# Open Application Standard Platform for Java

# Table of Contents

Introduction ..... vi
1. Architecture ..... 1
1.1. Key Principles ..... 1
1.2. Application Architecture ..... 1
1.2.1. Business Architecture ..... 1
1.2.2. Technical Architecture ..... 2
1.2.2.1. Technology Stack ..... 4
1.2.3. Infrastructure Architecture ..... 4
1.3. Enterprise Architecture ..... 5
2. Coding ..... 7
2.1. Coding Conventions ..... 7
2.1.1. Naming ..... 7
2.1.2. Packages ..... 7
2.1.3. Code Tasks ..... 9
2.1.3.1. TODO ..... 9
2.1.3.2. FIXME ..... 9
2.1.3.3. REVIEW ..... 9
2.1.4. Code-Documentation ..... 9
3. Layers ..... 10
3.1. Persistence Layer ..... 10
3.1.1. Entity ..... 10
3.1.1.1. A Simple Entity ..... 10
3.1.1.2. Entities and Datatypes ..... 10
Enumerations ..... 11
BLOB ..... 11
Date and Time ..... 12
3.1.1.3. Primary Keys ..... 12
3.1.2. Data Access Object ..... 12
3.1.2.1. DAO Interface ..... 12
3.1.2.2. DAO Implementation ..... 12
3.1.3. Queries ..... 13
3.1.3.1. Using Queries to Avoid Bidirectional Relationships ..... 14
3.1.4. Concurrency Control ..... 14
3.1.4.1. Optimistic Concurrency Control ..... 14
3.1.4.2. Pessimistic Concurrency Control ..... 15
3.1.5. Relationships ..... 16
3.1.5.1. n:1 and 1:1 Relationships ..... 16
3.1.5.2. 1:n and n:m Relationships ..... 16
3.1.5.3. Eager vs. Lazy Loading ..... 18
3.1.5.4. Cascading Relationships ..... 18
3.1.6. Embeddable ..... 19
3.1.7. Inheritance ..... 20
3.1.8. Testing Entities and DAOs ..... 22
3.1.9. Principles ..... 23
3.1.10. Security ..... 23
3.1.11. Database Configuration ..... 23
3.1.11.1. Configuration Of The Database System ..... 23

3.2. Logic Layer ..... 26
3.2.1. Use Case ..... 26
3.2.2. Component Interface ..... 27
3.2.2.1. Passing Parameters Among Components ..... 28
3.2.2.2. Use Case Example ..... 28
3.3. Service Layer ..... 30
3.3.1. Types of Services ..... 30
3.3.2. Versioning ..... 31
3.3.3. Interoperability ..... 31
3.3.4. Protocol ..... 31
3.3.4.1. SOAP ..... 31
JAX-WS ..... 31
SOAP Custom Mapping ..... 32
SOAP Testing ..... 32
3.3.4.2. REST ..... 32
JAX-RS ..... 33
HTTP Status Codes ..... 34
REST Exception Handling ..... 35
REST Media Types ..... 35
REST Testing ..... 35
3.3.4.3. HTTP-Invoker ..... 35
3.3.5. Service Considerations ..... 36
3.4. Client Layer ..... 37
3.4.1. Web Clients ..... 37
3.4.2. Native Desktop Clients ..... 37
3.4.3. Native Mobile Clients ..... 37
3.5. Batches ..... 38
4. Guides ..... 39
4.1. Logging ..... 39
4.1.1. Usage ..... 39
4.1.1.1. Maven Integration ..... 39
4.1.1.2. Configuration ..... 39
4.1.1.3. Logger Access ..... 39
4.1.1.4. How to log ..... 39
4.1.2. Operations ..... 40
4.1.2.1. Log Files ..... 40
4.1.2.2. Output format ..... 41
4.1.3. Security ..... 41
4.2. Security ..... 42
4.2.1. Authentication ..... 42
4.2.1.1. Mechanisms ..... 42
Basic ..... 42
Form Login ..... 42
Preserve original request anchors after form login redirect ..... 42
4.2.1.2. Users vs. Systems ..... 43
4.2.2. Authorization ..... 43
4.2.2.1. Access Management ..... 43
4.2.2.2. Method Authorization ..... 43
4.2.2.3. Check Role based Permissions ..... 44
4.2.2.4. Check Data based Permissions ..... 44

4.2.3. Vulnerability ..... 45
4.3. Dependency Injection ..... 46
4.3.1. Example Bean ..... 46
4.3.2. Spring Usage and Conventions ..... 46
4.3.2.1. Spring XML Files ..... 47
4.3.3. Key Principles ..... 47
4.4. Configuration ..... 49
4.4.1. Configuration Files ..... 49
4.5. Validation ..... 50
4.5.1. Example ..... 50
4.5.2. GUI-Integration ..... 50
4.5.3. Cross-Field Validation ..... 50
4.5.4. Complex Validation ..... 50
4.6. Aspect Oriented Programming (AOP) ..... 51
4.6.1. Transactions ..... 51
4.6.2. Security ..... 51
4.6.3. Exception Facade ..... 51
4.6.4. Logging ..... 51
4.6.5. AOP Key Principles ..... 51
4.7. Exception Handling ..... 52
4.8. Internationalization ..... 53
4.9. Monitoring ..... 54
4.9.1. JMX ..... 54
4.9.1.1. MBeans ..... 54
4.9.1.2. Using JMX ..... 54
4.9.1.3. Integration with Nagios ..... 54
4.9.2. GC-Logging ..... 54
4.9.3. Loadbalancing ..... 54
4.10. XML ..... 55
4.10.1. JAXB ..... 55
4.10.1.1. JAXB and Inheritance ..... 55
4.10.1.2. JAXB Custom Mapping ..... 55
4.11. JSON ..... 56
4.11.1. JSON and Inheritance ..... 56
4.11.2. JSON Custom Mapping ..... 57
4.12. Testing ..... 59
4.12.1. Module Testing ..... 59
4.12.2. Integration Testing ..... 59
4.12.3. System Testing ..... 59
4.12.4. Manual Developer Tests ..... 59
4.12.5. Testdata ..... 59
4.13. Transfer-Objects ..... 61
4.13.1. Business-Objects ..... 61
4.13.2. Service-Objects ..... 61
4.14. Datatypes ..... 62
4.14.1. Datatype Packaging ..... 62
4.14.2. Datatypes in Entities ..... 63
4.14.3. Datatypes in Transfer-Objects ..... 63
4.14.3.1. JAXB ..... 63
4.14.3.2. JAX-RS ..... 63

4.14.3.3. JAX-WS ..... 63
4.15. Transaction Handling ..... 64
4.15.1. Batches ..... 64
4.16. Accessibility ..... 65

# Introduction

This document contains the complete compendium of the *Open Application Standard Platform for Java* (OASP4J). The most recent version of this document as PDF can be found here.

# 1. Architecture

There are many different views on what is summarized by the term *architecture*. First we introduce the key principles for the OASP. For the further consideration we separate the architecture of a single application and the architecture of an entire application landscape.

## 1.1 Key Principles

For the OASP we follow these fundamental key principles for architecture and design:

- **KISS**
  Keep it small and simple
- **Open**
  Commitment to open standards and solutions (no required dependencies to commercial or vendor-specific standards or solutions)
- **Patterns**
  We concentrate on providing patterns, best-practices and examples rather than writing framework code.
- **Solid**
  We pick solutions that are established and have proved to be solid and robust in real-live (business) projects.

## 1.2 Application Architecture

For the architecture of an application we distinguish the following views:

- The Business Architecture describes an application from the business perspective. It divides the application into business components and with full abstraction of technical aspects.
- The Technical Architecture describes an application from the technical implementation perspective. It divides the application into technical layers and defines which technical products and frameworks are used to support these layers.
- The Infrastructure Architecture describes an application from the operational infrastructure perspective. It defines the nodes used to run the application including clustering, loadbalancing and networking.

### 1.2.1 Business Architecture

The *business architecture* divides the application into *business components*. A business component has a well-defined responsibility that it encapsulates. All aspects related to that responsibility have to be implemented within that business component. Further the business architecture defines the dependencies between the business components. These dependencies need to be free of cycles. A business component exports his functionality via well-defined interfaces as a self-contained API. A business component may use another business component via its API and compliant with the dependencies defined by the business architecture.

As the business domain and logic of an application can be totally different, the OASP can not define a standardized business architecture. Depending on the business domain it has to be defined from scratch

or from a domain reference architecture template. For very small systems it may be suitable to define just a single business component containing all the code.

### 1.2.2 Technical Architecture

The *technical architecture* divides the application into technical *layers* based on the multilayered architecture. A layer is a unit of code with the same technical category such as persistence or presentation logic. A layer is therefore often supported by a technical framework. Each business component can therefore be split into *component parts* for each layer. However, a business component may not have component parts for every layer (e.g. only a presentation part that utilized logic from other components).

An overview of the technical reference architecture of the OASP is given by figure "Technical Reference Architecture". It defines the following layers visualized as horizontal boxes:

- client layer for the front-end (GUI).
- service layer for the services used to expose functionality of the back-end to the client or other consumers.
- logic layer for the business logic.
- persistence layer for the persistence (data access).

Also you can see the (business) components as vertical boxes (e.g. *A* and *X*) and how they are composed out of component parts each one assigned to one of the technical layers.

Further, there are technical components for cross-cutting aspects grouped by the gray box on the right. Here is a complete list:

- Security
- Logging
- Monitoring
- Transaction-Handling
- Exception-Handling
- Internationalization
- Dependency-Injection

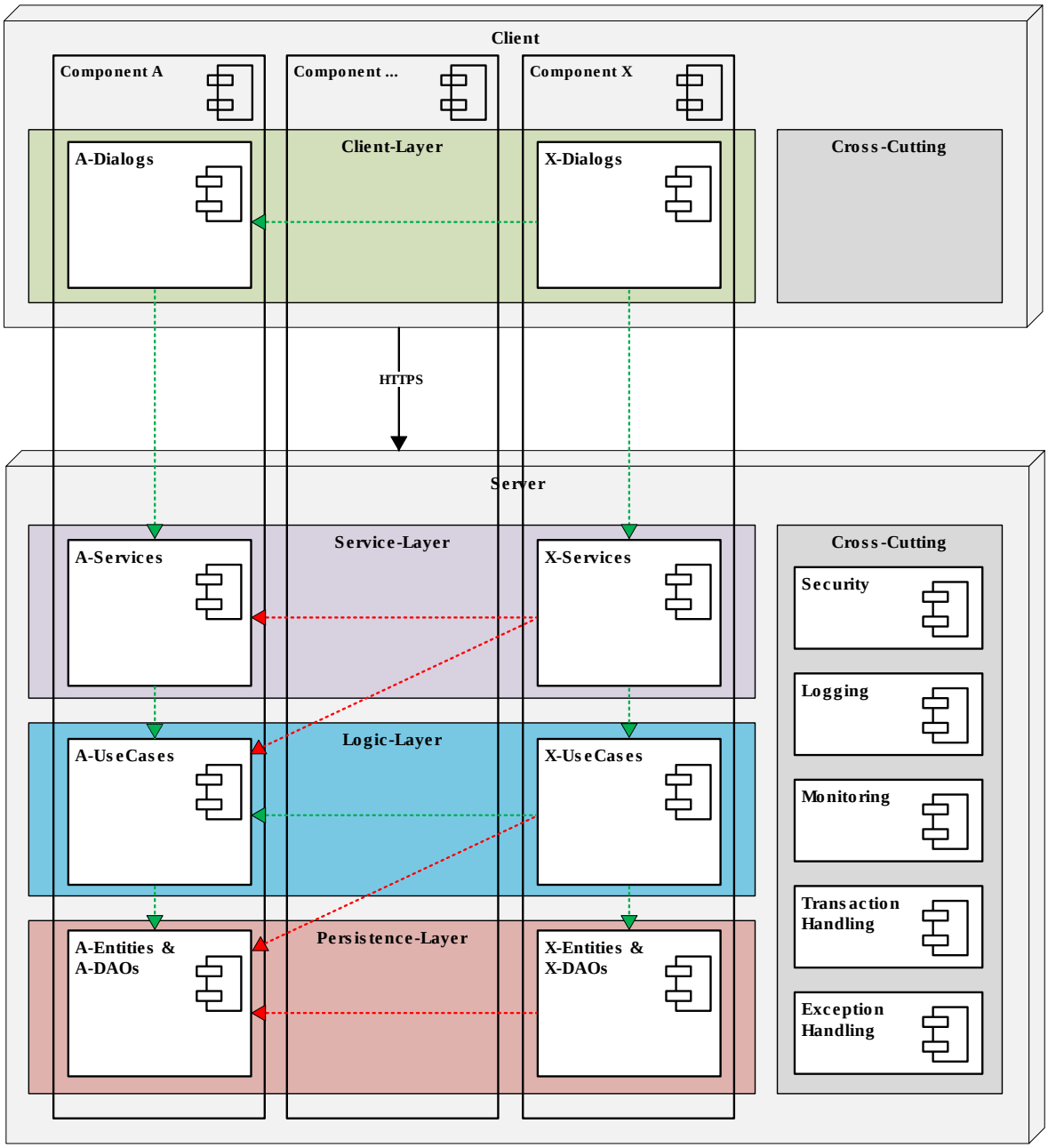



*Figure 1.1. Technical Reference Architecture*

We reflect this architecture in our code as described in our coding conventions allowing a traceability of business components, use-cases, layers, etc. into the code and giving developers a sound orientation within the project.

Further, the architecture diagram shows which dependencies are allowed and which are forbidden. Within a business component a component part can call the next component part on the layer directly below via a dependency on its API. This is illustrated by the vertical green dotted arrows. While this is natural and obvious it is generally forbidden to have dependencies upwards the layers or to skip a layer by a direct dependency on a component part two or more layers below. The general dependencies allowed between business components are defined by the business architecture. In our reference architecture diagram we assume that the business component X is allows to depend on component A. Therefore a use-case within the logic component part of X is allows to call a use-case from A via a dependency on the API. The same applies for dialogs on the client layer. This is illustrated by the horizontal green dotted arrows while the red dotted arrows show that other dependencies are not

allowed. Please note that persistence entities are part of the API of the persistence component part so only the logic component part of the same business component may depend on them.

#### 1.2.2.1 Technology Stack

The technology stack of the OASP is illustrated by the following table.

*Table 1.1. Technology Stack of OASP*

| Topic | Detail | Standard | Suggested implementation |
|---|---|---|---|
| runtime | language & VM | Java | Oralce JDK |
| runtime | servlet-container | JEE | tomcat |
| persistence | OR-mapper | JPA | hibernate |
| batch | framework | JSR352 | spring-batch |
| service | SOAP services | JAX-WS | TODO CXF or metro2 |
| service | REST services | JAX-RS | CXF |
| logging | framework | slf4j | logback |
| validation | framework | beanvalidation/JSR303 | hibernate-validator |
| component management | dependency injection | JSR330 & JSR250 | spring |
| security | Authentication & Authorization | JAAS | spring-security |
| monitoring | framework | JMX | spring |
| monitoring | HTTP Bridge | HTTP & JSON | jolokia |
| AOP | framework | dynamic proxies | spring AOP |

### 1.2.3 Infrastructure Architecture

The *infrastructure architecture* describes an application from the operational infrastructure perspective. It defines the nodes (physical or virtual machines) used to run the application as well as additional devices such as loadbalancers, firewalls, etc. and the communication paths between these elements.

An overview of the infrastructure reference architecture of the OASP is given by figure "Infrastructure Reference Architecture". Please note that it is based on SAGA and the enterprise architecture of the OASP. The infrastructure reference architecture defines the following tiers visualized as dashed boxes:

- Information-Tier
- Logic-Tier
- Data-Tier

Each of these tiers technically represent a demilitarized zone (DMZ) and are therefore separated by firewalls.

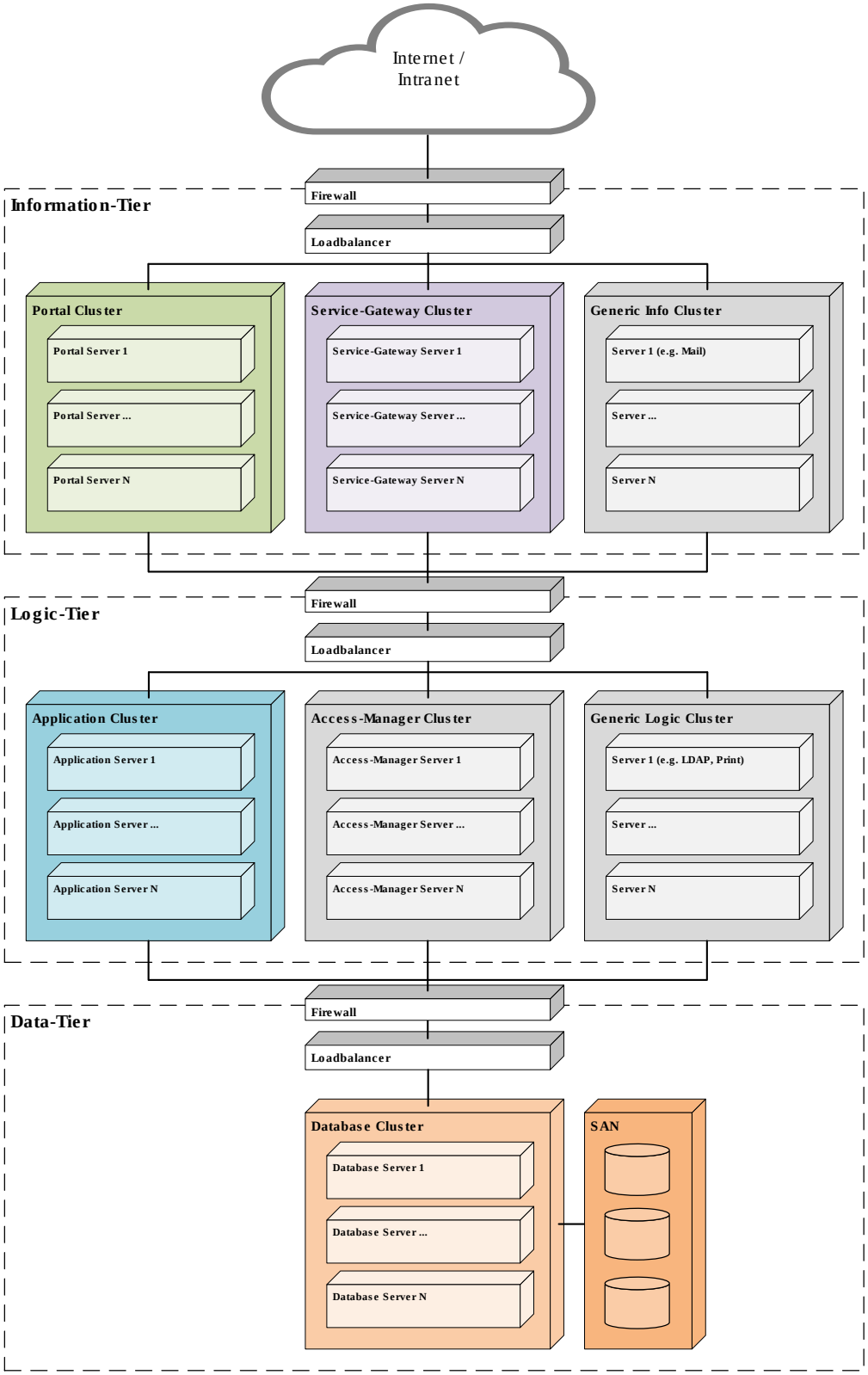



*Figure 1.2. Infrastructure Reference Architecture*

## 1.3 Enterprise Architecture

If you are using the OASP(4J) in your enterprise as a basis for an entire IT application landscape you can make even more benefit and savings. For such case we also define best practices for the enterprise

architecture. An overview of the infrastructure reference architecture of the OASP is given by figure "Enterprise Reference Architecture". TODO

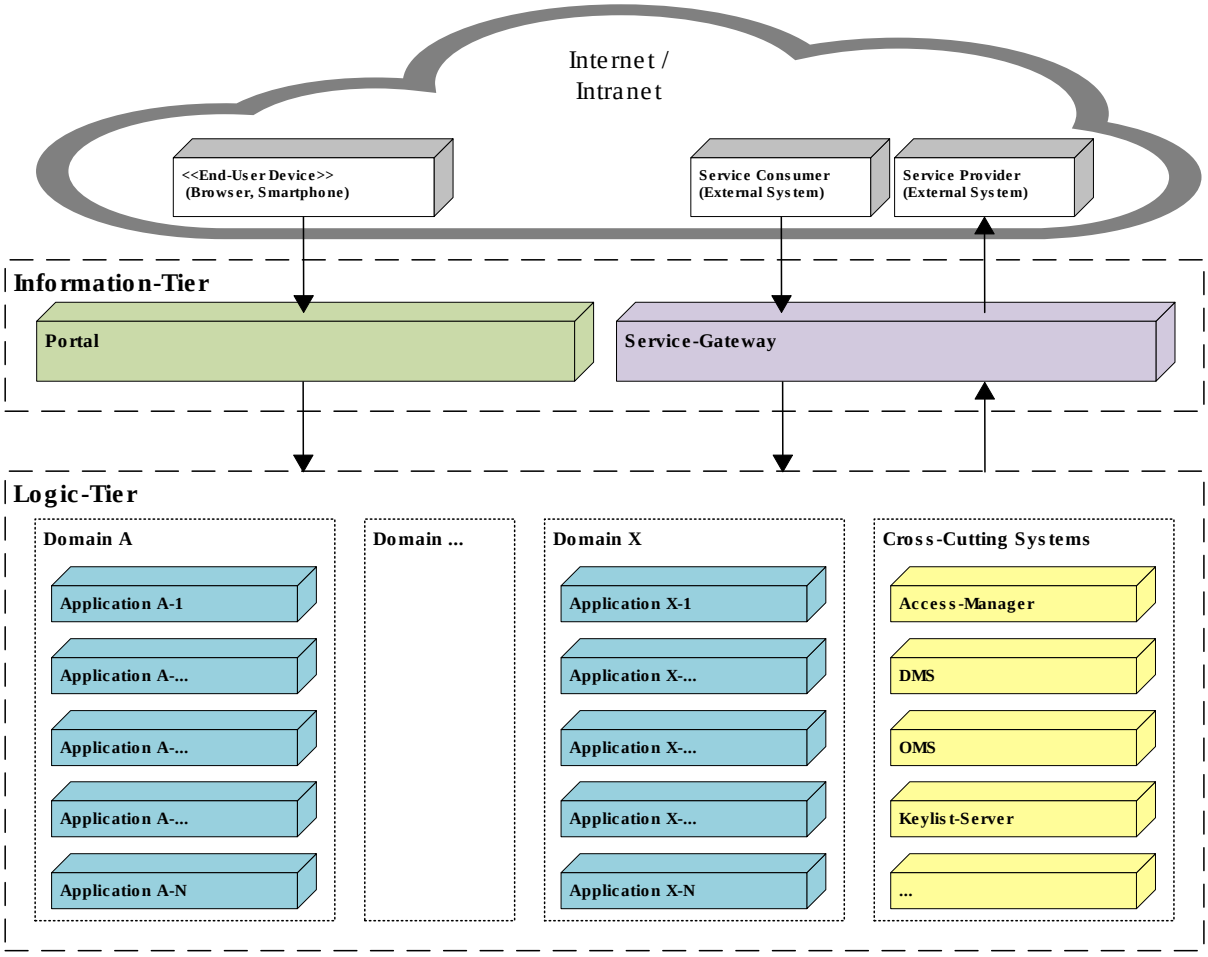



*Figure 1.3. Enterprise Reference Architecture*

# 2. Coding

## 2.1 Coding Conventions

The code should follow general conventions for Java (see Oracle Naming Conventions, Google Java Style, etc.).We consider this as common sense and provide configurations for SonarQube and related tools such as Checkstyle instead of repeating this here.

### 2.1.1 Naming

Besides general Java naming conventions, we follow the additional rules listed here explicitly:

- Always use short but speaking names (for types, methods, fields, parameters, variables, constants, etc.).
- Avoid having duplicate type names. The name of a class, interface, enum or annoation should be unique within your project unless this is intentionally desired in a special and reasonable situation.
- Avoid artificial naming constructs such as prefixes (I*) or suffixes (*IF) for interfaces.
- Use CamlCase even for abbreviations (XmlUtil instead of XMLUtil)
- Names of Generics should be easy to understand. Where suitable follow the common rule E=Element, T=Type, K=Key but feel free to use longer names for more specific cases such as ID, DTO or ENTITY. The capitalized naming helps to distinguish a generic type from a regular class.

### 2.1.2 Packages

Java Packages are the most important element to structure your code. We use a strict packaging convention to map technical layers and business components (slices) to the code (See technical architecture for further details). By using the same names in documentation and code we create a strong link that gives orientation and makes it easy to find from business requirements, specifications or story tickets into the code and back. Further we can use tools such as SonarQube and SonarGraph to verify architectural rules.

For an OASP based application we use the following Java-Package schema:

```
<basepackage>.<component>.<layer>.<scope>[.<detail>]*
```

For an application as part of an IT application landscape we recommend to use the followng schema for <basepackage>:

```
<organization>.<domain>.<application>
```

E.g. in our example application we find the DAO interfaces for the salesmanagement component in the package org.oasp.gastronomy.restaurant.salesmanagement.persistence.api.dao

*Table 2.1. Segments of package schema*

| Segment | Description | Example |
|---|---|---|
| <organization> | Is the basic Java Package name-space of the organization | org.oasp |

| Segment | Description | Example |
|---|---|---|
| | owning the code following common Java Package conventions. Consists of multiple segments corresponding to the Internet domain of the organization. | |
| <domain> | Is the business domain of the application. Especially important in large enterprises that have an large IT landscape with different domains. | gastronomy |
| <application> | The name of the application build in this project. | restaurant |
| <component> | The (business) component the code belongs to. It is defined by the business architecture and uses terms from the business domain. Use the implicit component general for code not belonging to a specific component (foundation code). | salesmanagement |
| <layer> | The name of the technical layer (See technical architecture) which is one of the predefined layers (persistence, logic, service, batch, gui, client) or common for code not assigned to a technical layer (datatypes, cross-cutting concerns). | persistence |
| <scope> | The scope which is one of api (official API to be used by other layers or components), base (basic code to be reused by other implementations) and impl (implementation that should never be imported from outside) | api |
| <detail> | Here you are free to further divide your code into sub-components and other concerns according to the size of your component part. | dao |

Please note that for library modules where we use org.oasp.module as <basepackage> and the name of the module as <component>. E.g. the API of our monitoring module can be found in the package org.oasp.module.monitoring.common.api.

### 2.1.3 Code Tasks

Code spots that need some rework can be marked with the following tasks tags. These are already properly pre-configured in your development environment for auto completion and to view tasks you are responsible for. It is important to keep the number of code tasks low. Therefore every member of the team should be responsible for the overall code quality. So if you change a piece of code and hit a code task that you can resolve in a reliable way do this as part of your change and remove the according tag.

#### 2.1.3.1 TODO

Used to mark a piece of code that is not yet complete (typically because it can not be completed due to a dependency on something that is not ready).

```
// TODO <author> <description>
```

A TODO tag is added by the author of the code who is also responsible for completing this task.

#### 2.1.3.2 FIXME

```
// FIXME <author> <description>
```

A FIXME tag is added by the author of the code or someone who found a bug he can not fix right now. The <author> who added the FIXME is also responsible for completing this task. This is very similar to a TODO but with a higher priority. FIXME tags indicate problems that should be resolved before a release is completed while TODO tags might have to stay for a longer time.

#### 2.1.3.3 REVIEW

```
// REVIEW <responsible> (<reviewer>) <description>
```

A REVIEW tag is added by a reviewer during a code review. Here the original author of the code is responsible to resolve the REVIEW tag and the reviewer is assigning this task to him. This is important for feedback and learning and has to be aligned with a review "process" where people talk to each other and get into discussion. In smaller or local teams a peer-review is preferable but this does not scale for large or even distributed teams.

### 2.1.4 Code-Documentation

As a general goal the code should be easy to read and understand. Besides clear naming the documentation is important. We follow these rules:

- APIs (especially component interfaces) are properly documented with JavaDoc.
- JavaDoc shall provide actual value - we do not write JavaDoc to satisfy tools such as checkstyle but to express information not already available in the signature.
- We make use of {@link} tags in JavaDoc to make it more expressive.
- JavaDoc of APIs describes how to use the type or method and not how the implementation internally works.
- To document implementation details, we use code comments (e.g. // we have to flush explicitly to ensure version is up-to-date). This is only needed for complex logic.

# 3. Layers

## 3.1 Persistence Layer

### 3.1.1 Entity

Entities are part of the persistence layer and contain the actual data. They are POJOs (Plain Old Java Objects) on which the relational data of a database is mapped and vice versa. The mapping is done utilizing the Java Persistence API (JPA). Usually an entity class corresponds to a table of a database and a property to a row of that table. As JPA implementation we recommend to use hibernate. For general documentation about JPA and hibernate follow the links above as we will not replicate the documentation. Here you will only find guidelines and examples how we recommend to use it properly. The following examples show how to map the data of a database to an entity.

#### 3.1.1.1 A Simple Entity

We use the JPA annotations (javax.persistence) to define how the attributes of an entity are mapped to columns of a database table. The following listing shows a simple example:

```
@Entity
@Table(name="TEXTMESSAGE")
public class Message extends AbstractEntity<Long> {

  private String text;

  public String getText() {
    return this.name;
  }

  public void setText(String text) {
    this.text = text;
  }
 }
```

The @Entity annotation defines that instances of this class will be entities which can be stored in the database. The @Table annotation is optional and can be used to define the name of the corresponding table in the database. If it is not specified, the simple name of the entity class is used instead.

In order to specify how to map the attributes to columns we annotate the corresponding getter methods (technically also private field annotation is also possible but approaches can not be mixed). The @Id annotation specifies that an property should be used as primary key. With the help of the @Column annotation it is possible to define the name of the column that an attribute is mapped to as well as other aspects such as nullable or unique. If no column name is specified, the name of the property is used as default.

Note that every entity class needs a constructor with public or protected visibility that does not have any arguments. Moreover, neither the class nor its getters and setters may be final.

Entities should be simple POJOs and not contain business logic.

#### 3.1.1.2 Entities and Datatypes

Standard datatypes like Integer, BigDecimal, String, etc. are mapped automatically by JPA. Custom datatypes are mapped as serialized BLOB by default what is typically undesired. The JPA itself does not support a way to properly map immutable custom datatypes (only as mutable types via @Embeddable). As the OASP suggests to use hibernate, we explain a hibernate-specific solution here.

We can write a hibernate specific UserType for each custom datatype to define the mapping:

```
@MappedSuperclass
@TypeDef(defaultForType = PostalCode.class, typeClass = PostalCodeUserType.class)
public class PostalCodeUserType extends StringDatatypeUserType<PostalCode> {

  public PostalCodeUserType() {
    super(PostalCode.class);
  }

  protected PostalCode toDatatype(String value) {
    return new PostalCode(value);
  }
}
```

Using @MappedSuperclass is just a trick to make hibernate find the class when it scans for entities. Further, @TypeDef allows to register the user type as default mapping for the custom datatype (here PostalCode). Therefore all entities with properties of that datatype will automatically be mapped properly (in our example as String). For further details see here.

> **Caution**
>
> Avoid using CompositeUserType as hibernate fails to support queries on the individual attributes. Either avoid composite datatypes at all if possible or use an @Embeddable instead.

#### Enumerations

By default JPA maps Enums via their ordinal. This is sensible if you change the order of the enum values in the code after being in production with your application. To avoid this and get a robust mapping you can

- treat enums as custom datatypes
- use the @Enumerated annotation for the property mapping (using EnumType.STRING)
- use the @PersistentValue annotation to annotate each enum value in the enum declaration.

> **Note**
>
> We should avoid giving multiple options without a strict recommendation.

#### BLOB

If binary or character large objects (BLOB/CLOB) should be used to store the value of an attribute, e.g. to store an icon, the @Lob annotation should be used as shown in the following listing:

```
@Lob
public byte[] getIcon() {
    return this.icon;
}
```

> **Warning**
>
> Using a byte array might cause problems if large BLOBs are used because the entire BLOB is loaded into the RAM of the server and has to be processed by the garbage collector. For larger BLOBs the type Blob and streaming should be used.

```
public Blob getAttachment() {
    return this.attachment;
}
```

**Date and Time**

To store date and time related values, the temporal annotation can be used as shown in the listing below:

```
@Temporal(TemporalType.TIMESTAMP)
public java.util.Date getStart() {
    return start;
}
```

Until Java8 the java data type java.util.Date (or Jodatime) has to be used. TemporalType defines the granularity. In this case, a precision of nanoseconds is used. If this granularity is not wanted, TemporalType.DATE can be used instead, which only has a granularity of milliseconds. Mixing these two granularities can cause problems when comparing one value to another. This is why we **only** use TemporalType.TIMESTAMP.

More advanced mappings can be found within the chapters about relationships and inheritance.

#### 3.1.1.3 Primary Keys

We only use simple Long values as primary keys (IDs). By default it is auto generated (@GeneratedValue(strategy=GenerationType.AUTO)). This is already provided by the class AbstractEntity that you can extend. In case you have business oriented keys (often as String), you can define an additional property for it and declare it as unique (@Column(unique=true)).

### 3.1.2 Data Access Object

*Data Acccess Objects* (DAOs) are part of the persistence layer. They are responsible for a specific entity and should be named <Entity>Dao[Impl]. The DAO offers the so called CRUD-functionalities (create, retrieve, update, delete) for the corresponding entity. Additionally a DAO may offer advanced operations such as query or locking methods.

#### 3.1.2.1 DAO Interface

For each DAO there is an interface named <Entity>Dao that defines the API. For CRUD support and common naming we derive it from the interface AbstractDao:

```
public interface MyEntityDao extends AbstractDao<MyEntity, Long> {

   List<MyEntity> findByCriteria(MyEntitySearchCriteria criteria);
}
```

As you can see, the interface GenericDao has two type parameters:

- the first one ENTITY for the entity class
- the second one ID for the type of the primary key.

#### 3.1.2.2 DAO Implementation

Implementing a DAO is quite simple. We crate a class named <Entity>DaoImpl that extends AbstractDaoImpl and implements our <Entity>Dao interface:

```
public class MyEntityDaoImpl extends AbstractDaoImpl<MyEntity, Long> implements MyEntityDao {

  public List<MyEntity> findByCriteria(MyEntitySearchCriteria criteria) {
    TypedQuery<MyEntity> query = createQuery(criteria);
```

```
    return query.getResultList();
  }
  ...
}
```

In the DAO implementation the method getEntityManager() can be used to access the EntityManager from the JPA that provides several methods to access the database. For example, with the help of this entity manager it is not only possible to implement CRUD, but also to execute queries, see here how this is done.

### 3.1.3 Queries

The Java Persistence API (JPA) defines its own query language, the java persistence query language (JPQL), which is similar to SQL but operates on entities and their attributes instead of tables and columns.

The OASP4J advises to specify all queries in one mapping file called NamedQueries.hbm.xml.

Add the following query to this file:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://
www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping default-lazy="true">
    <query name="staff.member.search.by.name">

        <![CDATA[select staff from StaffMember staff where staff.firstName = :firstName and
 staff.lastName = :lastName]]>

    </query>
</hibernate-mapping>
```

To be able to easily change the name of the query name (staff.member.search.by.name) later on, it is advisable to create a new class the contains all query names as constants:

```
package com.capgemini.gastronomy.restaurant.common.constants;

public class NamedQueries {

    public static final String STAFFMEMBER_SEARCH_BY_NAME = "staff.member.search.by.name";

}
```

Note that changing the name of the java constant (STAFFMEMBER_SEARCH_BY_NAME) can be done without much effort using Eclipse refactoring.

The following listing shows how to use this query (in class StaffMemberDaoImpl, remember to adapt StaffMemberDao!):

```
public List<StaffMember> getStaffMemberByName(String firstName, String lastName) {
    Query query = getEntityManager().createNamedQuery(NamedQueries.STAFFMEMBER_SEARCH_BY_NAME);

    query.setParameter("firstName", firstName);
    query.setParameter("lastName", lastName);

    return query.getResultList();
}
```

The EntityManager contains a method called createNamedQuery(String), which takes as parameter the name of the query and creates a new query object. As the query has two parameters, these have to be set using the setParameter(String, Object) method.

Note that using the createQuery(String) method, which takes as parameter the query as string (this string already contains the parameters) is not allowed as this makes the application vulnerable to SQL injection attacks.
When the method getResultList() is invoked, the query is executed and the result is delivered as list. As an alternative, there is a method called getSingleResult(), which returns the entity if the query returned exactly one and throws an exception otherwise.

#### 3.1.3.1 Using Queries to Avoid Bidirectional Relationships

With the usage of queries it is possible to avoid bidirectional relationships, which have some disadvantages (see relationships). So for example to get all WorkingTime's for a specific StaffMember without having an attribute in the StaffMember's class that stores these WorkingTime's, the following query is needed:

```
<query name="working.time.search.by.staff.member">

     <![CDATA[select work from WorkingTime work where work.staffMember = :staffMember]]>

</query>
```

The method looks as follows (extract of class WorkingTimeDaoImpl):

```
public List<WorkingTime> getWorkingTimesForStaffMember(StaffMember staffMember) {
    Query query = getEntityManager().createNamedQuery(NamedQueries.WORKING_TIMES_SEARCH_BY_STAFFMEMBER);

    query.setParameter("staffMember", staffMember);

    return query.getResultList();
}
```

Do not forget to adapt the WorkingTimeDao interface and the NamedQueries class accordingly.

To get a more detailed description of how to create queries using JPQL, please have a look here.

### 3.1.4 Concurrency Control

*The concurrency control defines the way concurrent access to the same data of a database is handled. When several users (or threads of application servers) concurrently accessing a database, anomalies may happen, e.g. a transaction is able to see changes from another transaction although that one did not jet commit these changes. Most of these anomalies are automatically prevented by the database system, depending on the isolation level used* (You can set the isolation level for the entityManagerFactory with the property hibernate.connection.isolation in the jpa.xml, see here for information on what values are possible).

One anomaly that is not prevented (at least when not using the isolation level serializable) are lost updates.
When two stakeholders concurrently access a record, compute some changes and write them back to the database, it may happen that the changes of one of the stakeholders are overwritten an so his/her updates are lost.

#### 3.1.4.1 Optimistic Concurrency Control

To avoid anomalies by concurrent data access, we use an optimistic locking approach. Optimistic concurrency control assumes that multiple database transactions can complete without affecting each other, and that therefore transactions can proceed without locking the data resources that they affect. This is beneficial, because a conflicting accesses to one entity happens rarely in most register applications and so the overhead for concurrency control can be kept small by using this approach.

We use optimistic concurrency control with versioning by default (which is implemented by Hibernate). Every entity gets a version attribute, which is incremented with every update of the corresponding row in the database. With the help of this attribute it is possible to detect concurrent updates. When an entity should be updated and the version attribute of the java object is lower than that of the row in the database, another transaction had modified this entity in the meantime. In this case the former transaction has to be rolled back to avoid lost updates.
The approach with versions also helps to avoid long running transactions. An entity can be read from the database within one transaction. If that entity is modified, the changes can be written back to the database within a separate transaction, because concurrent updates can be detected using the version attribute.

To use the versioning-approach, a new version attribute has to be introduced to the StaffMember entity, as shown in the following listing:

```
...

@Entity
public class StaffMember {
    ...
    private int version;
    ...
    @Version
    public int getVersion() {
        return version;
    }
    public void setVersion(int version) {
        this.version = version;
    }
    ...
}
```

The @Version annotation defines that this attribute should be used for storing the current version of the entity.
Fortunately, this is all that needs to be done, incrementing and controlling the version attribute is done by hibernate automatically.

#### 3.1.4.2 Pessimistic Concurrency Control

As already mentioned, optimistic concurrency control is beneficial as long as concurrent write accesses to one entity happens only rarely. If this is not the case, the version-based approach would cause a lot of transaction rollbacks, because many concurrent updates would be detected at the end of transactions. Additionally, deadlocks may occur.
In these cases pessimistic concurrency control, i.e. the use of explicit locks, should be preferred for that type of entities. Supposed for example that whenever there is a table to which no waiter is assigned to, all waiters promptly recognize that and try to assign that table to themselves (this is not what you would normally expect, just for demonstration purposes).
Note that when you implement the entity type Table both table and number are key words in Oracle 11g, so they cannot be used to name a relation or column.

The following listing shows how to retrieve a Table entity from the database and lock it, so that no one else can write on that entity until the current transaction commits:

```
public Table readTableWithWriteLock(int number) {
       Table ret = em.find(Table.class, number);
       em.lock(ret, LockModeType.READ);
       return ret;
}
```

When using the lock(Object, LockModeType) method with LockModeType.READ, Hibernate will issue a select ... for update. This means that no one else can update the entity (see here for more information on the statement). If LockModeType.WRITE is specified, Hibernate issues a select ... for update nowait instead, which has has the same meaning as the statement above, but if there is already a lock, the program will not wait for this lock to be release. Instead, an exception is raised.
Use one of the types if you want to modify the entity later on, for read only access no lock is required.

As you might have noticed, the behavior of Hibernate deviates from what one would expect by looking at the LockModeType (especially LockModeType.READ should not cause a select ... for update to be issued). The framework actually deviates from what is specified in the JPA for unknown reasons.

## 3.1.5 Relationships

### 3.1.5.1 n:1 and 1:1 Relationships

Entities often do not exist independently but are in some relation to each other. For example, for every period of time one of the StaffMember's of the restaurant example has worked, which is represented by the class WorkingTime, there is a relationship to this StaffMember.

The following listing shows how this can be modeled using JPA:

```
...

@Entity

public class WorkingTime {

   ...

   private StaffMember staffMember;

   @ManyToOne
   @JoinColumn(name="STAFFMEMBER")
   public StaffMember getStaffMember() {
      return staffMember;
   }

   public void setStaffMember(StaffMember staffMember) {
      this.staffMember = staffMember;
   }
}
```

To represent the relationship, an attribute of the type of the corresponding entity class that is referenced has been introduced. The relationship is a n:1 relationship, because every WorkingTime belongs to exactly one StaffMember, but a StaffMember usually worked more often than once.
This is why the @ManyToOne annotation is used here. For 1:1 relationships the @OneToOne annotation can be used which works basically the same way. To be able to save information about the relation in the database, an additional column in the corresponding table of WorkingTime is needed which contains the primary key of the referenced StaffMember. With the name element of the @JoinColumn annotation it is possible to specify the name of this column.

### 3.1.5.2 1:n and n:m Relationships

The relationship of the example listed above is currently an unidirectional one, as there is a getter method for retrieving the StaffMember from the WorkingTime object, but not vice versa.

To make it a bidirectional one, the following code has to be added to StaffMember:

```
private Set<WorkingTimes> workingTimes;
```

```
@OneToMany(mappedBy="staffMember")
public Set<WorkingTime> getWorkingTimes() {
     return workingTimes;
}

public void setWorkingTimes(Set<WorkingTime> workingTimes) {
   this.workingTimes = workingTimes;
}
```

To make the relationship bidirectional, the tables in the database do not have to be changed. Instead the column that corresponds to the attribute staffMember in class WorkingTime is used, which is specified by the mappedBy element of the @OneToMany annotation. Hibernate will search for corresponding WorkingTime objects automatically when a StaffMember is loaded.

The problem with bidirectional relationships is that if a WorkingTime object is added to the set or list workingTimes in StaffMember, this does not have any effect in the database unless the staffMember attribute of that WorkingTime object is set. That is why the OASP4J advices not to use bidirectional relationships but to use queries instead. How to do this is shown here. If a bidirectional relationship should be used nevertheless, approriate add and remove methods must be used.

For 1:n and n:m relations, the OASP4J demands that (unordered) Sets and no other collection types are used, as shown in the listing above. The only exception is whenever an ordering is really needed, (sorted) lists can be used.
For example, if WorkingTime objects should be sorted by their start time, this could be done like this:

```
private List<WorkingTimes> workingTimes;

    @OneToMany(mappedBy = "staffMember")
    @OrderBy("startTime asc")
    public List<WorkingTime> getWorkingTimes() {
        return workingTimes;
    }

    public void setWorkingTimes(List<WorkingTime> workingTimes) {
        this.workingTimes = workingTimes;
    }
```

The value of the @OrderBy annotation consists of an attribute name of the class followed by asc (ascending) or desc (descending).

To store information about a n:m relationship, a separate table has to be used, as one column cannot store several values (at least if the database schema is in first normal form).
For example if one wanted to extend the example application so that all ingredients of one FoodDrink can be saved and to model the ingredients themselves as entities (e.g. to store additional information about them), this could be modeled as follows (extract of class FoodDrink):

```
private Set<Order> ingredients;

    @ManyToMany
    @JoinTable
    public Set<Ingredient> getIngredients() {
        return ingredients;
    }

    public void setOrders(Set<Ingredient> ingredients) {
        this.ingredients = ingredients;
    }
```

Information about the relation is stored in a table called BILL_ORDER that has to have two columns, one for referencing the Bill, the other one for referencing the Order. Note that the @JoinTable annotation is

not needed in this case because a separate table is the default solution here (same for n:m relations) unless there is a mappedBy element specified.

For 1:n relationships this solution has the disadvantage that more joins (in the database system) are needed to get a Bill with all the Order's it refers to. This might have a negative impact on performance so that the solution to store a reference to the Bill row/entity in the Order's table is probably the better solution in most cases.

Note that bidirectional n:m relationships are not allowed for applications based on the OASP4J. Instead a third entity has to be introduced, which "represents" the relationship (it has two n:1 relationships).

#### 3.1.5.3 Eager vs. Lazy Loading

Using JPA/Hibernate it is possible to use either lazy or eager loading. Eager loading means that for entities retrieved from the database, other entities that are referenced by these entities are also retrieved, whereas lazy loading means that this is only done when they are actually needed, i.e. when the corresponding getter method is invoked.

Application based on the OASP4J must use lazy loading per default. Projects generated with the project generator are already configured so that this is actually the case (this is done in the file NamedQueries.hbm.xml).

For some entities it might be beneficial if eager loading is used. For example if every time a Bill is processed, the Order entities it refers to are needed, eager loading can be used as shown in the following listing:

```
@OneToMany(fetch = FetchType.EAGER)
    @JoinTable
    public Set<Order> getOrders() {
        return orders;
    }
```

This can be done with all four types of relationships (annotations: @OneToOne, @ManyToOne, @OneToMany, @ManyToOne).

#### 3.1.5.4 Cascading Relationships

It is not only possible to specify what happens if an entity is loaded that has some relationship to other entities (see above), but also if an entity is for example persisted or deleted. By default, nothing is done in these situations.
This can be changed by using the cascade element of the annotation that specifies the relation type (@OneToOne, @ManyToOne, @OneToMany, @ManyToOne). For example, if a StaffMember is persisted, all its WorkingTime's should be persisted and if the same applies for deletions (and some other situations, for example if an entity is reloaded from the database, which can be done using the refresh(Object) method of an EntityManager), this can be realized as shown in the following listing (extract of the StaffMember class):

```
    @OneToMany(mappedBy = "staffMember", cascade=CascadeType.ALL)
    public Set<WorkingTime> getWorkingTime() {
        return workingTime;
    }
```

There are several CascadeTypes, e.g. to specify that a "cascading behavior" should only be used if an entity is persisted (CascadeType.PERSIST) or deleted (CascadeType.REMOVE), see here for more information.

### 3.1.6 Embeddable

An embeddable Object is a way to implement relationships between entities, but with a mapping in which both entities are in the same database table. If these entities are often needed together, this is a good way to speed up database operations, as only one access to a table is needed to retrieve both entities.

Suppose the restaurant example application has to be extended in a way that it is possible to store information about the addresses of StaffMember's, this can be done with a new Address class:

```
package de.bund.bva.gastronomy.restaurant.persistence.staff.entity;

...

@Embeddable
public class Address {

    private String street;

    private String number;

    private Integer zipCode;

    private String city;

    @Column(name="STREETNUMBER")
    public String getNumber() {
        return number;
    }

    public void setNumber(String number) {
        this.number = number;
    }

    ...  // other getter and setter methods

    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (!(obj instanceof Address)) {
            return false;
        }
        Address other = (Address) obj;

        return (nullSaveEquals(street, other.street) && nullSaveEquals(number, other.number)
                && nullSaveEquals(city, other.city) && zipCode == other.zipCode);
    }

    private boolean nullSaveEquals(String one, String two) {
        if(one == null && two == null)
            return true;
        if(one == null || two == null)
            return false;
        return one.equals(two);
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1 + zipCode;
        result = prime * result + ((street == null) ? 0 : street.hashCode());
        result = prime * result + ((number == null) ? 0 : number.hashCode());
        result = prime * result + ((city == null) ? 0 : city.hashCode());
        return result;
    }
}
```

This class looks a bit like an entity class, apart from the fact that the @Embeddable annotation is used instead of the @Entity annotation and no primary key is needed here. In addition to that the methods equals(Object) and hashCode() need to be implemented as this is required by Hibernate (it is not required for entities because they can be unambiguously identified by their primary key). For some hints on how to implement the hashCode() method please have a look here.

Using the address in the StaffMember entity class can be done as shown in the following listing:

```
...

@Entity
public class StaffMember implements StaffMemberRo {

    ...

    private Address address;

    ...

    @Embedded
    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }
}
```

The @Embedded annotation needs to be used for embedded attributes. Note that if in all columns in the StaffMember's table that belong to the Address embeddable there are null values, the Address is null when retrieving the StaffMember entity from the database. This has to be considered when implementing the application core to avoid NullPointerException's.

Moreover, if the database tables are created automatically by Hibernate and a primitive data type is used in the embeddable (in the example this would be the case if int is used instead of Integer as data type for the zipCode), there will be a not null constraint on the corresponding column (reason: a primitive data type can never be null in java, so hibernate always introduces a not null constraint). This constraint would be violated if one tries to insert a StaffMember without an Address object (this might be considered as a bug in Hibernate).

Another way to realize the one table mapping are Hibernate UserType's, as described here.

### 3.1.7 Inheritance

Just like normal java classes, entity classes can inherit from others. The only difference is that you need to specify how to map a subtype hierarchy to database tables.

The Java Persistence API (JPA) offers three ways how to do this:

- One table per hierarchy. This table contains all columns needed to store all types of entities in the hierarchy. If a column is not needed for an entity because of its type, there is a null value in this column. An additional column is introduced, which denotes the type of the entity (called "dtype" which is of type varchar and stores the class name).
- One table per subclass. For each concrete entity class there is a table in the database that can store such an entity with all its attributes. An entity is only saved in the table corresponding to its most concrete type. To get all entities of a type that has subtypes, joins are needed.

- One table per subclass: joined subclasses. In this case there is a table for every entity class (this includes abstract classes), which contains all columns needed to store an entity of that class apart from those that are already included in the table of the supertype. Additionally there is a primary key column in every table. To get an entity of a class that is a subclass of another one, joins are needed.

Every of the three approaches has its advantages and drawbacks, which are discussed in detail here. In most cases, the first one should be used, because it is usually the fastest way to do the mapping, as no joins are needed when retrieving entities and persisting a new entity or updating one only affects one table. Moreover it is rather simple and easy to understand.
One major disadvantage is that the first approach could lead to a table with a lot of null values, which might have a negative impact on the database size.

The following listings show how to realize a class hierarchy among entity classes for the class FoodDrink and its subclass Drink:

```
...

@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public abstract class FoodDrink {

    private long id;

    private String description;

    private byte[] picture;

    private long version;


    @Id
    @Column(name = "ID")
    @GeneratedValue(generator = "SEQ_GEN")
    @SequenceGenerator(name = "SEQ_GEN", sequenceName = "SEQ_FOODDRINK")
    public long getId() {
        return this.id;
    }

    public void setId(long id) {
        this.id = id;
    }

    ...
}


...

@Entity
public class Drink extends FoodDrink {

    private boolean alcoholic;

    public boolean isAlcoholic() {
        return alcoholic;
    }

    public void setAlcoholic(boolean alcoholic) {
        this.alcoholic = alcoholic;
    }
}
```

To specify how to map the class hierarchy, the @Inheritance annotation is used. Its element strategy defines which type of mapping is used and can have the following values:

InheritanceType.SINGLE_TABLE (= one table per hierarchy), InheritanceType.TABLE_PER_CLASS (= one table per subclass) and InheritanceType.JOINED (= one table per subclass, joined tables).

Note that the OASP4J advises to avoid deep class hierarchies among entity classes unless they reduce complexity.

### 3.1.8 Testing Entities and DAOs

> **Note**
>
> Review (Jörg) we should link to a testing page here. Further this content is not state-of-the-art anymore.

To test your implementation of entities and DAOs (without GUI and application core), JUnit is well suited. Note that in a Maven project in Eclipse there is usually a source folder dedicated to tests (src/test/java). You should create your tests there in order not to mix them with application code.

The following listing shows an example of how to test StaffMember entities:

```
...

public class StaffMemberEntityTest extends AbstractJpaTests {

  private StaffMemberDao staffMemberDao;

  private static final String LOGIN = "B_Schmidt";
  private static final String FIRST_NAME = "Bob";
  private static final String LAST_NAME = "Schmidt";
  private static final Role ROLE = Role.Barkeeper;

  public void setStaffMemberDao(StaffMemberDao staffMemberDao) {
    this.staffMemberDao = staffMemberDao;
  }

  @Rollback(false)
  public void testPersistEntity() {
    StaffMember staff = new StaffMember();
    staff.setLogin(LOGIN);
    staff.setFirstName(FIRST_NAME);
    staff.setLastName(LAST_NAME);
    staff.setRole(ROLE);

    this.staffMemberDao.speichere(staff);
  }

  @Rollback(false)
  public void testReadEntity() {
        StaffMember staff = this.staffMemberDao.sucheMitId(LOGIN);
        assertEquals(staff.getFirstName(), FIRST_NAME);
        assertEquals(staff.getLastName(), LAST_NAME);
        assertEquals(staff.getRole(), ROLE);
  }

  @Override
  protected String[] getConfigLocations() {
    return new String[] {"resources/spring/application.xml" };
  }
}
```

The class AbstractJpaTest from which the test class is inherited is a helper class that allows to easily load Spring configurations (method: getConfigLocations()). Moreover it is attempted to set every property (everything that has a setter method) with an appropriate bean (staffMemberDao in this case).

The test itself is rather simple. There are two test methods (those method's names have to begin with test). The method testPersistEntity() persists a StaffMember entity. The method testReadEntity() loads this entity from the database and checks that all attributes are set correctly.

By default, each test method is run in the context of a new transaction, which is rolled back when the test method has finished. To commit such a transaction instead, the @Rollback(false) annotation has to be used.

To launch the tests, right click on the test class and select Run as → JUnit Test.

## 3.1.9 Principles

We strongly recommend these principles:

- Use the JPA where ever possible and use vendor (hibernate) specific features only for situations when JPA does not provide a solution. In the latter case consider first if you really need the feature.
- Create your entities as simple POJOs and use JPA to annotate the getters in order to define the mapping.
- Keep your entities simple and avoid putting advanced logic into entity methods.

## 3.1.10 Security

A common security threat is SQL-injection. In order to prevent it you have to strictly follow our rules for queries and prevent building them with string concatenation.

## 3.1.11 Database Configuration

### 3.1.11.1 Configuration Of The Database System

Most of the work that needs to be done to be able to access the database from the application has already been done by the project generator. There are however some setting that you might need to adapt yourself which are described in the following.

One thing you might need to adapt is how to access the database (username, password etc.). This is done in the jpa.properties, which is placed in the following folder:
*project folder*/src/main/resources/config/jpa.properties
It has the follwing content by default:

```
database.url=jdbc:oracle:thin:@localhost:1521:xe
database.username=user01
database.password=pw
database.schema.default=user01
database.connections.max.active=5
```

The database url defines which protocol is used jdbc:oracle:thin), where the database is located (here localhost, if the database system is running on another server, replace this by its IP address), which port is used (1521 by default) and which service name should be used. A service name identifies a database, the only valid value is xe if you are using the express edition (i.e. you only have one database).

The next three parameters set the user, his/her password and the database schema. Set user name and password according to the user you created in the database system. The schema name is usually the same as the user name (does not need to be created explicitly). Furthermore, the maximum number of active connections to the database system can be configured (you might need to increase this value later on).

One of the properties specified for this bean is hibernate.hbm2ddl.auto which has the following possible values:

- create
- create-drop
- update
- validate
- none

If create is set, database tables needed to store the ./glossary_persistence_entitiy.html[entities] are created automatically by Hibernate. Any existing data will be deleted (and that is why this option is not suitable for a productive system).
With the value create-drop, the same is done but the tables are dropped after the session ended.
update keeps existing tables and updates them according to the changes made to the entities.
If validate is set, the tables have to be created manually but are inspected weather they fit to the definitions of the entities or not.
none means that nothing is done (might cause exceptions later on if the table definitions do not fit to the entities).

If you have chosen to create the tables automatically and want to see their definitions, open the SQL command line, log in as the user that the tables were created with (not as system!) and type the following commands:

```
set long 1000
set pagesize 0
select DBMS_METADATA.GET_DDL('TABLE','Tabelname') from DUAL;
```

Tablename has to be replaced with the actual table name (in upper case letters). To get a list of all table names created by the current user, type the following:

```
SELECT table_name FROM user_tables;
```

Another property is hibernate.show_sql, which can be set to true to see the SQL statements generated by Hibernate on the console output. This often proves beneficial to debug an application.

Several other properties are set to configure the entityManagerFactory in the jpa.xml, but none them is really interesting in most cases so they will not be discussed here.

The entityManagerFactory uses the bean appDataSource to get a connection to the database system. It is a connection pool, i.e. it maintains several connections so that a process willing to access the database system can retrieve a connection from that pool and does not have to open a new one, which usually takes some time. Most of the parameters used are taken from the jpa.properties file.

Spring supports transactions for which the next bean called transactionManager is used. The entityManagerFactory is passed to it as a parameter. The next element in the jpa.xml specifies that transaction control is done via annotations.

By using a bean of class PersistenceExceptionTranslationPostProcessor all exceptions that occur in the context of a DAO are converted to other exceptions which are easier to understand, due to a problem with the EntityManager, as long as the DAO has a @Repository annotation.

The entityManagerFactoryBean creates EntityManager instances using the entityManagerFactory.

As you can see, the entityManager has to be set (note that the entityManagerFactoryBean will not be passed directly as a parameter but an EntityManager obtained from the entityManagerFactoryBean, see here for further information).

> **Note**
>
> Review (joedoehh) i agree with the principle to use jpa when possible and only use jpa provider features that are not jpa conform as last ressort. also we have to start somewhere and need examples, so in principle it is ok to choose hibernate as a standard provider for examples. that said i feel that too much provider specific details leak through. like when it is said that queries should be placed in a hibernate mapping file: which is confusing cause even with hibernate these days i am happy not to use mapping files any longer - and i dont want to get them back through the backdoor just as place to put my quries. its not easy, but i think we need to find a clear structure to seperate standard parts (JPA) from technical details concerning the choosen jpa provider. in result the text given here is more easy applicable in a non-hibernate environment (there are lots of it) and we have one place where to add details to show how things are done in using a different provider say openejb.

## 3.2 Logic Layer

The logic layer is the heart of the application and contains the main business logic. According to our business architecture we divide an application into *business components*. The *component part* assigned to the logic layer contains the functional use-cases the business component is responsible for. For further understanding consult the application architecture.

### 3.2.1 Use Case

For each general business operation we create a *use case*. In the code we use the prefix Uc for all use cases.

First we create an interface Uc<MyUseCase> that contains the method(s) with the business operation documented with JavaDoc. Typically there is only a single method per use case. If there are different variants of a business operation, you define multiple methods in the same use case interface. The API of the use cases has to be business oriented. This means that all parameters and return types of a use case method have to be business objects, datatypes (String, Integer, MyCustomerNumber, etc.), or collections of these. The API may not access objects from other business components not in the (transitive) dependencies of the declaring business component. Here is an example of a use case interface:

```
package org.oasp.gastronomy.restaurant.staffmanagement.logic.api;

/**
 * Interface for the use-case to find {@link StaffMemberDto staff members}.
 */
public interface UcFindStaffMember {
  /**
   * @param login The {@link StaffMemberDto#getLogin() login} of the requested staff member.
   * @return The {@link StaffMemberDto} with the given <code>login</code> or <code>null</code>
   *         if no such object exists.
   */
  StaffMemberBo getStaffMember(String login);
}
```

The implementation of the use case is named Uc<MyUseCase>Impl. It typically needs access to the persistent data. This is done by injecting the corresponding DAO. For the principle *data sovereignty* only DAOs of the same business component may be accessed directly from the use case. For accessing data from other components the use case has to use the corresponding logic layer API (TODO explain better). Further it may not expose persistent entities from the persistence layer and has to map them to transfer objects.

Within the different **Use Cases** entities are mapped via a BeanMapper to persistent entities. Let's take a quick look at the Use Case FindStaffMember:

```
package org.oasp.gastronomy.restaurant.staffmanagement.logic.impl;

public class UcFindStaffMemberImpl extends AbstractStaffMemberUc implements UcFindStaffMember {

  @Override
  public StaffMemberBo getStaffMember(String login) {

    return getBeanMapper().map(this.staffMemberDao.searchByLogin(login), StaffMemberBo.class);
  }

  @Override
  public List<StaffMemberBo> getAllStaffMember() {

    List<StaffMember> members = this.staffMemberDao.getAllStaffMembers();
    List<StaffMemberBo> membersBo = new ArrayList<>();
```

```
    for (StaffMember member : members) {
      membersBo.add(getBeanMapper().map(member, StaffMemberBo.class));
    }

    return membersBo;
  }
}
```

As you can see, provided entities are detached and mapped to corresponding business objects (here StaffMemberBo.class). These business objects are simple POJOs (Plain Old Java Objects) and stored in:
<package-name-prefix>.<domain>.<application-name>.<component>.api.
The mapping process of these entities and the declaration of the AbstractLayerImpl class are described here. For every business object there has to be a mapping entry in the src/main/resources/config/app/common/dozer-mapping.xml file. For example, the mapping entry of a TableBo to a Table looks like this:

```
  <mapping>
    <class-a>org.oasp.gastronomy.restaurant.tablemanagement.logic.api.TableBo</class-a>
    <class-b>org.oasp.gastronomy.restaurant.tablemanagement.persistence.api.entity.Table</class-b>
  </mapping>
```

Testing the component (interface) can be done in the same way that the DAOs are tested, see here how this is done.

### 3.2.2 Component Interface

A component may consist of several Use Cases but is only accessed by the next higher layer or other components through one interface, i.e. by using one Spring bean. The task of this bean is to delegate the invocations to the respective Use Cases. The only exception is, that for the basic data sub applications, the DAOs are accessed directly.

The following listing shows how to implement the component interface for staffmanagement:

```
package org.oasp.gastronomy.restaurant.staffmanagement.logic.impl;

public class StaffManagementImpl extends AbstractLayerImpl implements StaffManagement {

    private UcFindStaffMember ucFindStaffMember;

    private UcManageStaffMember ucManageStaffMember;

    // ... The setter go here

    @Override
    public StaffMemberBO getStaffMemberBO(String login) {
        return this.ucFindStaffMember.getStaffMember(login);
    }

    @Override
    public List<StaffMemberBO> getAllStaffMembers() {
        return this.ucFindStaffMember.getAllStaffMember();
    }

    @Override
    public void updateStaffMember(StaffMemberBo staffMember) throws ValidationException {
        this.ucManageStaffMember.updateStaffMember(staffMember);
    }

    @Override
    public void deleteStaffMember(String login) {
        this.ucManageStaffMember.deleteStaffMember(login);
    }
}
```

Similar to DAOs there is an interface for every component interface implementation in the package of the component (e.g. org.oasp.gastronomy.restaurant.staffmanagement.logic.api or org.oasp.gastronomy.restaurant.salesmanagement.logic.api), which contains all public methods. As shown above, all entities, that pass that interface, are redirected to the corresponding Use Cases where the actual mapping takes action.

#### 3.2.2.1 Passing Parameters Among Components

Entities have to be detached for the reasons of data ownership, if entities are passed among components or to the usage layer (as already mentioned here). The detachment is done by mapping this entity in the components interface with Dozer. The corresponding java POJO in the component layer has the same attributes as the entity that is persisted. The packages are:

```
Persistence Entities: <package-name-prefix>.<domain>.<application-
name>.<component>.persistence.api.entity
BusinessObjects(BOs): <package-name-prefix>.<domain>.<application-name>.<component>.logic.api
```

This mapping is a simple copy process. So changes out of the scope of the owning component to any BO do not directly affect the persistent entity. The create()-method to create a new StaffMember is only used from within the sub application (e.g. in the GUI-area), whereas getStaffMember() for example is also used by tablemanagement.

Another part within the application, where the mapping is necessary, is the service component. The service component provides other applications the possibility to access Use Cases implemented in the logical layer as well as transferring data. The data transfer has to be wrapped in special Transfer Objects (TO), that are offered to other applications via the service interface.

Spring has to be configured to set the property dozer correctly.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
                      http://www.springframework.org/schema/beans/spring-beans.xsd
                      http://www.springframework.org/schema/context
                      http://www.springframework.org/schema/context/spring-context.xsd">

  <context:component-scan base-package="org.oasp.module.beanmapping"/>
  <bean id="dozer" class="org.dozer.DozerBeanMapper" scope="singleton">
    <property name="mappingFiles">
      <list>
        <value>config/app/common/dozer-mapping.xml</value>
      </list>
    </property>
  </bean>

</beans>
```

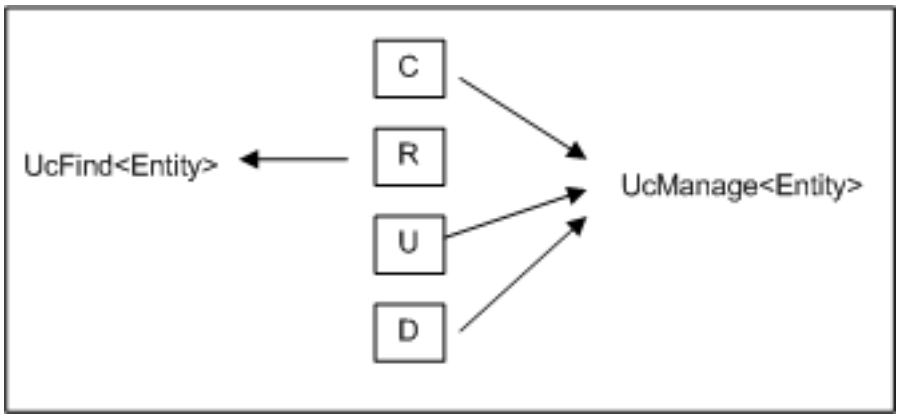
Dozer has been configured as Spring bean in the file src/main/resources/config/app/common/beans-dozer.xml.

#### 3.2.2.2 Use Case Example

The CRUD (Create, Read, Update, Delete) functionality is a basic Use Case that has to be implemented for each component and usually for each entity managed by that component. This Use Case is split for every entity in the component logical layer.

- UcFind<entity> provides methods for getting at least one entity from the database

- UcManage<entity> provides methods for managing the entity. At least, create-, update- and delete-functionalities are provided by that class.




The Use Cases are structured in the logical layer and in the components as follows:

As the graphic above illustrates, the necessary DAO entity to access the database is provides by an abstract class. Use Cases that need access to this DAO entity, have to extend that abstract class. Needed dependencies (in this case the staffMemberDao) are resolved by Spring, see here. For the validation (e.g. to check if all needed attributes of the StaffMember have been set) either Java code or Drools, a business rule management system, can be used.

## 3.3 Service Layer

The service layer is responsible to expose functionality of the logical layer to external consumers over a network. It is responsible for the following aspects:

- transaction control
- authorization
- transformation of functionality to technical protocols

> **Note**
>
> (JH) When using the enterprise architecture, all REST and SOAP services are deployed to the service-gateway and not part of the application. HTTP-Invoker fixes as EAM protocol? How about pure REST with single domain portal or Content Security Policy?

### 3.3.1 Types of Services

If you want to create a service please distinguish the following types of services:

- **External Services**
  are used for communication between different companies, vendors, or partners.
- **Internal Services**
  are used for communication between different applications in the same application landscape of the same vendor.
  - **Back-end Services**
    are internal services between Java back-ends typically with different release and deployment cycles (otherwise if not Java consider this as external service).
  - **JS-Client Services**
    are internal services provided by the Java back-end for JavaScript clients (GUI).
  - **Java-Client Services**
    are internal services provided by the Java back-end for for a native Java client (JavaFx, EclipseRcp, etc.).

The choices for technology and protocols will depend on the type of service. Therefore the following table gives a guideline for aspects according to the service types. These aspects are described below.

*Table 3.1. Aspects according to service-type*

| Aspect | External Service | Back-end Service | JS-Client Service | Java-Client Service |
|---|---|---|---|---|
| **versioning** | required | required | not required | not required |
| **interoperability** | mandatory | not required | implicit | not required |
| recommended **protocol** | SOAP or REST | HTTP-Invoker | REST+JSON | HTTP-Invoker |

### 3.3.2 Versioning

For services consumed by other applications we use versioning to prevent incompatibilities between applications when deploying updates. This is done by the following conventions:

- We define a two digit version number separated by underscore and prefixed with v for version (e.g. v1_0).
- We use the version number as part of the Java package defining the service API (e.g. com.foo.application.component.service.api.v1_0)
- We use the version number as part of the service name in the remote URL (e.g. https://application.foo.com/services/ws/component/MyService_v1_0)
- Whenever we need to change the API of a service we create a new version (e.g. v1_1) as an isolated copy of the previous version of the service. In the implementation of different versions of the same service we can place compatibility code and delegate to the same unversioned use-case of the logic layer whenever possible.
- For maintenance and simplicity we avoid keeping more than one previous version.

### 3.3.3 Interoperability

For services that are consumed by clients with different technology *interoperability* is required. This is addressed by selecting the right protocol following protocol-specific best practices and following our considerations especially *simplicity*.

### 3.3.4 Protocol

For services there are different protocols. Those relevant for and recommended by OASP4J are listed in the following sections with examples how to implement them in Java.

#### 3.3.4.1 SOAP

SOAP is a common protocol that is rather complex and heavy. It allows to build inter-operable and well specified services (see WSDL). SOAP is transport neutral what is not only an advantage. We strongly recommend to use HTTPS transport and ignore additional complex standards like WS-Security and use established HTTP-Standards such as RFC2617 (and RFC5280).

##### JAX-WS

For building web-services with Java we use the JAX-WS standard. TODO: There are two approaches

- code first
- contract first

Here is an example in case you define a code-first service. We define a regular interface to define the API of the service and annotate it with JAX-WS annotations:

```
@WebService
public interface HelloWorldWebService extends HelloWorldService {

  @WebMethod
```

```
  @WebResult(name = "message")
  String sayHi(@WebParam(name = "name") String name);

}
```

And here is a simple implementation of the service:

```
@Named("HelloWorldWebService")
@WebService(endpointInterface =
 "org.oasp.gastronomy.restaurant.tablemanagement.service.api.ws.HelloWorldWebService")
public class HelloWorldWebServiceImpl implements HelloWorldWebService {

  @Override
  public String sayHi(String name) {

    return "Hello " + name;
  }
}
```

Finally we have to register our service implementation in the spring configuration file beans-service.xml:

```
  <jaxws:endpoint id="helloWorld" implementor="#HelloWorldWebService" address="/ws/v1_0/HelloWorld"/>
```

The implementor attribute references an existing bean with the ID HelloWorldWebService that corresponds to the @Named annotation of our implementation (see dependency injection guide). The address attribute defines the URL path of the service.

#### SOAP Custom Mapping

In order to map custom datatypes or other types that do not follow the Java bean conventions, you need to TODO

REVIEW: Both SOAP and REST+XML will be done based on JAXB. So this is technically all about JAXB. Should we write a separate guide for JAXB and place references?

#### SOAP Testing

For testing SOAP services in general consult the testing guide.

For testing SOAP services manually we strongly recommend SoapUI.

### 3.3.4.2 REST

REST is an inter-operable protocol that is more lightweight than SOAP. However, it is no real standard and can cause confusion. Therefore we define best practices here to guide you. For a general introduction consult the wikipedia. REST services are called via HTTP(S) URIs. We distinguish between **collection** and **element** URIs:

- A collection URI is build from the rest service URI by appending the name of a collection. This is typically the name of a persistence entity. Such URI identifies the entire collection of all elements of this type. Example: https://mydomain.com/myapp/services/rest/v1_0/MyComponent/MyEntity (TODO: others suggest plural but in the end this is only causing trouble as some terms have to proper plural form)
- An element URI is build from a collection URI by appending an element ID. It identifies a single element (entity) within the collection. Example: https://mydomain.com/myapp/services/rest/v1_0/MyComponent/MyEntity/42

The following table specifies how to use the HTTP methods (verbs) for collection and element URIs properly (see wikipedia).

*Table 3.2. Usage of HTTP methods*

| HTTP Method | Meaning (Element URI) | Meaning (Collection URI) |
|---|---|---|
| GET | Read element | Read all elements (typically using paging and hit limit to prevent loading too much data) |
| PUT | Replace element | Replace entire collection (typically not supported) |
| POST | Not supported | Create a new element in the collection |
| DELETE | Delete element | Delete entire collection (typically not supported) |

#### JAX-RS

For implementing REST services we use the JAX-RS standard. As an implementation we recommend CXF. For JSON bindings we use Jackson while XML binding works out-of-the-box with JAXB. To implement a service you simply write a regular class and use JAX-RS annotations to annotate methods that shall be exposed as REST operations. Here is a simple example:

```
@Path("/tablemanagement")
@Named("TableManagementRestService")
@Transactional
public class TableManagementRestServiceImpl implements RestService {
  // ...
  @Produces(MediaType.APPLICATION_JSON)
  @GET
  @Path("/table/{id}/")
  @RolesAllowed(PermissionConstant.GET_TABLES)
  public TableBo getTable(@PathParam("id") String id) throws RestServiceException {

    Long idAsLong;
    if (id == null)
      throw new BadRequestException("missing id");
    try {
      idAsLong = Long.parseLong(id);
    } catch (NumberFormatException e) {
      throw new RestServiceException("id is not a number");
    } catch (NotFoundException e) {
      throw new RestServiceException("table not found");
    }
    return this.tableManagement.getTable(idAsLong);
  }
  // ...
}
```

Here we can see a REST service for the business component tablemanagement. The method getTable can be accessed via HTTP GET (see @GET) under the URL path tablemanagement/table/{id} (see @Path annotations) where {id} is the ID of the requested table and will be extracted from the URL and provided as parameter id to the method getTable. It will return its result (TableBo) as JSON (see @Produces). As you can see it delegates to the logic component tableManagement that contains the actual business logic while the service itself only contains mapping code and general input validation. Further you can see the @Transactional annotation for transaction handling and @RolesAllowed for security. The REST service implementation is a regular CDI bean that can use dependency injection.

> **Note**
>
> With JAX-RS it is important to make sure that each service method is annotated with the proper HTTP method (@GET,@POST,etc.) to avoid unnecessary debugging. So you should take care not to forget to specify one of these annotations.

All your services have to be declared in the beans-service.xml file. For the example this would look as following:

```
<jaxrs:server id="CxfRestServices" address="/rest">
  <!-- ... -->
  <jaxrs:serviceBeans>
    <ref bean="TableManagementRestService"/>
    <!-- ... -->
  </jaxrs:serviceBeans>
</jaxrs:server>
```

Here TableManagementRestService is the identifier used in the @Named annotation of the REST service implementation (see example above).

#### HTTP Status Codes

Further we define how to use the HTTP status codes for REST services properly. In general the 4xx codes correspond to an error on the client side and the 5xx codes to an error on the server side.

*Table 3.3. Usage of HTTP status codes*

| HTTP Code | Meaning | Response | Comment |
|---|---|---|---|
| 200 | OK | requested result | Result of successful GET |
| 204 | No Content | *none* | Result of successful POST, DELETE, or PUT (void return) |
| 400 | Bad Request | error details | The HTTP request is invalid (parse error, validation failed) |
| 401 | Unauthorized | *none* (security) | Authentication failed |
| 403 | Forbidden | *none* (security) | Authorization failed (TODO also reply 401?) |
| 404 | Not found | *none* | Either the service URL is wrong or the requested resource does not exist |
| 500 | Server Error | error code, UUID | Internal server error occurred (used for all technical exceptions) |

For more details about REST service design please consult the RESTful cookbook.

#### REST Exception Handling

For exceptions a service needs to have an exception facade that catches all exceptions and handles them by writing proper log messages and mapping them to a HTTP response with an according HTTP status code. Therefore the OASP provides a generic solution via RestServiceExceptionFacade. You need to follow the exception guide so that it works out of the box because the facade needs to be able to distinguish between business and technical exceptions. You need to configure it in your beans-service.xml as following:

```
  <jaxrs:server id="CxfRestServices" address="/rest">
    <jaxrs:providers>
      <bean class="org.oasp.module.rest.service.impl.RestServiceExceptionFacade"/>
      <!-- ... -->
    </jaxrs:providers>
    <!-- ... -->
  </jaxrs:server>
```

Now your service may throw exceptions but the facade with automatically handle them for you.

#### REST Media Types

The payload of a REST service can be in any format as REST by itself does not specify this. The most established ones that the OASP recommends are XML and JSON. Follow these links for further details and guidance how to use them properly. JAX-RS and CXF properly support these formats (MediaType.APPLICATION_JSON and MediaType.APPLICATION_XML can be specified for @Produces or @Consumes). Try to decide for a single format for all services if possible and NEVER mix different formats in a service.

In order to use JSON via Jackson with CXF you need to register the factory in your beans-service.xml and make CXF use it as following:

```
  <jaxrs:server id="CxfRestServices" address="/rest">
    <jaxrs:providers>
      <bean class="org.codehaus.jackson.jaxrs.JacksonJsonProvider">
        <property name="mapper">
          <ref bean="ObjectMapperFactory"/>
        </property>
      </bean>
      <!-- ... -->
    </jaxrs:providers>
    <!-- ... -->
  </jaxrs:server>

  <bean id="ObjectMapperFactory" factory-bean="RestaurantObjectMapperFactory" factory-
method="createInstance"/>
```

#### REST Testing

For testing REST services in general consult the testing guide.

For manual testing REST services there are browser plugins:

- FF: poster
- Chrome: postman (advanced-rest-client)

### 3.3.4.3 HTTP-Invoker

TODO

### 3.3.5 Service Considerations

The term *service* is quite generic and therefore easily misunderstood. It is a unit exposing coherent functionality via a well-defined interface over a network. For the design of a service we consider the following aspects:

- **self-contained**
  The entire API of the service shall be self-contained and have no dependencies on other parts of the application (other services, implementations, etc.).
- **idem-potent**
  E.g. creation of the same master-data entity has no effect (no error)
- **loosely coupled**
  Service consumers have minimum knowledge and dependencies on the service provider.
- **normalized**
  complete, no redundancy, minimal
- **coarse-grained**
  Service provides rather large operations (save entire entity or set of entities rather than individual attributes)
- **atomic**
  Process individual entities (for processing large sets of data use a batch instead of a service)
- **simplicity**
  avoid polymorphism, RPC methods with unique name per signature and no overloading, avoid attachments (consider separate download service), etc.

## 3.4 Client Layer

There are various technical approaches to build GUI clients. In general we have to distinguish the following types of clients:

- web clients
- native desktop clients
- native mobile clients

### 3.4.1 Web Clients

Currently we focus on building web-clients. And so far we offer a Java Script based client provided by OASP4JS.

### 3.4.2 Native Desktop Clients

Currently not addressed. TODO, JavaFx, EclipseRCP

### 3.4.3 Native Mobile Clients

Currently not addressed. Dependent on target mobile platform. Android may be addressed due to Java. Alternative via Cordova as Web Client.

## 3.5 Batches

TODO

A Batch is realized as standalone application using JSR352 (recommended via spring-batch). TODO best practices, example

# 4. Guides

## 4.1 Logging

We use SLF4J as API for logging. The recommended implementation is Logback for which we provide additional value such as configuration templates and an appender that prevents log-forging and reformatting of stack-traces for operational optimizations.

### 4.1.1 Usage

#### 4.1.1.1 Maven Integration

In the pom.xml of your application add this dependency (that also adds transitive dependencies to SLF4J and logback):

```
<dependency>
  <groupId>org.oasp.java</groupId>
  <artifactId>oasp4j-logging</artifactId>
  <version>1.0.0</version>
</dependency>
```

#### 4.1.1.2 Configuration

The configuration file is logback.xml and is to put in the directory src/main/resources of your main application. For details consult the logback configuration manual. OASP4J provides a production ready configuration here. Please use copy this configuration into your application as previously described in order to benefit from the provided operational and aspects. We do not include the configuration into the oasp4j-logging module to give you the freedom of customizations (e.g. tune log levels for components and integrated products and libraries of your application).

#### 4.1.1.3 Logger Access

The general pattern for accessing loggers from your code is a static logger instance per class. We pre-configured the development environment so you you can just type LOG and hit [ctrl][space] to insert the code pattern line into your class:

```
public class MyClass {
  private static final Logger LOG = LoggerFactory.getLogger(MyClass.class);
  ...
}
```

#### 4.1.1.4 How to log

We use a common understanding of the log-levels as illustrated by the following table. This helps for better maintenance and operation of the systems by combining both views.

*Table 4.1. Loglevels*

| Loglevel | Description | Impact | Active Environments |
|---|---|---|---|
| FATAL | Only used for fatal errors that prevent the application to work at all (e.g. startup fails | Operator has to react immediately | all |

| Loglevel | Description | Impact | Active Environments |
|---|---|---|---|
| | or shutdown/restart required) | | |
| ERROR | An abnormal error indicating that the processing failed due to technical problems. | Operator should check for known issue and otherwise inform development | all |
| WARNING | A situation where something worked not as expected. E.g. a business exception or user validation failure occurred. | No direct reaction required. Used for problem analysis. | all |
| INFO | Important information such as context, duration, success/ failure of request or process | No direct reaction required. Used for analysis. | all |
| DEBUG | Development information that provides additional context for debugging problems. | No direct reaction required. Used for analysis. | development and testing |
| TRACE | Like DEBUG but exhaustive information and for code that is run very frequently. Will typically cause large log-files. | No direct reaction required. Used for problem analysis. | none (turned off by default) |

Exceptions (with their stacktrace) should only be logged on FATAL or ERROR level. For business exceptions typically a WARNING including the message of the exception is sufficient.

## 4.1.2 Operations

#### 4.1.2.1 Log Files

We always use the following log files:

- **Error Log**: Includes log entries to detect errors.
- **Info Log**: Used to analyze system status and to detect bottlenecks.
- **Debug Log**: Detailed information for error detection.

The log file name pattern is as follows:

```
<LOGTYPE>_log_<HOST>_<APPLICATION>_<TIMESTAMP>.log
```

*Table 4.2. Segments of Logfilename*

| Element | Value | Description |
|---|---|---|
| <LOGTYPE> | info, error, debug | Type of log file |
| <HOST> | e.g. mywebserver01 | Name of server, where logs are generated |
| <APPLICATION> | e.g. myapp | Name of application, which causes logs |
| <TIMESTAMP> | YYYY-MM-DD_HH00 | date of log file |

Example: error_log_mywebserver01_myapp_2013-09-16_0900.log

Error log from mywebserver01 at application myapp at 16th September 2013 9pm.

#### 4.1.2.2 Output format

We use the following output format for all log entries to ensure that searching and filtering of log entries work consistent for all logfiles:

```
[D: <timestamp>] [P: <priority (Level)>] [C: <NDC>][T: <thread>][L: <logger name>]-[M: <message>]
```

- **D**: Date ( ISO8601: 2013-09-05 16:40:36,464)
- **P**: Priority (the log level)
- **C**: Correlation ID (ID to identify users across multiple systems, needed when application is distributed)
- **T**: Thread (Name of thread)
- **L**: Logger name (use class name)
- **M**: Message (log message)

Example:

```
[D: 2013-09-05 16:40:36,464] [P: DEBUG] [C: 12345] [T: main] [L: my.package.MyClass]-[M: My message...]
```

### 4.1.3 Security

In order to prevent log forging attacks we provide a special appender for logback in oasp4j-logging. If you use it (see ) you are safe from such attacks.

# 4.2 Security

Security is todays most important cross-cutting concern of an application and an enterprise IT-landscape. We seriously care about security and give you detailed guides to prevent pitfalls, vulnerabilities, and other disasters. While many mistakes can be avoided by following our guidelines you still have to consider security and think about it in your design and implementation.

## 4.2.1 Authentication

oasp4j uses Spring Security as a framework for authentication purposes. Therefore you need to define an authentication provider implementing the org.springframework.security.authentication.AuthenticationProvider interface from Spring Security. The implemented authentication provider can be registered as main authentication provider using the authentication-manager declaration.

```
<beans:beans xmlns="http://www.springframework.org/schema/security" xmlns:beans="http://
www.springframework.org/schema/beans">

  <beans:bean id="restaurantAuthenticationProvider"

  class="org.oasp.gastronomy.restaurant.general.common.api.security.ServletAuthenticationProvider"/>

  <authentication-manager alias="restaurantAuthenticationManager" erase-credentials="false">
    <authentication-provider ref="restaurantAuthenticationProvider"/>
  </authentication-manager>
</beans:beans>
```

### 4.2.1.1 Mechanisms

#### Basic

Http-Basic authentication can be easily implemented with this configuration:

```
<http auto-config="true" use-expressions="true">
  ...
  <http-basic/>
  ...
</http>
```

#### Form Login

For a form login the spring security implementation might look like this:

```
<http auto-config="false" use-expressions="true">
    ...
    <form-login login-page="/login" authentication-failure-url="/login?authentication_failed=1"
      login-processing-url="/j_spring_security_login" default-target-url="/services"/>
    <logout logout-url="/j_spring_security_logout" logout-success-url="/login?logout=1" invalidate-
session="true"/>
    <access-denied-handler error-page="/login?access_denied=1"/>
    ...
</http>
```

The interesting part is, that there is a login-processing-url, which should be adressed to handle the internal spring security authentication and similarly there is a logout-url, which has to be called to logout a user.

#### Preserve original request anchors after form login redirect

Spring Security will automatically redirect any unauthorized access to the defined login-page. After sucuessful login, the user will be redirected to the original requested URL. The only pitfall is, that anchors

in the request URL will not be transmitted to server and thus cannot be restored after successful login. Therefore the oasp4j-security module provides the RetainAnchorFilter, which is able to inject javascript code to the source page and to the target page of any redirection. Using javascript this filter is able to retrieve the requested anchors and store them into a cookie. Heading the target URL this cookie will be used to restore the original anchors again.

To enable this mechanism you have to integrate the RetainAnchorFilter as follows: First, declare the filter with

- storeUrlPattern: an regular expression matching the URL, where anchors should be stored
- restoreUrlPattern: an regular expression matching the URL, where anchors should be restored
- cookieName: the name of the cookie to save the anchors in the intermediate time

```
<beans:bean id="retainAnchorFilter" class="org.oasp.module.security.common.web.api.RetainAnchorFilter">
    <!-- first [^/]+ part describes host name and possibly port, second [^/]+ is the application name --
>
    <beans:property name="storeUrlPattern" value="http://[^/]+/[^/]+/login.*"/>
    <beans:property name="restoreUrlPattern" value="http://[^/]+/[^/]+/.*"/>
    <beans:property name="cookieName" value="TARGETANCHOR"/>
</beans:bean>
```

Second, register the filter as first filter in the request filter chain. You might want to use the before="FIRST" or after="FIRST" attribute if you have multiple request filters, which should be run before the default filters.

```
<http auto-config="false" use-expressions="true">
    <custom-filter ref="retainAnchorFilter" after="FIRST"/>
</http>
```

#### 4.2.1.2 Users vs. Systems

TODO

### 4.2.2 Authorization

TODO

#### 4.2.2.1 Access Management

The access management is enabled by url patterns, the requested will be matched against. The order of these url patterns is essential as the first matching pattern will declare the access restriction for the incoming request (see access attribute). Here an example:

```
<http auto-config="false" use-expressions="true">
    <intercept-url pattern="/" access="isAnonymous()"/>
    <intercept-url pattern="/index.jsp" access="isAnonymous()"/>
    <intercept-url pattern="/login*" access="isAnonymous()"/>
    <intercept-url pattern="/j_spring_security_login*" access="isAnonymous()"/>
    <intercept-url pattern="/j_spring_security_logout*" access="isAnonymous()"/>
    <intercept-url pattern="/**" access="isAuthenticated()"/>
    ...
</http>
```

#### 4.2.2.2 Method Authorization

For authorization purposes we also use Spring Security. We will use the jsr250 annotation @RolesAllowed for authorizing method calls against the permissions defined in the annotation body. The jsr250 has to be enabled using the similarly named attribute of global-method-security. Custom

authorization managers could be registered using the access-decision-manager-ref in the global-method-security. Here an example:

```
<global-method-security pre-post-annotations="enabled" secured-annotations="enabled"
    jsr250-annotations="enabled" access-decision-manager-ref="restaurantAuthorizationManager">
</global-method-security>
```

Furthermore the oasp4j-security module provides a simple and efficient way to define permissions and roles. The files will be declared in a so called access control schema which currently might look like this:

```
<security xmlns="http://oasp.org/security">
  <roles>
    <role name="Waiter">
      <permission name="TABLEMANAGEMENT_GET_TABLES"/>
      <permission name="TABLEMANAGEMENT_CHANGE_TABLE"/>
    </role>
    <role name="Chief">
      <include ref="Waiter"/>
      <permission name="TABLEMANAGEMENT_ADD_TABLE"/>
      <permission name="TABLEMANAGEMENT_REMOVE_TABLE"/>
    </role>
  </roles>
</security>
```

There are two roles---a Waiter and a Chief. The waiter has the permissions for getting tables and changing tables in the tablemanagment component. The chief has all permissions from the waiter and also additional ones like adding tables and removing tables in the tablemanagement. The oasp4j-security module automatically detects invalid configurations which might implement include cycles. Having such case, it will throw an exception.

Furthermore the oasp4j-module provides the infrastructure to manage the access control schema. Therefore the developer has to instantiate the RoleAuthorizationProvider from the module, which takes the application access control schema as constructor argument. As the interface of the RoleAuthorizationProvider takes a user token (object) and a list of permissions to be evaluated, the RoleAuthorizationProvider needs an additional RoleProvider, which does the mapping of user tokens to the roles a user has. A configuration of such an authorization manager might look like this:

```
<beans:bean id="restaurantAuthorizationManager"

  class="org.oasp.gastronomy.restaurant.general.common.api.security.RestaurantAuthorizationManager">
    <beans:property name="roleAuthorizationProvider" ref="oaspRoleAuthorizationProvider"/>
</beans:bean>

<beans:bean id="oaspRoleAuthorizationProvider"
            class="org.oasp.module.security.common.authorization.api.RoleAuthorizationProvider">
    <beans:constructor-arg>
      <beans:value>classpath:/config/app/security/accessControlSchema.xml</beans:value>
    </beans:constructor-arg>
    <beans:property name="rolesProvider">

  <beans:bean class="org.oasp.gastronomy.restaurant.general.common.api.security.ServletRoleProvider"/>
    </beans:property>
</beans:bean>
```

#### 4.2.2.3 Check Role based Permissions

We will use the jsr250 annotation @RolesAllowed for authorizing method calls against the permissions defined in the annotation body.

#### 4.2.2.4 Check Data based Permissions

TODO

### 4.2.3 Vulnerability

Independent from classical authentication and authorization mechanisms there are many common pitfalls that can lead to vulnerabilities and security issues in your application such as XSS, CSRF, SQL-injection, log-forging, etc. A good source of information about this is the OWASP. We address these common threats individually in *security* sections of our technological guides as a concrete solution to prevent an attack typically depends on the according technology. So to prevent SQL-injection follow our persistence-layer guide, for log-forging see our logging guide, etc.

https://www.aspectsecurity.com/research/aspsec_presentations/download-bypassing-web-authentication-and-authorization-with-http-verb-tampering/

Tool For Web Application Vulnerabilities : OWASP Zed Attack Proxy Project

1. Easy to Install
2. Supports Different types of Fuzzer Based Tests
3. Details Results Reports
4. Convenient to carry out Test on Staging environment

## 4.3 Dependency Injection

Dependency injection is one of the most important design patterns and is a key principle to a modular and component based architecture. The Java Standard for dependency injection is javax.inject (JSR330) that we use in combination with JSR250.

There are many frameworks which support this standard including all recent JEE application servers. We recommend to use Spring (a.k.a. springframework) that we use in our example application. However, the modules we provide typically just rely on JSR330 and can be used with any compliant container.

### 4.3.1 Example Bean

Here you can see the implementation of an example bean using JSR330 and JSR250:

```
@Named
public class MyBeanImpl implements MyBean {
  private MyOtherBean myOtherBean;
  @Inject
  public void setMyOtherBean(MyOtherBean myOtherBean) {
    this.myOtherBean = myOtherBean;
  }
  @PostConstruct
  public void init() {
    // initialization if required (otherwise omit this method)
  }
  @PreDestroy
  public void dispose() {
    // shutdown bean, free resources if required (otherwise omit this method)
  }
}
```

It depends on MyOtherBean that should be the interface of an other component that is injected into the setter because of the @Inject annotation. To make this work there must be exactly one implementation of MyOtherBean in the container (in our case spring). In order to put a Bean into the container we use the @Named annotation so in our example we put MyBeanImpl into the container. Therefore it can be injected into all setters that take the interface MyBean as argument and are annotated with @Inject. To make spring find all your beans annotated with @Named in the package com.mypackage.example and its sub-packages you use the following element in your spring XML configuration:

```
<context:component-scan base-package="com.mypackage.example"/>
```

In some situations you may have an Interface that defines a kind of "plugin" where you can have multiple implementations in your container and want to have all of them. Then you can request a list with all instances of that interface as in the following example:

```
  @Inject
  public void setConverters(List<MyConverter> converters) {
    this.converters = converters;
  }
```

### 4.3.2 Spring Usage and Conventions

Spring is an awesome framework that we highly recommend. However it has a long history and therefore offers many different ways to archive the same goals while some of them might lead you on the wrong track. The OASP4J helps you to do things right and defines conventions that give your development teams orientation.

#### 4.3.2.1 Spring XML Files

Besides JSR330 it is sometimes necessary to use spring XML files in order to configure specific aspects. These files should be named beans-*.xml and located under src/main/resources/config/app/ (see configuration guide). If they are for test purposes they should be named beans-test-*.xml and located under src/test/resources/config/app/. This helps you to find files easier and faster during development in your IDE. Additionally, we defined a recommendation how to structure the spring XML configurations of your application as you can see in our sample application.

- src/main/resources/config/app/
  - beans-application.xml
  - common/
    - beans-common.xml
    - …
  - logic/
    - beans-logic.xml
    - …
  - persistence/
    - beans-persistence.xml
    - beans-jpa.xml
    - …
  - service/
    - beans-service.xml
    - …

### 4.3.3 Key Principles

A Bean in CDI (Context and Dependency-Injection) or Spring is typically part of a larger component and encapsulates some piece of logic that should in general be replaceable. As an example we can think of a Use-Case, Data-Access-Object (DAO), etc. As best practice we use the following principles:

- **Separation of API and implementation**
  We create a self-contained API documented with JavaDoc. Then we create an implementation of this API that we annotate with @Named. This implementation is treated as secret. Code from other components that wants to use the implementation shall only rely on the API. Therefore we use dependency injection via the interface with the @Inject annotation.

- **Stateless implementation**
  By default implementations (CDI-Beans) shall always be stateless. If you store state information in member variables you can easily run into concurrency problems and nasty bugs. This is easy to avoid by using local variables and separate state classes for complex state-information. Try to avoid stateful

CDI-Beans wherever possible. Only add state if you are fully aware of what you are doing and properly document this as a warning in your JavaDoc.

- **Usage of JSR330**
  We use javax.inject (JSR330) and JSR250 as a common standard that makes our code portable (works in any modern JEE environemnt). However, we recommend to use the springframework as container. But we never use proprietary annotations such as @Autowired or @Required.

- **Setter Injection**
  For productive code (src/main/java) we use setter injection. Compared to private field injection this allows better testability and setting breakpoints for debugging. Compared to constructor injection it is better for maintenance. In spring integration tests (src/test/java) private field injection is preferred for simplicity.

- **KISS**
  To follow the KISS (keep it small and simple) principle we avoid advanced features (e.g. AOP, non-singleton beans) and only use them where necessary.

# 4.4 Configuration

TODO

[[guide-configuration_application-configuration-(developers)]] == Application Configuration (Developers) Spring XML, etc. TODO

[[guide-configuration_environment-configuration-(operators)]] == Environment Configuration (Operators) Properties files (Port numbers, host names, passwords, logins, timeouts, etc.) TODO

[[guide-configuration_business-configuration-(users/admins)]] == Business Configuration (Users/Admins) Stored in DB, Edited via Dialog, TODO

## 4.4.1 Configuration Files

Layout

- src/main/resources/
  - config/
    - env
      - db.properties
      - …
    - app
      - beans-application.xml
      - …

## 4.5 Validation

For regular validation we use the JSR303 standard that is also called bean validation (BV). Details can be found in the specification. As implementation we use hibernate-validation.

### 4.5.1 Example

TODO

### 4.5.2 GUI-Integration

TODO (works fine with GWT, no idea for AngularJS or other JS based frameworks - generation from BV annotated TOs to JS code???)

### 4.5.3 Cross-Field Validation

TODO BV has poor support for this. Best practice is to create and use beans for ranges, etc. that solve this.

### 4.5.4 Complex Validation

TODO for complex and stateful business validations we do not use BV (possible with groups and context, etc.) but follow KISS and just implement this on the server in a straight forward manner.

## 4.6 Aspect Oriented Programming (AOP)

AOP is a powerful feature for cross-cutting concerns. However, if used extensive and for the wrong things an application can get unmaintainable. Therefore we give you the best practices where and how to use AOP properly.

### 4.6.1 Transactions

@Transactional TODO

### 4.6.2 Security

Authrization via JSR250, TOOD, see

### 4.6.3 Exception Facade

If there is no alternate solution to AOP for implementing the exception facade pattern, we can use AOP. TODO

### 4.6.4 Logging

For tests and to trace down errors in different environments we can use trace- or exception logging aspects. TODO

### 4.6.5 AOP Key Principles

We follow these principles:

- We use spring AOP based on dynamic proxies (and fallback to cglib).
- We do not use AspectJ and other mighty and complex AOP frameworks
- We only use AOP where we consider it as necessary (see the sections of this guide).

## 4.7 Exception Handling

TODO

- We use unchecked exceptions (RuntimeException)
- For unexpected and exotic situations it is sufficient to throw existing exceptions such as IllegalStateException
- For business exceptions and application specific technical exceptions we define our own exception classes with I18N
- Our own exceptions derive from a exception base class supporting
  - UUID per instance (reused from cause)
  - Error code per class
  - message templating (see I18N)

## 4.8 Internationalization

Internationalization (I18N) is about writing code independent from locale-specific informations.

TODO

Proposal: http://m-m-m.sourceforge.net/apidocs/net/sf/mmm/util/nls/api/package-summary.html#documentation

# 4.9 Monitoring

TODO

## 4.9.1 JMX

We use JMX to provide monitoring information via MBeans.

#### 4.9.1.1 MBeans

```
TODO MBean example
```

#### 4.9.1.2 Using JMX

TODO: explain jvisualvm and VisualVM-MBeans Plugin give some hints and links on heap analysis and samplers

#### 4.9.1.3 Integration with Nagios

## 4.9.2 GC-Logging

TODO: Explain JVM options, minimum overhead, always turn on, link to gcvisualizer Explain G1 collector that scales for large heaps

## 4.9.3 Loadbalancing

TODO explain status URL and integration with LB

# 4.10 XML

XML (eXtensible Markup Language) is a W3C standard format for structured information. It has a large eco-system of additional standards and tools.

In Java there are many different APIs and frameworks for accessing, producing and processing XML. For the OASP we recommend to use JAXB for mapping Java objects to XML and vice-versa. Further there is the popular DOM API for reading and writing smaller XML documents directly. When processing large XML documents StAX is the right choice.

## 4.10.1 JAXB

We use JAXB to serialize Java objects to XML or vice-versa.

#### 4.10.1.1 JAXB and Inheritance

TODO @XmlSeeAlso http://stackoverflow.com/questions/7499735/jaxb-how-to-create-xml-from-polymorphic-classes

#### 4.10.1.2 JAXB Custom Mapping

In order to map custom datatypes or other types that do not follow the Java bean conventions, you need to define a custom mapping. If you create dedicated objects dedicated for the XML mapping you can easily avoid such situations. When this is not suitable follow these instructions to define the mapping: TODO

https://weblogs.java.net/blog/kohsuke/archive/2005/09/using_jaxb_20s.html

## 4.11 JSON

JSON (JavaScript Object Notation) is a popular format to represent and exchange data especially for modern web-clients. For mapping Java objects to JSON and vice-versa there is no official standard API. We use the established and powerful open-source solution Jackson.

### 4.11.1 JSON and Inheritance

If you are using inheritance for your objects mapped to JSON then polymorphism can not be supported out-of-the box. So in general avoid polymorphic objects in JSON mapping. However, this is not always possible. Have a look at the following example from our sample application:

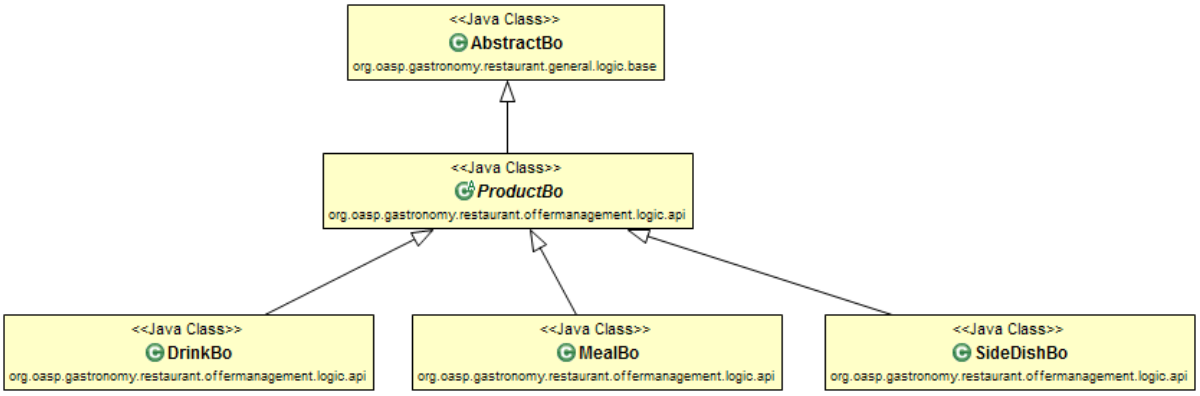



*Figure 4.1. Transfer-Objects using Inheritance*

Now assume you have a REST service operation as Java method that takes a ProductBo as argument. As this is an abstract class the server needs to know the actual sub-class to instantiate. We typically do not want to specify the classname in the JSON as this should be an implementation detail and not part of the public JSON format (e.g. in case of a service interface). Therefore we use a symbolic name for each polymorphic subtype that is provided as virtual attribute @type within the JSON data of the object:

```
{ "@type": "Drink", ... }
```

The easiest way to archive this is by adding annotations to your polymorphic Java objects:

```
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, include = As.PROPERTY, property = "@type")
@JsonSubTypes({ @Type(value = DringBo.class, name = "Drink"), @Type(value = MealBo.class, name =
 "Meal"),
  @Type(value = SideDishBo.class, name = "SideDish") })
public abstract class ProductBo extends AbstractBo {
  ...
}
```

However, to avoid dependencies to proprietary annotations of the JSON framework in your (business) objects the OASP provides you with the class ObjectMapperFactory in the oasp4j-rest module that you can subclass to configure jackson for your polymorphic types. Here is an example from the sample application:

```
@Named("RestaurantObjectMapperFactory")
public class RestaurantObjectMapperFactory extends ObjectMapperFactory {

  public RestaurantObjectMapperFactory() {
    super();
    setBaseClasses(ProductBo.class);
```

```
    setSubtypes(new NamedType(MealBo.class, "Meal"), new NamedType(DrinkBo.class, "Drink"), new
 NamedType(
        SideDishBo.class, "SideDish"));
  }
}
```

Here we use setBaseClasses to register the top-level classes of polymorphic objects. Then you declare all concrete polymorphic sub-classes together with their symbolic name for the JSON format via setSubtypes.

### 4.11.2 JSON Custom Mapping

In order to map custom datatypes or other types that do not follow the Java bean conventions, you need to define a custom mapping. If you create objects dedicated for the JSON mapping you can easily avoid such situations. When this is not suitable follow these instructions to define the mapping:

1. As an example, the use of JSR354 (javax.money) is appreciated in order to process monetary amounts properly. However, without custom mapping, the default mapping of Jackson will produce the following JSON for a MonetaryAmount:

   ```
   "currency": {"defaultFractionDigits":2, "numericCode":978, "currencyCode":"EUR"},
   "monetaryContext": {...},
   "number":6.99,
   "factory": {...}
   ```

   As clearly can be seen, the JSON contains too much information and reveals implementation secrets that do not belong here. Instead the JSON output expected and desired would be:

   ```
   "currency":"EUR","amount":"6.99"
   ```

   Even worse, when we send the JSON data to the server, Jackson will see that MonetaryAmount is an interface and does not know how to instantiate it so the request will fail. Therefore we need a customized Serializer and Deserializer.

2. We implement MonetaryAmountJsonSerializer to define how a MonetaryAmount is serialized to JSON:

   ```
   public final class MonetaryAmountJsonSerializer extends JsonSerializer<MonetaryAmount> {

     public static final String NUMBER = "amount";
     public static final String CURRENCY = "currency";

     public void serialize(MonetaryAmount value, JsonGenerator jgen, SerializerProvider provider) throws
    ... {
       if (value != null) {
         jgen.writeStartObject();
         jgen.writeFieldName(MonetaryAmountJsonSerializer.CURRENCY);
         jgen.writeString(value.getCurrency().getCurrencyCode());
         jgen.writeFieldName(MonetaryAmountJsonSerializer.NUMBER);
         jgen.writeString(value.getNumber().toString());
         jgen.writeEndObject();
       }
     }
   ```

   For composite datatypes it is important to wrap the info as an object (writeStartObject() and writeEndObject()). MonetaryAmount provides the information we need by the methods getCurrency() and getNumber(). So that we can easily write them into the JSON data.

3. Next, we implement MonetaryAmountJsonDeserializer to define how a MonetaryAmount is deserialized back as Java object from JSON:

```
public final class MonetaryAmountJsonDeserializer extends AbstractJsonDeserializer<MonetaryAmount> {
  protected MonetaryAmount deserializeNode(JsonNode node) {
    BigDecimal number = getRequiredValue(node, MonetaryAmountJsonSerializer.NUMBER,
 BigDecimal.class);
    String currencyCode = getRequiredValue(node, MonetaryAmountJsonSerializer.CURRENCY,
 String.class);
    MonetaryAmount monetaryAmount =
        MonetaryAmounts.getAmountFactory().setNumber(number).setCurrency(currencyCode).create();
    return monetaryAmount;
  }
}
```

For composite datatypes we extend from AbstractJsonDeserializer as this makes our task easier. So we already get a JsonNode with the parsed payload of our datatype. Based on this API it is easy to retrieve individual fields from the payload without taking care of their order, etc. AbstractJsonDeserializer also provides methods such as getRequiredValue to read required fields and get them converted to the desired basis datatype. So we can easily read the amount and currency and construct an instance of MonetaryAmount via the official factory API.

4. Finally we need to register our custom (de)serializers as following:

```
@Named("RestaurantObjectMapperFactory")
public class RestaurantObjectMapperFactory extends ObjectMapperFactory {

  public RestaurantObjectMapperFactory() {
    super();
    // ...
    SimpleModule module = getExtensionModule();
    module.addDeserializer(MonetaryAmount.class, new MonetaryAmountJsonDeserializer());
    module.addSerializer(MonetaryAmount.class, new MonetaryAmountJsonSerializer());
  }
}
```

After we have registered this factory (see above) we're done!

# 4.12 Testing

## 4.12.1 Module Testing

For module testing we use JUnit and TODO-Mock (EasyMock, PowerMock, Mockito)? Coverage, etc.

## 4.12.2 Integration Testing

For integration testing we use spring-test, etc.

## 4.12.3 System Testing

For system testing we use selenium, etc.

## 4.12.4 Manual Developer Tests

TODO

## 4.12.5 Testdata

For integration and system tests we need testdata to build our test on. This includes the general master data but also example data just for testing purposes. TODO…

> **joedoehh: Notes on testing**
> ***Design Goals***
>
> - Be able to unit test your spring components without the need of loading the full spring-context with its ad-hoc test configurations is a great advantage because it's clean, easy to maintain, faster to write, smooth to alter.
> - embrace assertThat()

> **joedoehh: Proposal on how to do testing**
> ***Tools:*** *(always most recent version):*
>
> - JUnit, Hamcrest, Mockito
> - Arquillian with DBUnit + Drone Extension (selenium, rest, …)
> - Tomcat 8 as container
>
> ***Steps / Units of Work:***
>
> - Step 1: Module test for classes of logical layer + DAOs of persistence layer
> - Step 2: Integration test horizontal following the example application slices offermanagement, salesmanagment, …
> - Step 3: Integration test vertically testing slices interaction between offermanagement, salesmanagment, …
> - Step 4: Functional testing: Web UI, Rest layer, from browser to db and back
>
> ***Principles:***
>
> - every test must be executable manually from ide and maven so later in ci environments jenkins can kick of test execution

- promote usage of `assertThat()` - no one needs or should use `assertTrue()`, `assertNotNull()`, … any longer these days
- no big testframework
- out-of-scope testdatamangement, testools for manual tests, …

## 4.13 Transfer-Objects

TODO explain general pattern and decoupling from persistent entities, data sovereignty

### 4.13.1 Business-Objects

TODO: DTO/CTO concept

http://m-m-m.sourceforge.net/apidocs/net/sf/mmm/util/transferobject/api/package-summary.html#documentation

### 4.13.2 Service-Objects

For external services consumed by other applications we create separate transfer-objects to keep the service API stable (see service layer).

## 4.14 Datatypes

A datatype is an object representing a value of a specific type with the following aspects:

- It has a technical or business specific semantic.
- Its JavaDoc explains the meaning and semantic of the value.
- It is immutable and therefore stateless (its value assigned at construction time and can not be modified).
- It is Serializable.
- It properly implements #equals(Object) and #hashCode() (two different instances with the same value are equal and have the same hash).
- It shall ensure syntactical validation so it is NOT possible to create an instance with an invalid value.
- It is responsible for formatting its value to a string representation suitable for sinks such as UI, loggers, etc. Also consider cases like a Datatype representing a password where toString() should return something like "**" instead of the actual password to prevent security accidents.
- It is responsible for parsing the value from other representations such as a string (as needed).
- It shall provide required logical operations on the value to prevent redundancies. Due to the immutable attribute all manipulative operations have to return a new Datatype instance (see e.g. BigDecimal.add(java.math.BigDecimal)).
- It should implement Comparable if a natural order is defined.

Based on the Datatype a presentation layer can decide how to view and how to edit the value. Therefore a structured data model should make use of custom datatypes in order to be expressive. Common generic datatypes are String, Boolean, Number and its subclasses, Currency, etc. Please note that both Date and Calendar are mutable and have very confusing APIs. Therefore, use JSR-310 or jodatime instead. Even if a datatype is technically nothing but a String or a Number but logically something special it is worth to define it as a dedicated datatype class already for the purpose of having a central javadoc to explain it. On the other side avoid to introduce technical datatypes like String32 for a String with a maximum length of 32 characters as this is not adding value in the sense of a real Datatype. It is suitable and in most cases also recommended to use the class implementing the datatype as API omitting a dedicated interface.

— mmm project *datatype javadoc*

See mmm datatype javadoc.

### 4.14.1 Datatype Packaging

For the OASP we use a common packaging schema. The specifics for datatypes are as following:

| Segment | Value | Explanation |
| --- | --- | --- |
| <component> | * | Here we use the (business) component defining the |

| Segment | Value | Explanation |
| --- | --- | --- |
| | | datatype or general for generic datatypes. |
| <layer> | common | Datatypes are used across all layers and are not assigned to a dedicated layer. |
| <scope> | api | Datatypes are always used directly as API even tough they may contain (simple) implementation logic. Most datatypes are simple wrappers for generic Java types (e.g. String) but make these explicit and might add some validation. |

### 4.14.2 Datatypes in Entities

The usage of custom datatypes in entities is explained in the persistence layer guide.

http://m-m-m.sourceforge.net/apidocs/net/sf/mmm/persistence/impl/hibernate/usertype/package-summary.html

### 4.14.3 Datatypes in Transfer-Objects

#### 4.14.3.1 JAXB

TODO (explain)

#### 4.14.3.2 JAX-RS

The usage of custom datatypes in transfer-objects used in REST services is explained in the serivce layer guide.

#### 4.14.3.3 JAX-WS

TODO (avoid)

# 4.15 Transaction Handling

Transactions are technically processed by the presentation layer. However, the transaction control has to be performed in upper layers. To avoid dependencies on persistence layer and technical code in upper layers, we use AOP to add transaction control via annotations as aspect.

As we recommend using spring, we use the @Transactional annotation (for a JEE application server you would use @TransactionAttribute instead). We use this annotation in the service layer to annotate services that participate in transactions (what typically applies to all services).

```
@Transactional
public class MyExampleServiceImpl {
  public MyDataTo getData(MyCriteriaTo criteria) {
    ...
  }
  ...
}
```

## 4.15.1 Batches

Transaction control for batches is a lot more complicated and is described in the batch layer.

## 4.16 Accessibility

TODO

http://www.w3.org/TR/WCAG20/

http://www.w3.org/WAI/intro/aria

http://www.einfach-fuer-alle.de/artikel/bitv/

http://www.banu.bund.de