Open Application Standard Platform for Java V1.0.0

Table of Contents

Introduction

The *Open Application Standard Platform* (OASP) provides a solution to building applications which combine best-in-class frameworks and libraries as well as industry proven practices and code conventions. It massively speeds up development, reduces risks and helps you to deliver better results.

This document contains the complete compendium of the *Open Application Standard Platform for Java* (OASP4J). From this link you will also find the latest release or nightly snapshot of this documentation.

1. Architecture

There are many different views on what is summarized by the term *architecture*. First we introduce the key principles and architecture principles of the OASP. Then we go into details of the the architecture of an application.

1.1 Key Principles

For the OASP we follow these fundamental key principles for all decisions about architecture, design, or choosing standards, libraries, and frameworks:

- **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 Architecture Principles

Additionally we define the following principles that our architecture is based on:

- **Component Oriented Design**
  We follow a strictly component oriented design to address the following sub-principles:
  - Separation of Concerns
  - Reusability and avoiding redundant code
  - Information Hiding via component API and its exchangeable implementation treated as secret.
  - *Design by Contract* for self-contained, descriptive, and stable component APIs.
  - Layering as well as separation of business logic from technical code for better maintenance.
  - *Data Sovereignty* (and *high cohesion with low coupling*) says that a component is responsible for its data and changes to this data shall only happen via the component. Otherwise maintenance problems will arise to ensure that data remains consistent. Therefore interfaces of a component that may be used by other components are designed *call-by-value* and not *call-by-reference*.
- **Homogeneity**
  Solve similar problems in similar ways and establish a uniform code-style.

1.3 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, load-balancing and networking.

1.3.1 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.

1.3.2 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 category such as service 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.
- data-access layer for the data access (esp. persistence).

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 left. 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 the allowed dependencies illustrated by the dark green connectors. Within a business component a component part can call the next component part on the layer directly below via a dependency on its API (vertical connectors). 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 allowed to depend on component A. Therefore a use-case within the logic component part of X is allowed to call a use-case from A via a dependency on the component API. The same applies for dialogs on the client layer. This is illustrated by the horizontal connectors. Please note that persistence entities are part of the API of the data-access component part so only the logic component part of the same business component may depend on them.

The technical architecture has to address non-functional requirements:

- **scalability**
  is established by keeping state in the client and making the server state-less (except for login session). Via load-balancers new server nodes can be added to improve performance (horizontal scaling).
- **availability** and **reliability**
  are addressed by clustering with redundant nodes avoiding any single-point-of failure. If one node fails the system is still available. Further the software has to be robust so there are no dead-locks or other bad effects that can make the system unavailable or not reliable.
- **security**
  is archived in the OASP by the right templates and best-practices that avoid vulnerabilities. See security guidelines for further details.
- **performance**
  is obtained by choosing the right products and proper configurations. While the actual implementation of the application matters for performance a proper design is important as it is the key to allow performance-optimizations (see e.g. caching).

1.3.2.1 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	CXF
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

1.3.3 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. Please note that this highly depends on the operational infrastructure so it is just a suggestion.

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

- **Information-Tier**
  contains the systems for communication with users and external systems. Here authentication and per-authorization takes place in order to keep not permitted requests out of the lower tiers. Also this is the central place for an IDS. Besides protocol-transformation and other technical aspects the systems in this tier act as proxies to the real applications in the *logic-tier*.

- **Application-Tier**
  contains the actual business applications as well as applications for cross cutting logic (e.g. Servers for GIS, LDAP, Printing, etc.). These systems should be build following the principles of the application architecture.

- **Data-Tier**
  contains the systems for storing the data such as an RDBMS.

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

Figure 1.2. Infrastructure 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 io.oasp.gastronomy.restaurant.salesmanagement.dataaccess.api.dao

Table 2.1. Segments of package schema

Segment	Description	Example
<organization>	Is the basic Java Package name-space of the organization	io.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 (dataaccess, logic, service, batch, gui, client) or common for code not assigned to a technical layer (datatypes, cross-cutting concerns).	dataaccess
<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 io.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 io.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 Client Layer

There are various technical approaches to build GUI clients. The OASP proposes rich clients that connect to the server via data-oriented services (e.g. using REST with JSON). In general we have to distinguish the following types of clients:

- web clients
- native desktop clients
- (native) mobile clients

3.1.1 Web Clients

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

3.1.2 Native Desktop Clients

Currently not addressed. There are plans about JavaFx for the future.

3.1.3 Mobile Clients

Dependent on target mobile platform. Android may be addressed due to Java.

To support all mobile platforms with moderate effort, we recommend to use Cordova and ionic.

3.1.4 Security

Security is not only an aspect for the server-side but also for clients. Especially in web-clients security threads such as XSS and CSRF have to be addressed. Therefore you should use proper frameworks that systematically prevent such security pitfalls. It should be easy for developers to do things correct and hard to do things wrong. This is absolutely not the case with plain web-technology. Therefore you should follow the guidelines of OASP4JS to prevent security problems.

3.2 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

3.2.1 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

3.2.2 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/v1_0/MyService)
- 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.

> **Note**
>
> (JH) Should we distinguish between major and minor version here (this is more implementation specific and can cause mystic discussions where linear versioning v1, v2, v3 would be more KISS)?

3.2.3 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*.

3.2.4 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.

3.2.4.1 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. 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 TablemanagmentWebService {

  @WebMethod
  @WebResult(name = "message")
  TableEto getTable(@WebParam(name = "id") String id);

}
```

And here is a simple implementation of the service:

```
@Named("TablemanagementWebService")
@WebService(endpointInterface =
 "io.oasp.gastronomy.restaurant.tablemanagement.service.api.ws.TablemanagmentWebService")
public class TablemanagementWebServiceImpl implements TablemanagmentWebService {

  private Tablemanagement tableManagement;

  @Override
  public TableEto getTable(String id) {

    return this.tableManagement.findTable(id);
  }
```

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

```
  <jaxws:endpoint id="tableManagement" implementor="#TablemanagementWebService" address="/ws/
Tablemanagement/v1_0"/>
```

The implementor attribute references an existing bean with the ID TablemanagementWebService 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 write adapters for JAXB (see XML).

SOAP Testing

For testing SOAP services in general consult the testing guide.

For testing SOAP services manually we strongly recommend SoapUI.

3.2.4.2 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 an entity. Such URI identifies the entire collection of all elements of this type. Example: https://mydomain.com/myapp/services/rest/mycomponent/myentity
- 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/mycomponent/myentity/42

The following table specifies how to use the HTTP methods (verbs) for collection and element URIs properly (see wikipedia). For general design considerations beyond this documentation see the API Design eBook.

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)

HTTP Method	Meaning (Element URI)	Meaning (Collection URI)
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.

JAX-RS Configuration

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
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="io.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)

3.2.4.3 HTTP-Invoker

HTTP-Invoker is a very simple and easy to use communication protocol that is part of spring remoting. It simply sends the serialized method call with all its arguments and sends the data via HTTP(S).

3.2.5 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.

3.2.6 Security

A common security threat is CSRF for REST services. Therefore all REST operations that are performing modifications (PUT, POST, DELETE, etc. - all except GET) have to be secured against CSRF attacks. In OASP4J we are using spring-security that already solves CSRF token generation and verification. The integration is part of the application template as well as the sample-application.

3.3 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.

3.3.1 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. 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 transfer-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:

```
public interface UcFindStaffMember {

  StaffMemberEto getStaffMemberByLogin(String login);

  StaffMemberEto getStaffMember(Long id);
}
```

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 component interface. Further it shall 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 io.oasp.gastronomy.restaurant.staffmanagement.logic.impl;

public class UcFindStaffMemberImpl extends AbstractStaffMemberUc implements UcFindStaffMember {

  public StaffMemberEto getStaffMemberByLogin(String login) {
    StaffMemberEntity staffMember = getStaffMemberDao().searchByLogin(login);
    return getBeanMapper().map(staffMember, StaffMemberEto.class);
  }

  public StaffMemberEto getStaffMember(Login id) {
    StaffMemberEntity staffMember = getStaffMemberDao().find(id);
    return getBeanMapper().map(staffMember, StaffMemberEto.class);
  }
}
```

As you can see, provided entities are mapped to corresponding business objects (here StaffMemberEto.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 TableEto to a Table looks like this:

```
<mapping>
  <class-a>io.oasp.gastronomy.restaurant.tablemanagement.logic.api.TableEto</class-a>
  <class-b>io.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.

3.3.2 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 io.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 StaffMemberEto getStaffMember(String login) {
        return this.ucFindStaffMember.getStaffMember(login);
    }

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

    @Override
    public void updateStaffMember(StaffMemberEto 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. io.oasp.gastronomy.restaurant.staffmanagement.logic.api or io.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.

3.3.2.1 Passing Parameters Among Components

Entities have to be detached for the reasons of data sovereignty, if entities are passed among components or layers (to service layer). For further details see Bean-Mapping. Therefore we are using transfer-objects (TO) with the same attributes as the entity that is persisted. The packages are:

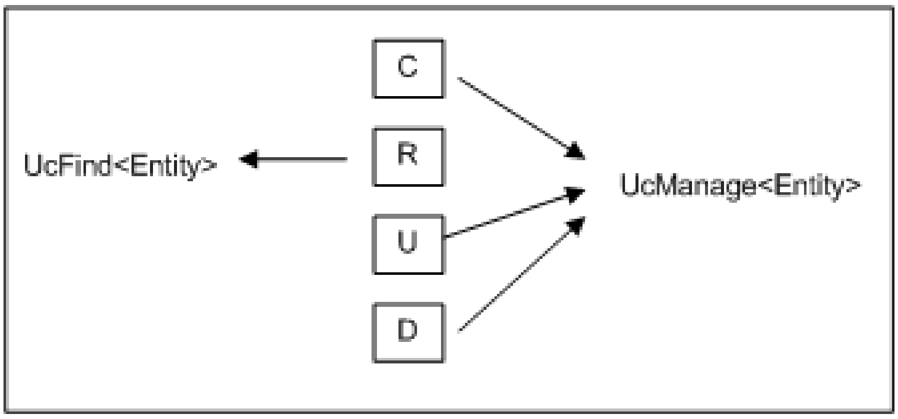
```
Persistence Entities: <package-name-prefix>.<domain>.<application-
name>.<component>.persistence.api.entity
Transfer Objects(TOs): <package-name-prefix>.<domain>.<application-name>.<component>.logic.api
```

This mapping is a simple copy process. So changes out of the scope of the owning component to any TO do not directly affect the persistent entity.

3.3.2.2 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.

3.4 Data-Access Layer

The data-access layer is responsible for all outgoing connections to access and process data. This is mainly about accessing data from a persistent data-store but also about invoking external services.

3.4.1 Persistence

For mapping java objects to a relational database we use the Java Persistence API (JPA). 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.

3.4.1.1 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 configured via JPA annotations (javax.persistence). Usually an entity class corresponds to a table of a database and a property to a row of that table.

A Simple Entity

The following listing shows a simple example:

```
@Entity
@Table(name="TEXTMESSAGE")
public class Message extends AbstractPersistenceEntity {

  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 a 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.

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. In order to map atomic custom datatypes (implementations of SimpleDatatype) we implement an AttributeConverter. Here is a simple example:

```
@Converter(autoApply = true)
public class MoneyAttributeConverter implements AttributeConverter<Money, BigDecimal> {

  public BigDecimal convertToDatabaseColumn(Money attribute) {
    return attribute.getValue();
  }

  public Money convertToEntityAttribute(BigDecimal dbData) {
    return new Money(dbData);
  }
}
```

The annotation @Converter is detected by the JPA vendor if the annotated class is in the packages to scan (see beans-jpa.xml). Further, autoApply = true implies that the converter is automatically used for all properties of the handled datatype. Therefore all entities with properties of that datatype will automatically be mapped properly (in our example Money is mapped as BigDecimal).

In case you have a composite datatype that you need to map to multiple columns the JPA does not offer a real solution. As a workaround you can use a bean instead of a real datatype and declare it as @Embeddable. If you are using hibernate you can implement CompositeUserType. Via the @TypeDef annotation it can be registered to hibernate. If you want to annotate the CompositeUserType implementation itself you also need another annoation (e.g. MappedSuperclass tough not technically correct) so it is found by the scan.

Enumerations

By default JPA maps Enums via their ordinal. Therefore the database will only contain the ordinals (0, 1, 2, etc.) so inside the database you can not easily understand their meaning. Using @Enumerated with EnumType.STRING allows to map the enum values to their name (Enum.name()). Both approaches are fragile when it comes to code changes and refactorings (if you change the order of the enum values or rename them) after being in production with your application. If you want to avoid this and get a robust mapping you can define a dedicated string in each enum value for database representation that you keep untouched. Then you treat the enum just like any other custom datatype.

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 will cause problems if BLOBs get large 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.

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 io.oasp.module.jpa.persistence.api.AbstractPersistenceEntity 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)).

3.4.1.2 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.

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 io.oasp.module.jpa.persistence.api.Dao:

```
public interface MyEntityDao extends Dao<MyEntity> {

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

As you can see, the interface Dao has a type parameter for the entity class. All CRUD operations are only inherited so you only have to declare the additional methods.

DAO Implementation

Implementing a DAO is quite simple. We crate a class named <Entity>DaoImpl that extends io.oasp.module.jpa.persistence.base.AbstractDao and implements your <Entity>Dao interface:

```
public class MyEntityDaoImpl extends AbstractDao<MyEntity> implements MyEntityDao {

  public List<MyEntity> findByCriteria(MyEntitySearchCriteria criteria) {
    TypedQuery<MyEntity> query = createQuery(criteria, getEntityManager());
    return query.getResultList();
  }
  ...
}
```

As you can see AbstractDao already implements the CRUD operations so you only have to implement the additional methods that you have declared in your <Entity>Dao interface. In the DAO implementation you can use the method getEntityManager() to access the EntityManager from the JPA. You will need the EntityManager to create and execute queries.

3.4.1.3 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.

Static Queries

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

Add the following query to this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings version="1.0" xmlns="http://java.sun.com/xml/ns/persistence/orm" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm http://java.sun.com/xml/ns/persistence/
orm_1_0.xsd">
  <named-query name="get.open.order.positions.for.order">
    <query><![CDATA[SELECT op FROM OrderPosition op where op.order.id = ? AND op.state NOT IN (PAYED,
 CANCELLED)]]></query>
  </named-query>
  ...
</hibernate-mapping>
```

To avoid redundant occurrences of the query name (get.open.order.positions.for.order) we define the constants for each named query:

```
package io.oasp.gastronomy.restaurant.general.common.api.constants;

public class NamedQueries {
  public static final String GET_OPEN_ORDER_POSITION_FOR_ORDER = "get.open.order.positions.for.order";
}
```

Note that changing the name of the java constant (GET_OPEN_ORDER_POSITION_FOR_ORDER) can be done easily with refactoring. Further you can trace where the query is used by searching the references of the constant.

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.

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 or here.

Dynamic Queries

For dynamic queries we recommend to use QueryDSL. It allows to implement queries in a powerful but readable and type-safe way (unlike Criteria API). If you already know JPQL you will quickly be able to read and write QueryDSL code. It feels like JPQL but implemented in Java instead of plain text.

Please be aware that code-generation can be painful especially with large teams. We therefore recommend to use QueryDSL without code-generation. Here is an example from our sample application:

```
  public List<OrderEntity> findOrders(OrderSearchCriteriaTo criteria) {

    OrderEntity order = Alias.alias(OrderEntity.class);
    EntityPathBase<OrderEntity> alias = Alias.$(order);
    JPAQuery query = new JPAQuery(getEntityManager()).from(alias);
    Long tableId = criteria.getTableId();
    if (tableId != null) {
      query.where(Alias.$(order.getTableId()).eq(tableId));
    }
    OrderState state = criteria.getState();
    if (state != null) {
      query.where(Alias.$(order.getState()).eq(state));
    }
    applyCriteria(criteria, query);
    return query.list(alias);
  }
```

Using Wildcards

For flexible queries it is often required to allow wildcards (especially in dynamic queries). While users intuitively expect glob syntax the SQL and JPQL standards work different. Therefore a mapping is required (see here).

Query Meta-Parameters and Paging

A query allows to set some meta-parameters such as maxResults, firstResult (offset), or timeout. The OASP provides the method applyCriteria in AbstractGenericDao that applies meta-parameters to a query based on AbstractSearchCriteria. So all you need to do is derive your individual search criteria object from AbstractSearchCriteria and you can use applyCriteria in the query implementation of your DAO. Then the query allows paging by setting maxResults (AbstractSearchCriteria.setMaximumHitCount(Integer)) to the number of hits per page (plus one extra hit to determine if there are more hits available) and increasing the firstResult (AbstractSearchCriteria.setHitOffset(int)) by the number of hits per page to step to the next page. If you allow the client to specify maxResults it is recommended to limit this value on the server side to prevent performance problems or DOS-attacks.

3.4.1.4 Relationships

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.

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).

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).

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.

3.4.1.5 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:

```
...
@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, equals, hashCode
}
```

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.

3.4.1.6 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).

As a best practice we advise you to avoid deep class hierarchies among entity classes (unless they reduce complexity).

3.4.1.7 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* (property hibernate.connection.isolation in the jpa.xml, see here).

A remaining anomaly is when two stakeholders concurrently access a record, do some changes and write them back to the database. The JPA addresses this with different locking strategies (see here or here).

As a best practice we are using optimistic locking for regular end-user services (OLTP) and pessimistic locking for batches.

Optimistic Locking

The class io.oasp.module.jpa.persistence.api.AbstractPersistenceEntity already provides optimistic locking via a modificationCounter with the @Version annotation. Therefore JPA takes care of optimistic locking for you. When entities are transferred to clients, modified and sent back for update you need to ensure the modificationCounter is part of the game. If you follow our guides about transfer-objects and services this will also work out of the box. You only have to care about two things:

- How to deal with optimistic locking in relationships?
  Assume an entity A contains a collection of B entities. Should there be a locking conflict if one user modifies an instance of A while another user in parallel modifies an instance of B that is contained in the other instance?

- What should happen in the UI if an OptimisticLockException occurred?

According to KISS our recommendation is that the user gets an error displayed that tells him to do his change again on the recent data. Try to design your system and the work processing in a way to keep such conflicts rare and you are fine.

Pessimistic Locking

For back-end services and especially for batches optimistic locking is not suitable. A human user shall not cause a large batch process to fail because he was editing the same entity. Therefore such use-cases use pessimistic locking what gives them a kind of priority over the human users. In your DAO implementation you can provide methods that do pessimistic locking via EntityManager operations that take a LockModeType. Here is a simple example:

```
getEntityManager().lock(entity, LockModeType.READ);
```

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.

3.4.1.8 Testing Entities and DAOs

See testing guide.

3.4.1.9 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.

3.4.2 Database Configuration

The configuration for spring and hibernate is already provided by OASP in our sample application and the application template. So you only need to worry about a few things to customize.

3.4.2.1 Database System and Access

Obviously you need to configure why type of database you want to use as well as the location and credentials to access it. The defaults are configured in application-default.properties that is bundled and deployed with the release of the software. It should therefore contain the properties as in the given example:

```
database.url=jdbc:postgresql://database.enterprise.com/app
database.user.login=appuser01
database.hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
database.hibernate.hbm2ddl.auto=validate
```

The environment specific settings (especially passwords) are configured by the operators in application.properties. For further details consult the configuration guide. It can also override the default values. The relevant configuration properties can be seen by the following example for the development environment (located in src/test/resources):

```
database.url=jdbc:postgresql://localhost/app
database.user.password=************
database.hibernate.hbm2ddl.auto=create
```

For further details about database.hibernate.hbm2ddl.auto please see here. For production and acceptance environments we use the value validate that should be set as default.

3.4.2.2 Database Migration

For database migrations we use Flyway. Flyway can be used standalone or can be integrated via its api to make sure the database migration takes place on startup. To enable auto migration on startup (not recommended for productive environment) set the following property in the application.properties file.

```
database.migration.auto = true
```

It is set to false by default via application-default.properties and shall be done explicitly in production environments. For development environment it is set to true in order to simplify development. This is also recommend for regular test environments.

If you want to use Flyway set the following property in any case to prevent Hibernate from doing changes on the database (pre-configured by default of OASP):

```
database.hibernate.hbm2ddl.auto=validate
```

New database migrations are added to src/main/resources/db/migrations. They can be SQL based or Java based and follow this naming convention: V<version>__<description> (e.g.: V0003__Add_new_table.sql). For new SQL based migrations also stick to the following conventions:

- properties in camlCase
- tables in UPPERCASE
- ID properties with underscore (e.g. table_id)
- constraints all UPPERCASE with
- PK_{table} for primary key
- FK_{sourcetable}2{target} for foreign keys
- UK_{table}_{property} for unique constraints
- Inserts always with the same order of properties in blocks for each table
- Insert properties always starting with id, modificationCounter, [dtype,] …

3.4.3 Security

3.4.3.1 SQL-Injection

A common security threat is SQL-injection. Never build queries with string concatenation or your code might be vulnerable as in the following example:

```
String query = "Select op from OrderPosition op where op.comment = " + userInput;
return getEntityManager().createQuery(query).getResultList();
```

Via the parameteter userInput an attacker can inject SQL (JPQL) and execute arbitrary statements in the database causing extreme damage. In order to prevent such injections you have to strictly follow our rules for queries: Use named queries for static queries and QueryDSL for dynamic queries. Please also consult the SQL Injection Prevention Cheat Sheet.

3.4.3.2 Limited Permissions for Application

We suggest that you operate your application with a database user that has limited permissions so he can not modify the SQL schema (e.g. drop tables). For initializing the schema (DDL) or to do schema migrations use a separate user that is not used by the application itself.

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>io.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. Simply copy this configuration into your application 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 can just type LOG and hit [ctrl][space] (and then [arrow up]) 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. The security guides provided by this document will not automatically prevent you from any harm, but they may give you hints and best practices already used in different software products.

4.2.1 Authentication

Definition:

> Authentication is the verification that somebody interacting with the system is the actual subject for whom he claims to be.

The one authenticated is properly called *subject* or *principal*. However, for simplicity we use the common term *user* even though it may not be a human (e.g. in case of a service call from an external system).

To prove his authenticity the user provides some secret called *credentials*. The most simple form of credentials is a password.

> **Note**
>
> Please never implement your own authentication mechanism or credential store. You have to be aware of implicit demands such as salting and hashing credentials, password life-cycle with recovery, expiry, and renewal including email notification confirmation tokens, central password policies, etc. This is the domain of access managers and identity management systems. In a business context you will typically already find a system for this purpose that you have to integrate (e.g. via LDAP).

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="io.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.

4.2.1.2 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="io.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.

simple Spring Security filter insertion.

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

Nevertheless, the oasp4j follows a different approach. The simple interface of Spring Security for inserting custom filters as stated above is driven by a relative alignment of the different filters been executed. You relatively can insert custom filters before or after existing ones and also at the beginning or at the end. You might easily see, that the real filter chain will get more and more invisible. Thus the oasp4j follows the default ordering of the Spring Security filter chain, such that it gets more transparent for any developer, which filters will be executed in which order and at which position a new custom filter may be inserted.

This documentation depends on Spring Security v3.2.5.RELEASE:

- general filter ordering
- detailed filter ordering

These lists will be maintained each release, which will include a Spring Security upgrade. Thus first, we will not loose any changes from the possibly updated default filter chain of Spring Security. Second, due to the absolute declaration of the filter order, you might not get any strange behavior in your system after upgrading to a new version of Spring Security.

4.2.1.3 Users vs. Systems

If we are talking about authentication we have to distinguish two forms of principals:

- human users
- autonomous systems

While e.g. a Kerberos/SPNEGO Single-Sign-On makes sense for human users it is pointless for authenticating autonomous systems. So always keep this in mind when you design your authentication mechanisms and separate access for human users from access for systems.

4.2.2 Authorization

Definition:

> Authorization is the verification that an authenticated user is allowed to perform the operation he intends to invoke.

4.2.2.1 Clarification of terms

For clarification we also want to give a common understanding of related terms that have no unique definition and consistent usage in the wild.

Table 4.3. Security terms related to authorization

Term	Meaning and comment
Permission	A permission is an object that allows a principal to perform an operation in the system. This permission can be *granted* (give) or *revoked* (taken away). Sometimes people also use the term *right* what is actually wrong as a right (such as the right to be free) can not be revoked.
Group	We use the term group in this context for an object that contains permissions. A group may also contain other groups. Then the group represents the set of all recursively contained permissions.

Term	Meaning and comment
Role	We consider a role as a specific form of group that also contains permissions. A role identifies a specific function of a principal. A user can act in a role. For simple scenarios a principal has a single role associated. In more complex situations a principal can have multiple roles but has only one active role at a time that he can choose out of his assigned roles. For KISS it is sometimes sufficient to avoid this by creating multiple accounts for the few users with multiple roles. Otherwise at least avoid switching roles at runtime in clients as this may cause problems with related states. Simply restart the client with the new role as parameter in case the user wants to switch his role.
Access Control	Any permission, group, role, etc., which declares a control for access management.

4.2.2.2 Suggestions on the access model

The access model provided by oasp4j-security follows this suggestions:

- Each Access Control (permission, group, role, …) is uniquely identified by a human readable string.
- We create a unique permission for each use-case.
- We define groups that combine permissions to typical and useful sets for the users.
- We define roles as specific groups as required by our business demands.
- We allow to associate users with a list of Access Controls.
- For authorization of an implemented use case we determine the required permission. Furthermore, we determine the current user and verify that the required permission is contained in the tree spanned by all his associated Access Controls. If the user does not have the permission we throw a security exception and thus abort the operation and transaction.
- We try to avoid negative permissions, that is a user has no permission by default but only those granted to him additively permit him for executing use cases.
- Technically we consider permissions as a secret of the application. Administrators shall not fiddle with individual permissions but grant them via groups. So the access management provides a list of strings identifying the Access Controls of a user. The individual application itself contains these Access Controls in a structured way, whereas each group forms a permission tree.

4.2.2.3 oasp4j-security

The OASP provides a ready to use module oasp4j-security that is based on spring-security and makes your life a lot easier.

Figure 4.1. OASP4J Security Model

The diagram shows the model of oasp4j-security that separates two different aspects:

- The *Indentity- and Access-Management* is provided by according products and typically already available in the enterprise landscape (e.g. an active directory). It provides a hierarchy of *primary access control objects* (roles and groups) of a user. An administrator can grant and revoke permissions (indirectly) via this way.

- The application security is using oasp4j-security defines a hierarchy of *secondary access control objects* (groups and permissions) in the file access-control-schema.xml (see example from sample app). This hierarchy defines the application internal access control schema that should be an implementation secret of the application. Only the top-level access control objects are public and define the interface to map from the primary to secondary access control objects. This mapping is simply done by using the same names for access control objects to match.

Access Control Schema

The `oasp4j-security` module provides a simple and efficient way to define permissions and roles. The file `access-control-schema.xml` is used to define the mapping from groups to permissions. The general terms discussed above can be mapped to the implementation as follows:

Table 4.4. General security terms related to oasp4j access control schema

Term	oasp4j-security implementation	Comment
Permission	`AccessControlPermission`	
Group	`AccessControlGroup`	When considering different levels of groups of different meanings, declare `type` attribute, e.g. as "group".
Role	`AccessControlGroup`	With `type="role"`.
Access Control	`AccessControl`	Super type that represents a tree of `AccessControlGroups` and `AccessControlPermissions`. If a principal "has" a `AccessControl` he also "has" all `AccessControls` with according permissions in the spanned sub-tree.

Example access-control-schema.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<access-control-schema>
  <group id="ReadMasterData" type="group">
    <permissions>
      <permission id="OfferManagement_GetOffer"/>
      <permission id="OfferManagement_GetProduct"/>
      <permission id="TableManagement_GetTable"/>
      <permission id="StaffManagement_GetStaffMember"/>
    </permissions>
  </group>

  <group id="Waiter" type="role">
    <inherits>
      <group-ref>Barkeeper</group-ref>
    </inherits>
    <permissions>
      <permission id="TableManagement_ChangeTable"/>
    </permissions>
  </group>
  ...
</access-control-schema>
```

This example `access-control-schema.xml` declares

- a group named `ReadMasterData`, which grants four different permissions, e.g., `OfferManagement_GetOffer`

- a group named `Waiter`, which
  - also grants all permissions from the group `Barkeeper`
  - in addition grants the permission `TableManagement_ChangeTable`
  - is marked to be a `role` for further application needs.

The oasp4j-security module automatically validates the schema configuration and will throw an exception if invalid.

Unfortunately, Spring Security does not provide differentiated interfaces for authentication and authorization. Thus we have to provide an `AuthenticationProvider`, which is provided from Spring Security as an interface for authentication and authorization simultaneously. To integrate the oasp4j-security provided access control schema, you can simply inherit your own implementation from the oasp4j-security provided abstract class `AbstractAccessControlBasedAuthenticationProvider` and register your `ApplicationAuthenticationProvider` as an `AuthenticationManager`. Doing so, you also have to declare the two Beans `AccessControlProvider` and `AccessControlSchemaProvider` as listed below, which are precondition for the `AbstractAccessControlBasedAuthenticationProvider`.

Example integration of oasp4j-security access control schema.

```
<bean id="AuthenticationManager" class="org.springframework.security.authentication.ProviderManager">
    <constructor-arg>
      <list>
        <ref bean="ApplicationAuthenticationProvider"/>
      </list>
    </constructor-arg>
</bean>

<bean id="AccessControlProvider" class="io.oasp.module.security.common.impl.accesscontrol.AccessControlProviderImpl"/
>
<bean id="AccessControlSchemaProvider" class="io.oasp.module.security.common.impl.accesscontrol.AccessControlSchemaProvider
>
```

Configuration on URL level

The authorization (in terms of Spring security "access management") can be enabled seperately for different url patterns, the request 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:

Extensive example of authorization on URL level.

```
<bean id="FilterSecurityInterceptor" class="org.springframework.security.web.access.intercept.FilterSecurityInterceptor">
    <property name="authenticationManager" ref="AuthenticationManager"/>
    <property name="accessDecisionManager" ref="FilterAccessDecisionManager"/>
    <property name="securityMetadataSource">
      <security:filter-security-metadata-source use-expressions="true">
        <security:intercept-url pattern="/" access="isAnonymous()"/>
        <security:intercept-url pattern="/index.jsp" access="isAnonymous()"/>
        <security:intercept-url pattern="/security/login*" access="isAnonymous()"/>
        <security:intercept-url pattern="/j_spring_security_login*" access="isAnonymous()"/>
        <security:intercept-url pattern="/j_spring_security_logout*" access="isAnonymous()"/>
        <security:intercept-url pattern="/services/rest/security/currentuser/" access="isAnonymous() or
 isAuthenticated()"/>
        <security:intercept-url pattern="/**" access="isAuthenticated()"/>
      </security:filter-security-metadata-source>
    </property>
```

```
</bean>

<bean id="FilterAccessDecisionManager" class="org.springframework.security.access.vote.UnanimousBased">
    <constructor-arg>
      <list>
        <bean class="org.springframework.security.web.access.expression.WebExpressionVoter"/>
      </list>
    </constructor-arg>
</bean>
```

Configuration on Java Method level

As state of the art oasp4j will focus on role-based authorization to cope with authorization for executing use case of an application. We will use the JSR250 annotations, mainly `@RolesAllowed`, for authorizing method calls against the permissions defined in the annotation body. This has to be done for each use-case method in logic layer. Here is an example:

```
public class UcFindTableImpl extends AbstractTableUc implements UcFindTable {

  @RolesAllowed(PermissionConstants.FIND_TABLE)
  public TableEto findTable(Long id) {

    return getBeanMapper().map(getTableDao().findOne(id), TableEto.class);
  }
}
```

Now this method can only be called if a user is logged-in that has the permission `FIND_TABLE`.

Check Data-based Permissions

Currently, we have no best practices and reference implementations to apply permission based access on an application’s data. Nevertheless, this is a very important topic due to the high standards of data privacy & protection especially in germany. We will further investigate this topic and we will adress it in one of the next releases. For further tracking have a look at issue #125.

4.2.3 Vulnerabilities and Protection

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. The following table illustrates common threats and contains links to the solutions and protection-mechanisms provided by the OASP:

Table 4.5. Security threats and protection-mechanisms

Thread	Protection	Link to details
A1 Injection	validate input, escape output, use proper frameworks	data-access-layer guide
A2 Broken Authentication and Session Management	encrypt all channels, use a central identity management with strong password-policy	Authentication
A3 XSS	prevent injection (see A1) for HTML, JavaScript and CSS and understand same-origin-policy	client-layer

Thread	Protection	Link to details
A4 Insecure Direct Object References	Using direct object references (IDs) only with appropriate authorization	See issue #86
A5 Security Misconfiguration	Use OASP application template and guides to avoid	application template
A6 Sensitive Data Exposure	Use secured exception facade, design your data model accordingly	REST exception handling
A7 Missing Function Level Access Control	Ensure proper authorization for all use-cases, use `@DenyAll` als default to enforce	Method authorization
A8 CSRF	secure mutable service operations with an explicit CSRF security token sent in HTTP header and verified on the server	service-layer security
A9 Using Components with Known Vulnerabilities	subscribe to security newsletters, recheck products and their versions continuously, use OASP dependency management	CVE newsletter
A10 Unvalidated Redirects and Forwards	Avoid using redirects and forwards, in case you need them do a security audit on the solution.	OASP proposes to use rich-clients (SPA/RIA). We only use redirects for login in a safe way.
Log-Forging	Escape newlines in log messages	logging security

Tool for testing your web application against 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

For flexibility an application needs to be configurable. So far there is no general standard for configurations and how to structure and name configuration files. Therefore the OASP4J gives you detailed instructions and best-practices how to deal with configurations and manage them. This prevents chaos and leads to success in maintenance and operations. In general we distinguish the following kinds of configurations that are explained in the following sections:

- application configuration maintained by developers
- environment configuration maintained by operators
- business configuration maintained by business administrators

4.4.1 Application Configuration

The application configuration contains all internal settings of the application (component implementations, integration, database mappings, etc.) and is maintained by the application developers. As we make intensive use of the spring framework this is especially about Spring XML configuration files. According to dependency-injection we only use Spring XML to configure specific integrations and general setup. The application configuration resides in the folder app.

4.4.1.1 beans-application.xml

As you can see in our configuration file layout we use beans-application.xml to bundle the configuration of the entire application. This file represents the root of the application configuration and imports all other configurations for different aspects (layers and technical components).

[[guide-configuration_beans-[aspect].xml]] === beans-[aspect].xml Every aspects of an application configuration lies in an own folder, named after the aspect. The folder contains the configuration file beans-aspect.xml, which is is the root configuration file for this aspect.

Every configuration beans-aspect.xml can again be divided into additional configurations located in the same aspect folder.

Example of an aspect structure:

- aspect
  - beans-aspect.xml
  - …

For further explanation of the Spring XML configuration consult spring documentation.

You can find a more comprehensive example of the structure of an application configuration at the configuration file layout example.

4.4.1.2 logback.xml

The main configuration file logback.xml resides directly in src/main/resources and is not located in the app folder. The logging configuration is explained in detail in the logging guide.

4.4.2 Environment Configuration

The environment configuration contains configuration parameters (port numbers, host names, passwords, logins, timeouts, certificates, etc.) specific for the different environments. These are under the control of the operators responsible for the application. As we suggest to only have one application per servlet-container (tomcat) the environment specific configuration is to be placed into tomcat/lib/config/env/application.properties so it will be found by the classloader of the web-application (the deployed WAR file). In this application.properties you only define the minimum properties that are environment specific and inherit everything else from the application-default.properties, which itself resides in the app folder.

4.4.2.1 application.properties

The file application.properties and application-default.properties can define various properties. The common defaults for the application shall be defined in application-default.properties that is bundled and deployed with the application. The application.properties file itself must NOT declare all properties of the application-default.properties again.

These properties are explained in the corresponding configuration sections of the guides for each topic:

- persistence configuration
- service configuration

4.4.3 Business Configuration

The business configuration contains all business configuration values of the application, which can be edited by administrators through the GUI. The business configuration values are stored in the database in key/value pairs.

The database table business_configuration has the following columns:

- ID
- Property name
- Property type (Boolean, Integer, String)
- Property value
- Description

According to the entries in this table, the GUI shows a generic form to change business configuration. The hierachy of the properties determines the place in the GUI, so the GUI bundles properties from the same hierarchy level and name. Boolean values are shown as checkboxes, integer and string values as text fields. The properties are read and saved in a typed form, an error is raised if you try to save a string in an integer property for example.

We recommend the following base layout for the hierarchical business configuration:

component.[subcomponent].[subcomponent].propertyname

4.4.4 Configuration Files

We read configurations from the java classpath to make things flexible and easy. In your application project you will find and add the configurations below the directory src/main/resources/.

The OASP defines a generic layout for configurations on the classpath in the following form:

- src/main/resources

- config/
  - env (not delivered with application, only in * src/test/resources/)
    - application.properties
    - …
  - app
    - common
      - beans-common.xml
      - …
    - logic
      - beans-logic.xml
      - …
    - persistence
      - beans-jpa.xml
      - beans-persistence.xml
      - NamedQueries.xml
      - …
    - security
      - access-control-schema.xml
      - beans-security.xml
      - …
    - beans-application.xml
    - application-default.properties
    - …
- logback.xml

4.5 Validation

Validation is about checking syntax and semantics of input data. Invalid data is rejected by the application. Therefore validation is required in multiple places of an application. E.g. the GUI will do validation for usability reasons to assist the user, early feedback and to prevent unnecessary server requests. On the server-side validation has to be done for consistency and security.

In general we distinguish these forms of validation:

- *stateless validation* will produce the same result for given input at any time (for the same code/ release).
- *stateful validation* is dependent on other states and can consider the same input data as valid in once case and as invalid in another.

4.5.1 Stateless Validation

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

4.5.1.1 Example

A description of how to enable BV can be found in the relevant Spring documentation. For a quick summary follow these steps:

- Make sure that hibernate-validator is located in the classpath by adding a dependency to the pom.xml.

```
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-validator</artifactId>
    </dependency>
```

- Define Spring beans:

```
<bean id="validator" class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>
<bean class="org.springframework.validation.beanvalidation.MethodValidationPostProcessor"/>
```

- Add the @Validated annotation to the implementation (spring bean) to be validated. For methods to validate go to their declaration and add constraint annotations to the method parameters.
  - @Valid annotation to the arguments to validate (if that class itself is annotated with constraints to check).
  - @NotNull for required arguments.
  - Other constraints (e.g. @Size) for generic arguments (e.g. of type String or Integer). However, consider to create custom datatypes and avoid adding too much validation logic (especially redundant in multiple places).

OffermanagementRestServiceImpl.java.

```
@Validated
public class OffermanagementRestServiceImpl implements RestService {
  ...
  public void createOffer(@Valid OfferEto offer) {
  ...
```

- Finally add appropriate validation constraint annotations to the fields of the ETO class.

OfferEto.java.

```
  @NotNegativeMoney
  private Money currentPrice;
```

A list with all bean validation constraint annotations available for hibernate-validator can be found here. In addition it is possible to configure custom constraints. Therefor it is neccessary to implement a annotation and a corresponding validator. A description can also be found in the Spring documentation or with more details in the hibernate documentation.

4.5.1.2 GUI-Integration

TODO

4.5.1.3 Cross-Field Validation

BV has poor support for this. Best practice is to create and use beans for ranges, etc. that solve this. A bean for a range could look like so:

```
public class Range<V extends Comparable<V>> {

  private V min;
  private V max;

  public Range(V min, V max) {

    super();
    if ((min != null) && (max != null)) {
      int delta = min.compareTo(max);
      if (delta > 0) {
        throw new ValueOutOfRangeException(null, min, min, max);
      }
    }
    this.min = min;
    this.max = max;
  }

  public V getMin() ...
  public V getMax() ...
```

4.5.2 Stateful Validation

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. An example is the deletion of a table in the example application. Here the state of the table must be checked first:

UcManageTableImpl.java.

```
  public boolean deleteTable(Long tableId) {

    TableEntity table = getTableDao().find(tableId);
    if (!table.getState().isFree()) {
      throw new IllegalEntityStateException(table, table.getState());
    }
    getTableDao().delete(table);
    return true;
  }
```

Implementing this small check with BV would be a lot more effort.

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 AOP Key Principles

We follow these principles:

- We use spring AOP based on dynamic proxies (and fallback to cglib).
- We avoid AspectJ and other mighty and complex AOP frameworks whenever possible
- We only use AOP where we consider it as necessary (see below).

4.6.2 AOP Usage

We recommend to use AOP with care but we consider it established for the following cross cutting concerns:

- Transaction-Handling
- Authorization
- Trace-Logging (for testing and debugging)
- Exception facades for services but only if no other solution is possible (use alternatives such as JAX-RS provider instead).

4.7 Exception Handling

4.7.1 Exception Principles

For exceptions we follow these principles:

- We only use exceptions for *exceptional* situations and not for programming control flows, etc. Creating an exception in Java is expensive and hence you should not do it just for testing if something is present, valid or permitted. In the latter case design your API to return this as a regular result.
- We use unchecked exceptions (RuntimeException)
- We distinguish *internal exceptions* and *user exceptions*:
  - Internal exceptions have technical reasons. For unexpected and exotic situations it is sufficient to throw existing exceptions such as IllegalStateException. For common scenarios a own exception class is reasonable.
  - User exceptions contain a message explaining the problem for end users. Therefore we always define our own exception classes with a clear, brief but detailed message.
- Our own exceptions derive from an exception base class supporting
  - unique ID per instance
  - Error code per class
  - message templating (see I18N)
  - distinguish between *user exceptions* and *internal exceptions*

All this is offered by mmm-util-core that we propose as solution.

4.7.2 Exception Example

Here is an exception class from our sample application:

```
public class IllegalEntityStateException extends RestaurantBusinessException {

  private static final long serialVersionUID = 1L;

  public IllegalEntityStateException(RestaurantEntity entity, Object state) {

    super(createBundle(NlsBundleRestaurantRoot.class).errorIllegalEntityState(entity, state));
  }

  public IllegalEntityStateException(RestaurantEntity entity, Object currentState, Object newState) {

    super(createBundle(NlsBundleRestaurantRoot.class).errorIllegalEntityStateChange(entity,
 currentState, newState));
  }
}
```

The message templates are defined in the interface NlsBundleRestaurantRoot as following:

```
public interface NlsBundleRestaurantRoot extends NlsBundle {

  @NlsBundleMessage("The entity {entity} is in state {state}!")
  NlsMessage errorIllegalEntityState(@Named("entity") Object entity, @Named("state") Object state);
  @NlsBundleMessage("The entity {entity} in state {currentState} can not be changed to state
 {newState}!")
  NlsMessage errorIllegalEntityStateChange(@Named("entity") Object entity, @Named("currentState") Object
 currentState, @Named("newState") Object newState);
}
```

4.7.3 Handling Exceptions

For catching and handling exceptions we follow these rules:

- We do not catch exceptions just to wrap or to re-throw them.
- If we catch an exception and throw a new one, we always **have** to provide the original exception as cause to the constructor of the new exception.
- At the entry points of the application (e.g. a service operation) we have to catch and handle all throwables. This is done via the *exception-facade-pattern* via an explicit facade or aspect. The OASP4J already provides ready-to-use implementations for this such as RestServiceExceptionFacade. The exception facade has to…
  - log all errors (user errors on info and technical errors on error level)
  - convert the error to a result appropriable for the client and secure for Sensitive Data Exposure. Especially for security exceptions only a generic security error code or message may be revealed but the details shall only be logged but **not** be exposed to the client. All *internal exceptions* are converted to a generic error with a message like:

    > An unexpected technical error has occurred. We apologize any inconvenience. Please try again later.

4.8 Internationalization

Internationalization (I18N) is about writing code independent from locale-specific informations. For I18N of text messages we are suggesting mmm native-language-support.

4.9 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.

4.9.1 JAXB

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

4.9.1.1 JAXB and Inheritance

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

4.9.1.2 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

4.10 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.

4.10.1 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:

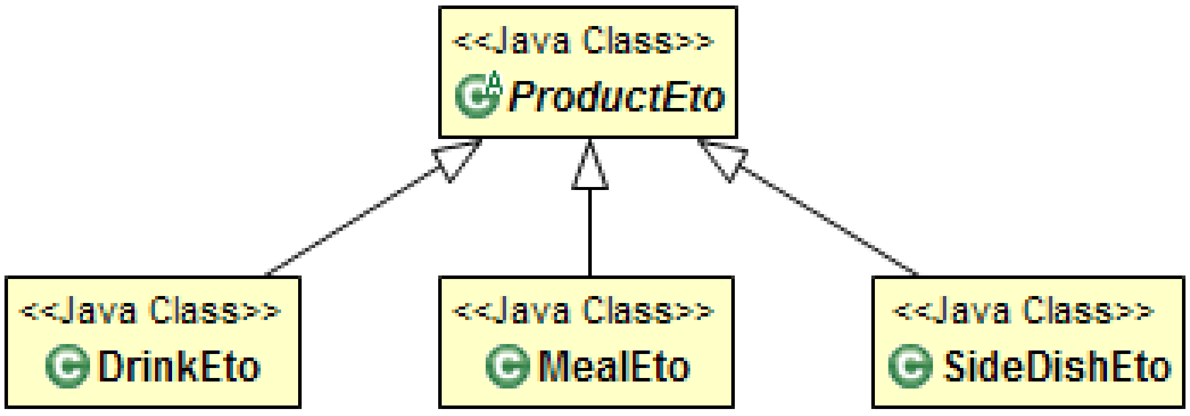


Figure 4.2. Transfer-Objects using Inheritance

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.

4.10.2 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!

4.11 Testing

4.11.1 General best practices

For testing please follow our general best practices:

- Tests should have a clear goal that should also be documented.
- Tests have to be classified into different types that are explained in the following section.
- Tests should follow a clear naming convention.
- Automated tests need to properly assert the result of the tested operation(s) in a reliable way. E.g. avoid stuff like assertEquals(42, service.getAllEntities()) or even worse tests that have no assertion at all (might still be reasonable to test that an entire configuration setup such as spring config of application is intact).
- Tests need to be independent of each other. Never write test-cases or tests (in Java @Test methods) that depend on another test to be executed before.
- In general tests should NOT have side-effects. Otherwise following tests may fail and it is very hard to trace down the actual reason of the error. If changing the state (e.g. database) is required for some advanced integration test (e.g. only real commit will trigger and test DB constraints) then it has to be ensured that the test-case performs a cleanup (e.g. in @AfterClass). A best practice for integration testing is to combine operations that are neutral in combination (e.g. create entity, read again by id, change and update entity, search entity by updated values, delete entity). However, you need to ensure that the transaction only gets committed if this entire combination succeeds.
- Plan your tests and testdata management properly before implementing.
- Instead of having a too strong focus on test coverage better ensure you have covered your critical core functionality properly and review the code including tests.
- Test code shall NOT be seen as second class code. You shall consider design, architecture and code-style also for your test code but do not over-engineer it.
- Test automation is good but should be considered in relation to cost per use. Creating full coverage via UI integration tests can cause a massive amount of test-code that can turn out as a huge maintenance hell. Always consider all aspects including product life-cycle, criticality of use-cases to test, and variability of the aspect to test (e.g. UI, test-data).
- Use continuous integration and establish that the entire team wants to have clean builds and running tests.

4.11.2 Test Automation

For test automation we use JUnit. Further we are using according enhancements and addons such as harmcrest, spring-test or mockito.

4.11.3 Test Levels

4.11.3.1 Module Testing

For module testing means to test single classes or tiny modules of the code-base in isolation.

4.11.3.2 Component Testing

Component testing means to test a business component with its component part for logic and data-access layer. Tests invoke the logic layer but data-access layer will be tested implicitly. If the business component invokes external systems, these must be replaced by mocks.

4.11.3.3 Integration Testing

Integration testing means to test the integrated application. Therefore the application is build as deployable file and started in a container. Then tests are performed via HTTP on services of the service layer. External systems have to be mocked.

4.11.4 Test Coverage

We are using tools (SonarQube/Jacoco) to measure the coverage of the tests. Please always keep in mind that the only reliable message of a code coverage of X% is that (100-X)% of the code is entirely untested. It does not say anything about the quality of the tests or the software though it often relates to it.

4.12 Transfer-Objects

The technical data model is defined in form of persistent entities. However, passing persistent entities via *call-by-reference* across the entire application will soon cause problems:

- Changes to a persistent entity are directly written back to the persistent store when the transaction is committed. When the entity is send across the application also changes tend to take place in multiple places endangering data sovereignty and leading to inconsistency.
- You want to send and receive data via services across the network and have to define what section of your data is actually transferred. If you have relations in your technical model you quickly end up loading and transferring way too much data.
- Modifications to your technical data model shall not automatically have impact on your external services causing incompatibilities.

To prevent such problems transfer-objects are used leading to a *call-by-value* model and decoupling changes to persistent entities.

4.12.1 Business-Transfer-Objects

For each persistent entity we create or generate a corresponding *entity transfer object* (ETO) that has the same properties except for relations. In order to centralize the properties (getters and setters with their javadoc) we use a common interface for the entity and its ETO.

If we need to pass an entity with its relation(s) we create a corresponding *composite transfer object* (CTO) that only contains other transfer-objects or collections of them. This pattern is illustrated by the following UML diagram from our sample application.

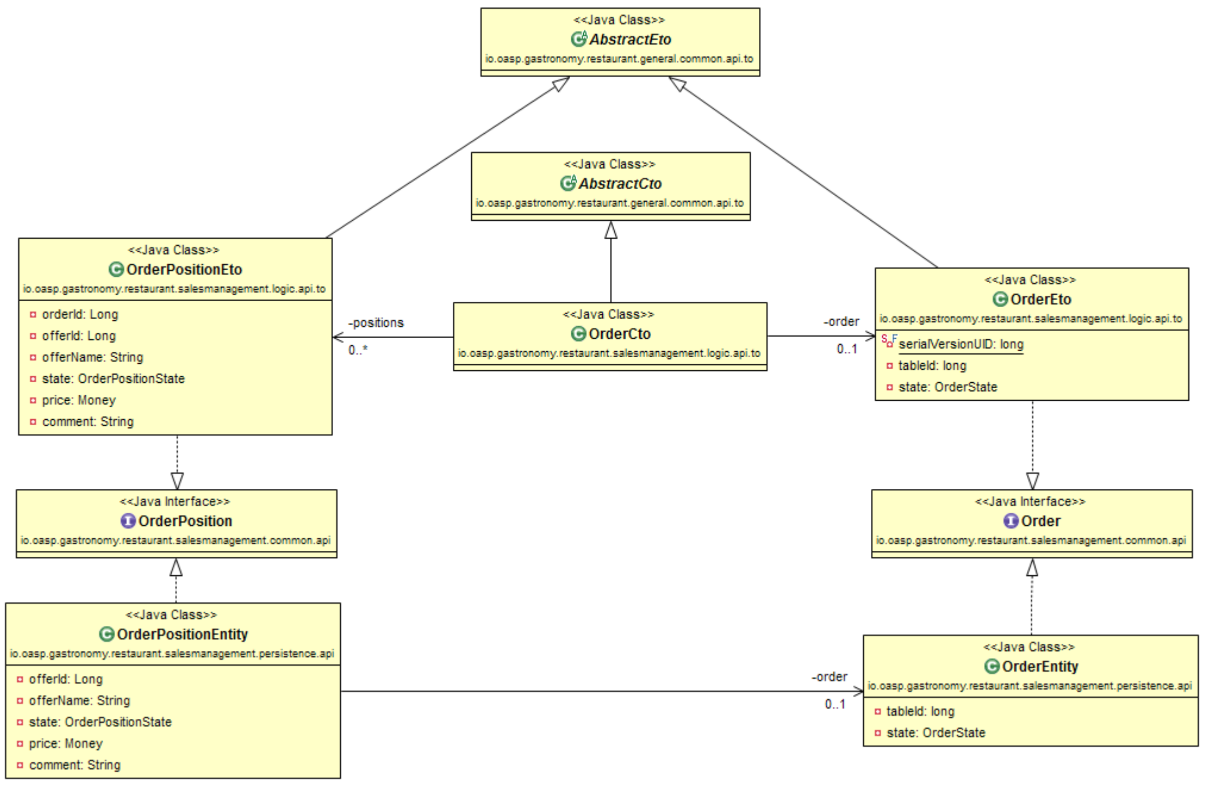

<<Java Class>>
AbstractEto
io.oasp.gastronomy.restaurant.general.common.api.to
<<Java Class>>
AbstractCto
io.oasp.gastronomy.restaurant.general.common.api.to
<<Java Class>>
OrderPositionEto
io.oasp.gastronomy.restaurant.salesmanagement.logic.api.to
orderId: Long
offerId: Long
offerName: String
state: OrderPositionState
price: Money
comment: String
-positions
0..*
<<Java Class>>
OrderCto
io.oasp.gastronomy.restaurant.salesmanagement.logic.api.to
-order
0..1
<<Java Class>>
OrderEto
io.oasp.gastronomy.restaurant.salesmanagement.logic.api.to
serialVersionUID: long
tableId: long
state: OrderState
<<Java Interface>>
OrderPosition
io.oasp.gastronomy.restaurant.salesmanagement.common.api
<<Java Interface>>
Order
io.oasp.gastronomy.restaurant.salesmanagement.common.api
<<Java Class>>
OrderPositionEntity
io.oasp.gastronomy.restaurant.salesmanagement.persistence.api
offerId: Long
offerName: String
state: OrderPositionState
price: Money
comment: String
-order
0..1
<<Java Class>>
OrderEntity
io.oasp.gastronomy.restaurant.salesmanagement.persistence.api
tableId: long
state: OrderState

Figure 4.3. ETOs and CTOs

Finally, there are typically transfer-objects for data that is never persistent. A common example are search criteria objects (derived from SearchCriteriaTo in our sample application).

The logic layer defines these transfer-objects (ETOs, CTOs, etc.) and will only pass such objects instead of persistent entities.

4.12.2 Service-Transfer-Objects

If we need to do service versioning and support previous APIs or for external services with a different view on the data, we create separate transfer-objects to keep the service API stable (see service layer).

4.13 Bean-Mapping

For decoupling you sometimes need to create separate objects (beans) for a different view. E.g. for an external service you will use a transfer-object instead of the persistence entity so internal changes to the entity do not implicitly change or break the service.

Therefore you have the need to map similar objects what creates a copy. This also has the benefit that modifications to the copy have no side-effect on the original source object. However, to implement such mapping code by hand is very tedious and error-prone (if new properties are added to beans bot not to mapping code):

```
public PersonTo mapPerson(PersonEntity source) {
  PersonTo target = new PersonTo();
  target.setFirstName(source.getFirstName());
  target.setLastName(source.getLastName());
  ...
  return target;
}
```

Therefore we are using a BeanMapper for this purpose that makes our lives a lot easier.

4.13.1 Bean-Mapper Dependency

To get access to the BeanMapper we use this dependency in our POM:

```
    <dependency>
      <groupId>io.oasp.java</groupId>
      <artifactId>oasp4j-beanmapping</artifactId>
    </dependency>
```

4.13.2 Bean-Mapper Usage

Then we can get the BeanMapper via dependency-injection what we typically already provide by an abstract base class (e.g. AbstractUc). Now we can solve our problem very easy:

```
PersonEntity person = ...;
...
return getBeanMapper().map(person, PersonTo.class);
```

There is also additional support for mapping entire collections.

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

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.

4.14.3 Datatypes in Transfer-Objects

4.14.3.1 XML

For mapping datatypes with JAXB see XML guide.

4.14.3.2 JSON

For mapping datatypes from and to JSON see JSON custom mapping.

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