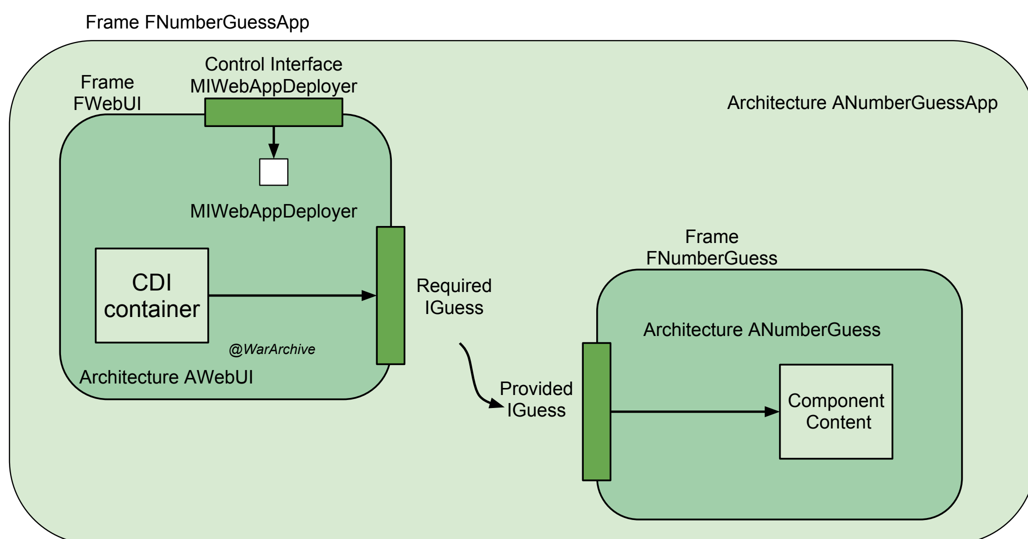


Figure 5.1: Architecture of the Numberguess application

Architecture ANumberGuessApp

`ANumberGuessApp` is a composite architecture representing an envelope for the application. Its frame is a top-level frame `FNumberGuessApp`. As a top-level frame it does not have provided nor required interfaces.

```
public interface IGuess {
    void initRandom();
    int makeGuess(int guess);
}
```

Figure 5.2: IGuess interface from the NumberGuess sample application

```
<h:form>
    Guess the number: <h:inputText value="#{numberguess.guess}" />
    <h:commandButton action="#{numberguess.makeGuess}" value="guess" />

    My number is <h:outputText value="#{numberguess.message}" />

</h:form>
```

Figure 5.3: An excerpt from the JSF view template for the NumberGuess game.

Architecture ANumberGuess

`ANumberGuess` architecture is the core of the application, it implements the logic of the game. It generates a secret random number and evaluates the player's guess. The frame of this architecture is `FNumberGuess`. It provides the `IGuess` interface for other components. The `IGuess` interface is shown in figure 5.2

Architecture AWebUI

The second architecture called `AWebUI` represents web user interface, which receives user's input and shows the result of his guess. This architecture has a `FWebUI` frame, which requires the `IGuess` interface.

`AWebUI` is build as a WAR architecture and it uses the JSF framework for creating the user interface. The WAR archive of the architecture contains two view templates. The first view accepts the player's guess and shows feedback on the guess. The code of the view is in figure 5.3. In the view we can see how CDI beans are referred to through EL expressions with `#{}`. The second view announces to the player that his guess was correct and the player is able to start the game again. Behind the views is the `NumberGuess` bean managed by CDI. This object stores user's input and calls the `ANumberGuess` architecture through the `IGuess` interface. The `IGuess` interface is injected from CDI context. The code for the NumberGuess bean is shown in figure 5.4. The ADL file of architecture is shown in figure 5.5.

```
@Named("numberguess")
@SessionScoped
public class NumberGuess implements Serializable {
    private Integer guess;
    private String message;
    @Inject
    private IGuess guessInterface;

    @PostConstruct
    public void init() {
        guessInterface.initRandom();
    }

    public String makeGuess() {
        int result = guessInterface.makeGuess(guess);
        if (result == 0) {
            return WON;
        } else if (result > 0) {
            message = BIGGER;
        } else {
            message = SMALLER;
        }
        return null;
    }
}
```

Figure 5.4: The NumberGuess CDI bean used in implementation of web interface for the NumberGuess game. Result value of makeGuess method determines JSF view to be displayed.

```
<?xml version="1.0" encoding="UTF-8"?>
<architecture name="sofaee.demo.numberguess.AWebUI"
        frame="sofatype://sofaee.demo.numberguess.FWebUI"
        war="true"
        lang="JEE" >
    <property name="context" type="prop_value" />
</architecture>
```

Figure 5.5: ADL file of architecture AWebUI in the NumberGuess application. Attribute “war” adds WarArchive annotation.

5.2 Accounts

Accounts is another application for SOFA EE. It is a web application for management of personal finances. It demonstrates how to build data-oriented enterprise applications with SOFA EE.

Similarly as in the NumberGuess application, `AAccountsApp` is a composite architecture implementing top-level frame `FAccountsApp`, which encapsulates the whole application. This architecture consists of two primitive architectures: one architecture is used for web user interface, the other one for storing data into a relational database. Overall scheme of the application is shown in figure 5.6.

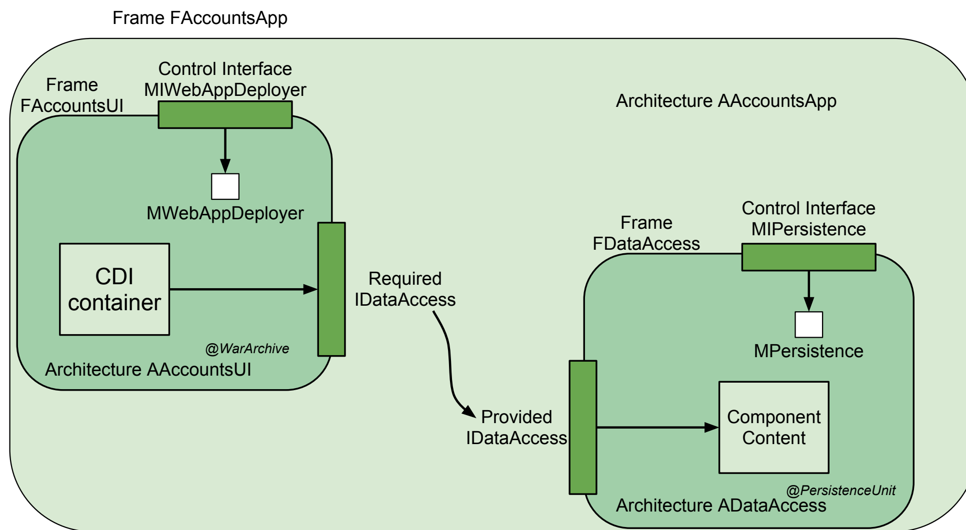


Figure 5.6: Architecture of Accounts application

Architecture AAccountsUI

Architecture `AAccountsUI` is similar to the `AWebUI` architecture in the NumberGuess application. It also uses JSF framework for a web based user interface. Its frame `FAccountsUI` has one required business interface `IDataAccess` used to access information about accounts and transactions. All this information is presented in the view backed with `Accounts` CDI bean.

Architecture ADataAccess

More interesting is the second architecture `ADataAccess`, as it shows the implementation of a PersistenceUnit component. The frame of the architecture `FDataAccess` provides business interface `IDataAccess`. The code of the

```
<?xml version="1.0"?>
<architecture name="sofaee.demo.accounts.ADataAccess"
   frame="sofatype://sofaee.demo.accounts.FDataAccess"
   impl="sofaee.demo.accounts.ADataAccess"
   lang="Java"
   persistenceUnit="true"
   >
  <dependency bundle="sofatype://sofaee.demo.accounts.CEntities" />
</architecture>
```

Figure 5.7: Architecture ADataAccess from Accounts application. Attribute persistenceUnit adds PersistenceUnit annotation.

`IDataAccess` interface is shown in figure 5.9. The ADL file of the architecture is shown in figure 5.7.

Domain model of the application is represented by two classes: `Accounts` and `Transaction`. They are in one-to-many relationship, i.e. there can be multiple transactions assigned to one account. Both classes are marked as JPA entities with `@Entity` annotation. Entities are bundled in a separate codebundle, which is marked as a dependency of `ADataAccess` architecture. The code of the entities is shown in figure 5.10.

The connection to the database is described in the `META-INF/persistence.xml` file bundled in the code bundle of the architecture. The file is shown in figure 5.8. Accounts uses embedded in-memory database called H2 [20], which is suitable for demo purposes, because it does not require a complicated setup. Initial data of the database are loaded automatically from the `META-INF/import.sql` script during the start of `ADataAccess`.

`ADataAccess` uses JPA typesafe Criteria Query to fetch all accounts from the database. SQL query is generated automatically and results of the query are mapped to the instances of the domain model. To fetch transactions for a particular account we just call the `Accounts.getTransactions()` method and JPA creates a database query in the background. Usage of JPA in ADataAccess is shown in figure 5.11.

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/
        xml/ns/persistence/persistence_2_0.xsd"
   version="2.0">
   <persistence−unit name="accounts">
      <properties>
         <property name="javax.persistence.jdbc.driver" value="org.h2.Driver" />
         <property name="javax.persistence.jdbc.url" value="jdbc:h2:tcp://localhost/
              mem:accounts" />
         <property name="javax.persistence.jdbc.user" value="sa" />
         <property name="javax.persistence.jdbc.password" value="" />
         <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect" /
              >
         <property name="hibernate.hbm2ddl.auto" value="create−drop" />
      </properties>
   </persistence−unit>
</persistence>
```

Figure 5.8: META-INF/persistence.xml file defining connection to the database and other JPA properties

```
public interface IDataAccess {

   /**
   * retrieve all accounts from database
   */
   List<Account> getAccounts();

   /**
   * get transactions for account
   */
   Set<Transaction> getTransactions(Account account);

}
```

Figure 5.9: IDataAccess interface used in the Accounts application

```
@Entity
@Table(name = "accounts")
public class Account {
    @Id
    private Long id;
    @Column
    private String name;
    @Column
    private String number;
    @OneToMany(mappedBy = "account", fetch = FetchType.LAZY)
    private Set<Transaction> transactions;
}

@Entity
@Table(name="transactions")
public class Transaction {
    @Id
    private Long id;
    @Column
    private String desc;
    @Column
    private BigDecimal amount;
    @Column
    private Date datetime;
    @ManyToOne
    @JoinColumn(name = "account_id")
    private Account account;
}
```

Figure 5.10: Source code of the domain model of the Accounts application. Classes are annotated with JPA entities.

```
public class ADataAccess implements IDataAccess, SOFALifecycle {
    private EntityManagerFactory emFactory;
    private EntityManager em;

    /** Typesafe Criteria Query is used to access database */
    @Override
    public List<Account> getAccounts() {
        CriteriaQuery<Account> query = emFactory.getCriteriaBuilder().createQuery(
                Account.class);
        query.from(Account.class);
        List<Account> accounts = em.createQuery(query).getResultList();
        return accounts;
    }

    /** Database query is generated automatically by JPA */
    @Override
    public Set<Transaction> getTransactions(Account account) {
        em.merge(account);
        return account.getTransactions();
}
```

Figure 5.11: Using JPA in ADataAccess architecture

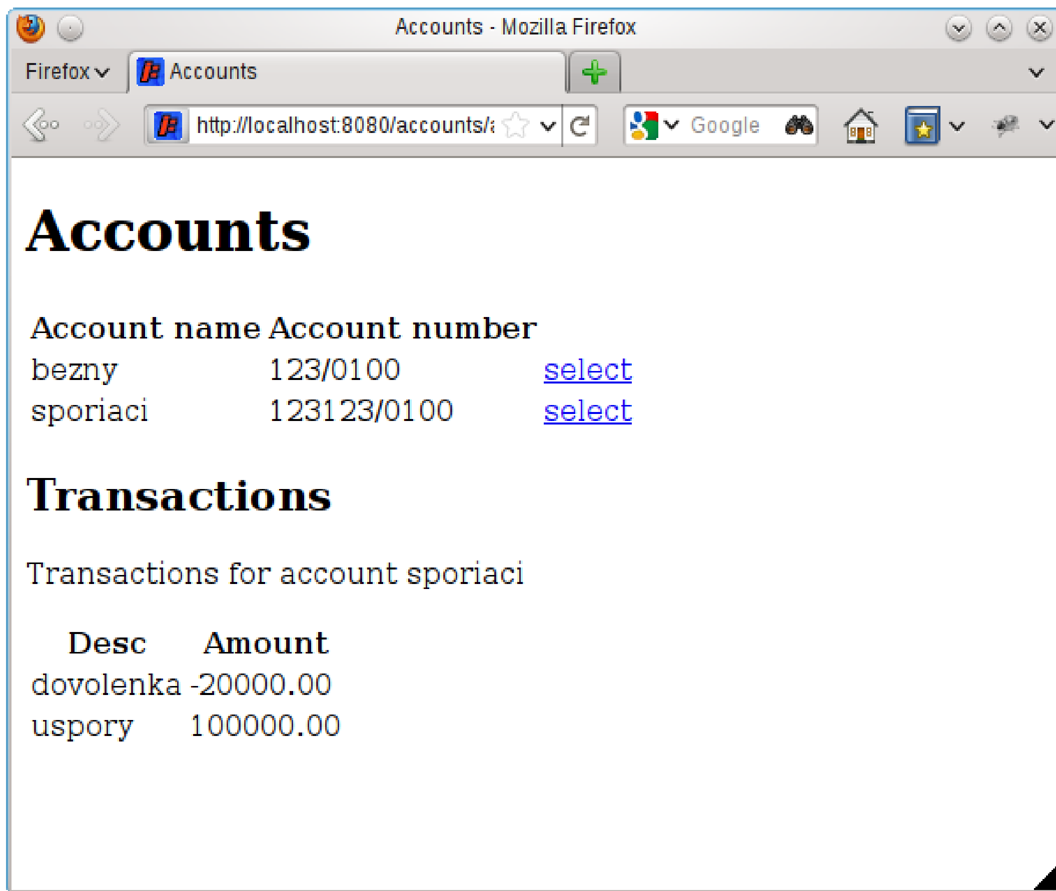


Figure 5.12: Screenshot of web browser running Accounts application.

Chapter 6

Conclusion and future work

In the presented thesis we have analyzed the possibility of implementation of an enterprise application using the SOFA 2 component system and contemporary industry technologies for developing such applications. We have created a new SOFA profile called SOFA EE which integrates technologies from Java Enterprise Edition platform.

In order to provide support for developing of enterprise applications we have introduced and described two new types of components – components for web based user interfaces and components for data management and persistence. Components are realized through new types of primitive architectures.

Main technologies integrated into SOFA EE are:

Context and Dependency Injection is the new Java EE 6 standard for componentization of enterprise applications in Java. It was integrated in order to bridge the SOFA 2 component model into the Java EE 6 component model. Integration allows leverage of contemporary frameworks for web based user interfaces like Java Server Faces.

Java Servlet API is the core API for creating web based applications in Java. We have extended SOFA 2 deployment dock to employ Jetty Servlet Container in order to provide web server functionality for the enterprise applications.

Java Persistence API is a Java object-to-relational mapping standard, which allows seamless persistence of objects into relational database systems. We have leveraged Hibernate as a JPA provider and we exposed Java Persistence API for SOFA 2 components.

We have created two simple applications to demonstrate building of enterprise applications with this model. The first application *NumberGuess* demonstrates connection of a component with web based interface to the ordinary SOFA 2 component. The second application, called *Accounts*, uses both new types of

components. It uses one component for the web based interface and another component for storing application data in the relational database. The example applications show that the chosen technologies and programming model is viable.

Table 6.1 shows specifications of Java Enterprise Edition and their support in Java EE 6 WebProfile and SOFA EE profile.

6.1 Future work

There are several possibilities to further improve and extend the SOFA EE profile. Both CDI and SOFA have a rich component model and it would be beneficial to improve interoperability of components from both worlds.

The current web based interface is implemented as a primitive architecture. It would be interesting to explore the possibility of web based interfaces created by composite architectures. Provided integration with Java Server Faces serves as a good base for this work, because it is naturally a component oriented framework.

SOFA 2 classloading mechanism is causing a lot of problems in integration of existing Java EE frameworks, because they heavily rely on Java reflection and classpath scanning. Possibility of replacing the SOFA 2 classloader with hierarchical classloaders similar to ones found in application servers could be investigated.

Table 6.1: Overview of Java EE platform technologies [10] and their support in JEE WebProfile and SOFA EE

Java EE Platform Technology	Web Profile	SOFA EE
Web Application Technologies		
JSR 315: Java Servlet 3.0	✓	✓[1]
JSR 314: JavaServer Faces (JSF) 2.0	✓	✓
JSR 245: JavaServer Pages 2.2	✓	[2]
JSR 52: A Standard Tag Library for JSP 1.2	✓	
JSR-45: Debugging Support for Other Languages 1.0	✓	
Enterprise Application Technologies		
JSR 299: Contexts and Dependency Injection	✓	✓
JSR 330: Dependency Injection for Java	✓	✓
JSR 318: Enterprise JavaBeans 3.1	✓[3]	
JSR 317: Java Persistence API 2.0	✓	✓
JSR 250: Common Annotations for the Java Platform	✓	✓
JSR 907: Java Transaction API (JTA) 1.1	✓	
JSR 303: Bean Validation 1.0	✓	
JSR 322: Java EE Connector Architecture 1.6		
JSR 914: Java Message Service (JMS) API 1.1		
JSR 919: JavaMail 1.4		
Web Services Technologies		
JSR 311: JAX-RS: The Java API for RESTful Web Services 1.1		
JSR 109: Implementing Enterprise Web Services 1.3		
JSR 224: Java API for XML-Based Web Services (JAX-WS) 2.2		
JSR 222: Java Architecture for XML Binding (JAXB) 2.2		
JSR 181: Web Services Metadata for the Java Platform		
JSR 101: Java APIs for XML based RPC 1.1		
JSR 67: Java APIs for XML Messaging 1.3		
JSR 93: Java API for XML Registries 1.0 (JAXR) 1.0		
Management and Security Technologies		
JSR 196: Java Authentication SPI for Containers 1.0		
JSR 115: Java Authorization Contract for Containers 1.3		
JSR 88: Java EE Application Deployment 1.2		
JSR 77: J2EE Management 1.1		

[1] SOFA EE supports only Servlet API 2.5, because Jetty version 7 does not support 3.0 yet
[2] SOFA EE does not support JSP due to conflict between EL version for JSP 2.1 and JSF 2.1. However support should be easily added with Servlet 3.0 in Jetty version 8, when it becomes available
[3] Web Profile supports EJB Lite, simplified version of EJB

Content of the CD

This thesis is accompanied with a CD with distribution of SOFA EE profile, source code of SOFA EE profile, source code of example applications and a pdf version of this text.

Table 2: Content of the CD

Path	**Description**
sofa-ee-dist	SOFA EE distribution
cushion-ee-dist	Distribution of Cushion tool
sofa-ee-src	Source files of SOFA EE
examples/accounts	Source files of Accounts application
examples/numberguess	Source files of Numberguess application
thesis.pdf	PDF file of this thesis
README	Instructions how to launch SOFA EE and example applications

Bibliography

[1] Tomáš Bureš, Petr Hnětynka, František Plášil, *SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model*, Aug 2006

[2] Enterprise JavaBeans Technology http://www.oracle.com/technetwork/java/javaee/ejb/index.html

[3] Spring Framework http://www.springsource.org/

[4] SOFA 2 Component System http://sofa.ow2.org/

[5] The Fractal Project http://fractal.ow2.org/

[6] Ondřej Černý, Petr Hošek, Michal Papež, Václav Remeš, *SOFA 2 Component System User's Guide*

[7] Ondřej Černý, Petr Hošek, Michal Papež, Václav Remeš, *SOFA 2 Component System Developer's Guide*

[8] Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf/

[9] Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley Signature Series, 2002

[10] Ed Ort, *Introducing the Java EE 6 Platform: Part 3*, 2009, http://java.sun.com/developer/technicalArticles/JavaEE/JavaEE6Overview_Part3.html

[11] HTTP Protocol, RFC 2616 availaible at http://www.w3.org/Protocols/rfc2616/rfc2616.html

[12] Java Servlet Technology, http://www.oracle.com/technetwork/java/javaee/servlet/index.html

[13] Java Server Faces Technology, http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html

[14] Hibernate, http://www.hibernate.org/

[15] Java Persistence API, http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html

[16] Gavin King et. al. *Weld - JSR-299 Reference Implementation* http://docs.jboss.org/weld/reference/latest/en-US/html/

[17] http://download.oracle.com/docs/cd/E19501-01/819-3659/beadf/index.html

[18] Jetty Web Server, http://www.eclipse.org/jetty/

[19] Weld, http://seamframework.org/Weld

[20] H2 Database Engine, http://www.h2database.com/html/main.html

List of Tables

2.1 Overview of Java application servers 12

2.2 Some of Java Persistence API annotations 16

2.3 Overview of CDI annotations 17

4.1 Overview of SOFA EE boostrap 27

6.1 Overview of Java EE platform technologies 43

2 Content of the CD 44

List of Figures

2.1 Example of a composite architecture with two primitive components. . . . 8
2.2 SOFA 2 runtime architecture 11
2.3 Servlet API 13
2.4 Structure of the WAR archive. 14
2.5 Class loader runtime hierarchy 19

3.1 Annotations in SOFA 2 22

4.1 Interface for extension of the Deployment Dock with capabilities. 26
4.2 Microinterface type of WebAppDeployer control interface 28
4.3 Call sequence during deploying of web interface component . . . 29
4.4 Architecture for component implementing web user interface. . . 30
4.5 MIPersistence microinterface type 30
4.6 Architecture for component implementing persistence unit. . . . 31

5.1 Architecture of the Numberguess application 33
5.2 IGuess interface from the NumberGuess sample application . . . 34
5.3 An excerpt from the JSF view template for the NumberGuess game. 34
5.4 NumberGuess CDI bean 35
5.5 Architecture AWebUI in NumberGuess application 35
5.6 Architecture of Accounts application 36
5.7 Architecture ADataAccess from Accounts application 37
5.8 META-INF/persistence.xml file defining connection to the database and other JPA properties 38
5.9 IDataAccess interface used in the Accounts application 38

5.10 Domain model of Accounts application 39
5.11 Using JPA in ADataAccess architecture 39
5.12 Screenshot of web browser running Accounts application. 40