Open Application Standard Platform for Java V2.0.0

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. This view is not explored further in this guide.

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.
- batch layer for exposing functionality in batch-processes (e.g. mass imports).
- 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

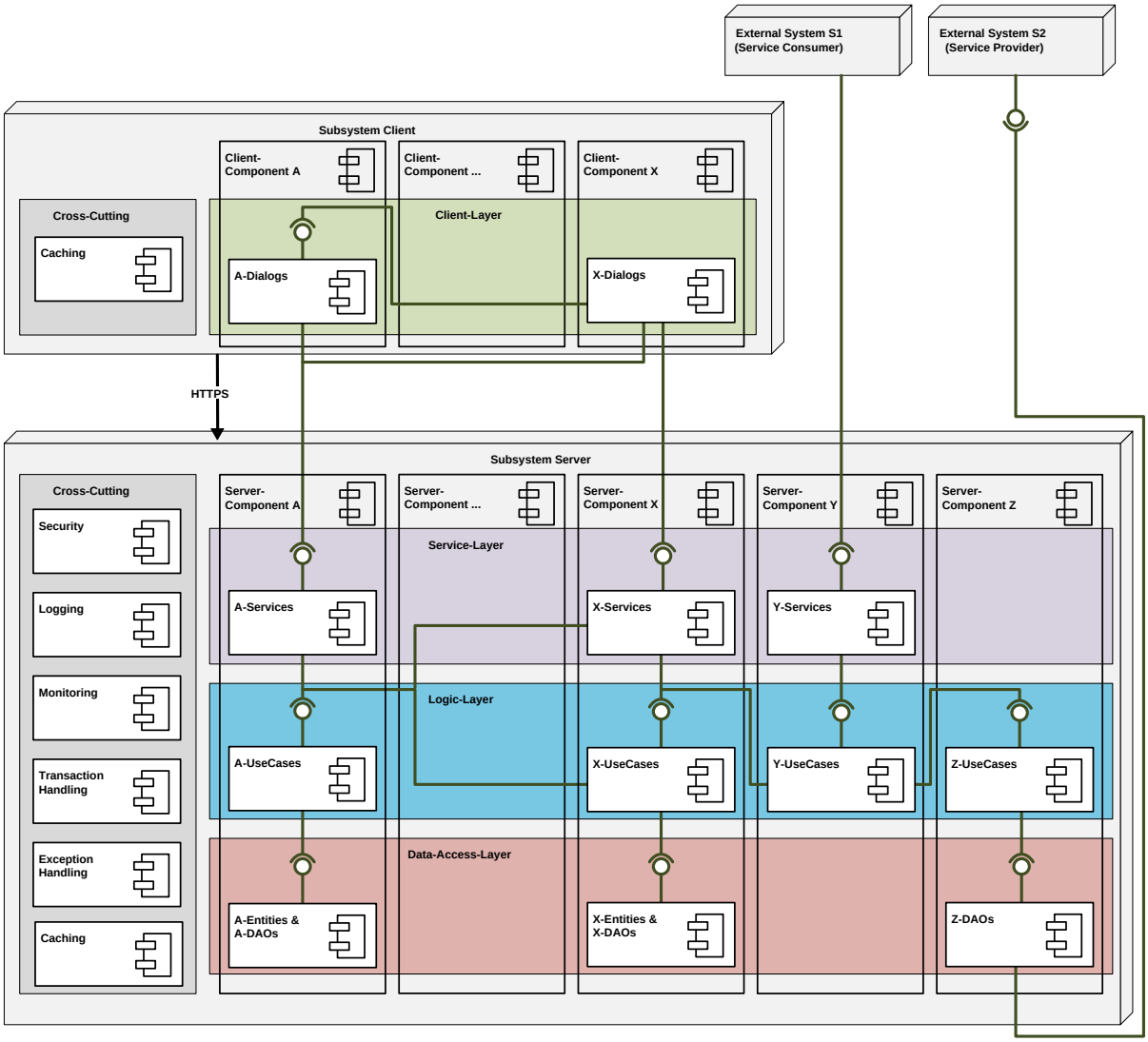


Figure 1.1. Technical Reference Architecture

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	Oracle JDK
runtime	servlet-container	JEE	tomcat
component management	dependency injection	JSR330 & JSR250	spring
configuration	framework	-	spring-boot
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

Topic	Detail	Standard	Suggested implementation
validation	framework	beanvalidation/JSR303	hibernate-validator
security	Authentication & Authorization	JAAS	spring-security
monitoring	framework	JMX	spring
monitoring	HTTP Bridge	HTTP & JSON	jolokia
AOP	framework	dynamic proxies	spring AOP

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 owning the code following	io.oasp

Segment	Description	Example
	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

Currently we focus on web-clients. So far we offer a responsive Java Script based client provided by OASP4js that integrates seamless with OASP-server. A separate guide is provided for OASP4JS.

3.2 Service Layer

The service layer is responsible to expose functionality of the logical layer to external consumers over a network via 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	REST	REST+JSON	REST

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 version number and prefixed with v for version (e.g. v1).
- If we support previous versions we use that version numbers as part of the Java package defining the service API (e.g. com.foo.application.component.service.api.v1)
- We use the version number as part of the service name in the remote URL (e.g. https://application.foo.com/services/rest/component/v1/resource)

- Whenever we need to change the API of a service in an incompatible we increment the version (e.g. v2) as an isolated copy of the previous version of the service. In the implementation of different versions of the same service we can place compatibility code and delegate to the same unversioned use-case of the logic layer whenever possible.
- For maintenance and simplicity we avoid keeping more than one previous version.

3.2.3 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 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. For CRUD operations in REST 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/v1/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/v1/myentity/42

The "pure" REST architecture style is more suitable for creating "scalable" systems on the open web. But for usual business applications its complexity outweight its benefits, therefore the OASP proposes a more "pragmatic" approach to REST services.

On the next table we compare the main differences between the "canonical" REST approach (or RESTful) and the OASP proposal.

Table 3.2. Usage of HTTP methods

HTTP Method	RESTful Meaning	OASP
GET	Read single element. Search on an entity (with parametrized url)	Read a single element.
PUT	Replace entity data. Replace entire collection (typically not supported)	Not used

HTTP Method	RESTful Meaning	OASP
POST	Create a new element in the collection	Create or update an element in the collection. Search on an entity (parametrized post body) Bulk deletion.
DELETE	Delete an entity. Delete an entiry collection (typically not supported)	Delete an entity. Delete an entiry collection (typically not supported)

Please consider these guidelines and rationales: * We use POST on the collection URL for both create and update operations on an entity. This avoids pointless discussions in distinctions between PUT and POST and what to do if a "creation" contains an ID or if an "update" is missing the ID property. * Hence, we do NOT use PUT but always use POST for write operations. As we always have a technical ID for each entity, we can simply distinguish create and update by the presence of the ID property.

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")
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 @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

Starting from CXF 3.0.0 it is possible to enable the auto-discovery of JAX-RS roots and providers thus avoiding having to specify each service bean in the beans-service.xml file.

When the jaxrs server is instantiated all the scanned root and provider beans (beans annotated with `javax.ws.rs.Path` and `javax.ws.rs.ext.Provider`) are configured. The xml configuration still allows us to specify the root path for all endpoints.

```
<jaxrs:server id="CxfRestServices" address="/rest" />
```

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.

Metadata

OASP has support for the following metadata in REST service invocations:

Name	Description	Further information
Correlation ID	A unique identifier to associate different requests belonging to the same session / action	Logging guide
Validation errors	Standardized format for a service to communicate validation errors to the client	Server-side validation is documented in the Validation guide. The protocol to communicate these validation errors to the client is worked on at https://github.com/oasp/oasp4j/issues/218
Pagination	Standardized format for a service to offer paginated access to a list of entities	Server-side support for pagination is documented in the Data-Access Layer Guide.

Recommendations for REST requests and responses

The OASP proposes, for simplicity, a deviation from the REST common pattern:

- Using POST for updates (instead of PUT)
- Using the payload for addressing resources on POST (instead of identifier on the URL)
- Using parametrized POST for searches

This use of REST will lead to simpler code both on client and on server. We discuss this use on the next points.

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.

Unparameterized loading of a single resource

- **HTTP Method**: GET
- **URL example**: /products/123

For loading of a single resource, embed the identifier of the resource in the URL (for example / products/123).

The response contains the resource in JSON format, using a JSON object at the top-level, for example:

```
{
    "name": "Steak",
    "color": "brown"
}
```

Unparameterized loading of a collection of resources

- **HTTP Method**: GET
- **URL example**: /products

For loading of a collection of resources, make sure that the size of the collection can never exceed a reasonable maximum size. For parameterized loading (searching, pagination), see below.

The response contains the collection in JSON format, using a JSON object at the top-level, and the actual collection underneath a result key, for example:

```
{
    "result": [
        {
            "name": "Steak",
            "color": "brown"
        },
        {
            "name": "Broccoli",
            "color": "green"
        }
    ]
}
```

Avoid returning JSON arrays at the top-level, to prevent CSRF attacks (see https://www.owasp.org/index.php/OWASP_AJAX_Security_Guidelines)

Saving a resource

- **HTTP Method**: POST
- **URL example**: /products

The resource will be passed via JSON in the request body. If updating an existing resource, include the resource's identifier in the JSON and not in the URL, in order to avoid ambiguity.

If saving was successful, an empty HTTP 204 response is generated.

If saving was unsuccessful, refer below for the format to return errors to the client.

Parameterized loading of a resource

- **HTTP Method**: POST
- **URL example**: /products/search

In order to differentiate from an unparameterized load, a special *subpath* (for example search) is introduced. The parameters are passed via JSON in the request body. An example of a simple, paginated search would be:

```
{
    "status": "OPEN",
    "pagination": {
        "page": 2,
        "size": 25
    }
}
```

The response contains the requested page of the collection in JSON format, using a JSON object at the top-level, the actual page underneath a result key, and additional pagination information underneath a pagination key, for example:

```
{
    "pagination": {
        "page": 2,
        "size": 25,
        "total": null
    },
    "result": [
        {
            "name": "Steak",
            "color": "brown"
        },
        {
            "name": "Broccoli",
            "color": "green"
        }
    ]
}
```

Compare the code needed on server side to accept this request:

```
@Path("/order")
 @POST
 public List<OrderCto> findOrders(OrderSearchCriteriaTo criteria) {
    return this.salesManagement.findOrderCtos(criteria);
 }
```

With the equivalent code required if doing it the REST way by issuing a GET request:

```
  @Path("/order")
  @GET
  public List<OrderCto> findOrders(@Context UriInfo info) {

    RequestParameters parameters = RequestParameters.fromQuery(info);
    OrderSearchCriteriaTo criteria = new OrderSearchCriteriaTo();
    criteria.setTableId(parameters.get("tableId", Long.class, false));
    criteria.setState(parameters.get("state", OrderState.class, false));
    return this.salesManagement.findOrderCtos(criteria);
  }
```

Pagination details

The client can choose to request a count of the total size of the collection, for example to calculate the total number of available pages. It does so, by specifying the pagination.total property with a value of true.

The service is free to honour this request. If it chooses to do so, it returns the total count as the pagination.total property in the response.

Deletion of a resource

- **HTTP Method**: DELETE
- **URL example**: /products/123

For deletion of a single resource, embed the identifier of the resource in the URL (for example / products/123).

Error results

The general format for returning an error to the client is as follows:

```
{
    "message": "A human-readable message describing the error",
    "code": "A code identifying the concrete error",
    "uuid": "An identifier (generally the correlation id) to help identify corresponding requests in
 logs"
}
```

If the error is caused by a failed validation of the entity, the above format is extended to also include the list of individual validation errors:

```
{
    "message": "A human-readable message describing the error",
    "code": "A code identifying the concrete error",
    "uuid": "An identifier (generally the correlation id) to help identify corresponding requests in
 logs",
    "errors": {
        "property failing validation": [
            "First error message on this property",
            "Second error message on this property"
        ],
        // ....
    }
}
```

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:

- Firefox: httprequester (or poster)
- Chrome: postman (advanced-rest-client)

3.2.4.2 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.

3.2.5 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

Your services are the major entry point to your application. Hence security considerations are important here.

3.2.6.1 CSRF

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.

For testing in development environment the CSRF protection can be disabled using the JVM option -DCsrfDisabled=true when starting the 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 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.

If the implementation of the component interface gets too complex it is recommended to further sub-divide it in separate use-case-interfaces to be aggregated in the main component interface. This suits for better maintainability.

First we create an interface 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 only access objects from other business components in the (transitive) dependencies of the declaring business component. Here is an example of a use case interface:

```
public interface StaffManagement {

  StaffMemberEto getStaffMemberByLogin(String login);

  StaffMemberEto getStaffMember(Long id);

  ...
}
```

3.3.2 Component Implementation

The implementation of the use case 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.

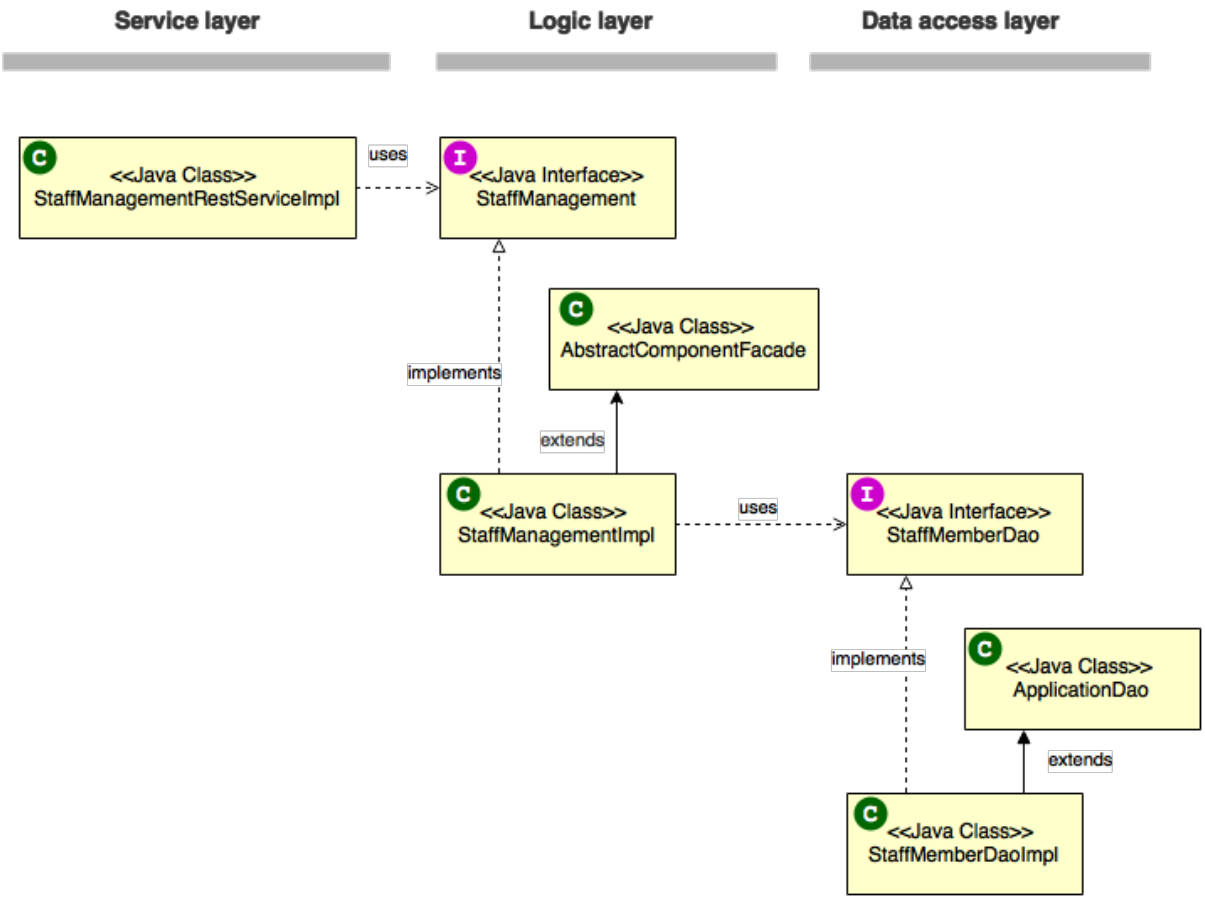
Within a use-case implementation, entities are mapped via a BeanMapper to persistent entities. Let's take a quick look at some of the StaffManagement methods:

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

public class StaffManagementImpl extends AbstractComponentFacade implements StaffManagement {

  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>
```

Below, a class diagram illustrating the pattern is shown (here: the `StaffManagement` business component):



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

3.3.3 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.

3.3.4 Security

The logic layer is the heart of the application. It is also responsible for authorization and hence security is important here.

3.3.4.1 Direct Object References

A security threat are Insecure Direct Object References. This simply gives you two options:

- avoid direct object references at all
- ensure that direct object references are secure

Especially when using REST, direct object references via technical IDs are common sense. This implies that you have a proper in place. This is especially tricky when your authorization does not only rely on the type of the data and according static permissions but also on the data itself. Vulnerabilities for this threat can easily happen by design flaws and inadvertence. Here an example from our sample application:

We have a generic use-case to manage BLOBs. In the first place it makes sense to write a generic REST service to laod and save these BLOBs. However, the permission to read or even update such BLOB depend on the business object hosting the BLOB. Therefore such a generic REST service would open the door for this OWASP A4 vulnerability. To solve this in a secure way you need individual services for each hosting business object to manage the linked BLOB. There you have to check permissions based on the parent business object. In this example the ID of the BLOB would be the direct object reference and the ID of the business object (and a BLOB property indicator) would be the indirect object reference.

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 column of that table. An persistent entity instance then represents a row of the database table.

A Simple Entity

The following listing shows a simple example:

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

  private String text;

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

  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).

Pagination

The OASP provides the method findPaginated in AbstractGenericDao that executes a given query (for now only QueryDSL is supported) with pagination parameters based on SearchCriteriaTo. So all you need to do is derive your individual search criteria objects from SearchCriteriaTo, prepare a QueryDSL-query with the needed custom search criterias, and call findPaginated. Here is an example from our sample application:

```
@Override
public PaginatedListTo<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));
  }

  return findPaginated(criteria, query, alias);
}
```

Then the query allows pagination by setting pagination.size (SearchCriteriaTo.getPagination().setSize(Integer)) to the number of hits per page and pagination.page (SearchCriteriaTo.getPagination().setPage(int)) to the desired page. If you allow the client to specify pagination.size, it is recommended to limit this value on the server side (SearchCriteriaTo.limitMaximumPageSize(int)) to prevent performance problems or DOS-attacks. If you need to also return the total number of hits available, you can set SearchCriteria.getPagination().setTotal(boolean) to true.

Pagination example

For the table entity we can make a search request by accessing the REST endpoint with pagination support like in the following examples:

```
POST oasp4j-sample-server/services/rest/tablemanagement/v1/table/search
{
  "pagination": {
    "size":2,
    "total":true
  }
}

//Response
{
    "pagination": {
        "size": 2,
        "page": 1,
        "total": 11
    },
    "result": [
        {
            "id": 101,
            "modificationCounter": 1,
            "revision": null,
            "waiterId": null,
            "number": 1,
            "state": "OCCUPIED"
        },
```

```
        {
            "id": 102,
            "modificationCounter": 1,
            "revision": null,
            "waiterId": null,
            "number": 2,
            "state": "FREE"
        }
    ]
}
```

> **Note**
>
> as we are requesting with the total property set to true the server responds with the total count of rows for the query.

For retrieving a concrete page, we provide the page attribute with the desired value. Here we also left out the total property so the server doesn't incur on the effort to calculate it:

```
POST oasp4j-sample-server/services/rest/tablemanagement/v1/table/search
{
  "pagination": {
    "size":2,
    "page":2
  }
}

//Response

{
    "pagination": {
        "size": 2,
        "page": 2,
        "total": null
    },
    "result": [
        {
            "id": 103,
            "modificationCounter": 1,
            "revision": null,
            "waiterId": null,
            "number": 3,
            "state": "FREE"
        },
        {
            "id": 104,
            "modificationCounter": 1,
            "revision": null,
            "waiterId": null,
            "number": 4,
            "state": "FREE"
        }
    ]
}
```

Query Meta-Parameters

Queries can have meta-parameters and the OASP currently provides support for *timeout*. The OASP provides the method applyCriteria in AbstractGenericDao that applies meta-parameters to a query based on SearchCriteriaTo. If you already use the pagination support (see above), you do not need to call applyCriteria manually, as it is called internally by findPaginated.

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? To take influence besides placing collections take a look at GenericDao.forceIncrementModificationCounter.

- 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 Database Auditing

See auditing guide.

3.4.1.9 Testing Entities and DAOs

See testing guide.

3.4.1.10 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 which 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

See database migration guide.

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.

3.5 Batch Layer

We understand batch processing as bulk-oriented, non-interactive, typically long running execution of tasks. For simplicity we use the term batch or batch job for such tasks in the following documentation.

OASP uses Spring Batch as batch framework.

This guide explains how Spring Batch is used in OASP applications. Please note that it is not jet fully consistent concerning batches with the sample application. You should adhere to this guide by now.

3.5.1 Batch architecture

In this chapter we will describe the overall architecture (especially concerning layering) and how to administer batches.

3.5.1.1 Layering

Batches are implemented in the batch layer. The batch layer is responsible for batch processes, whereas the business logic is implemented in the logic layer in. Compared to the service layer you may understand the batch layer just as a different way of accessing the business logic. From a component point of view each batch is implemented as a subcomponent in the corresponding business component. The business component is defined by the business architecture.

Let’s make an example for that. The sample application implements a batch for exporting bills. This bill-export-batch belongs to the salesmanagement business component. So the bill-export-batch is implemented in the following package:

```
<basepackage>.salesmanagement.batch.impl.billexport.*
```

Batches should invoke use cases in the logic layer for doing their work. Only "batch specific" technical aspects should be implemented in the batch layer.

Example: For a batch, which imports product data from a CSV file this means that all code for actually reading and parsing the CSV input file is implemented in the batch layer. The batch calls the use case "create product" in the logic layer for actually creating the products for each line read from the CSV input file.

Accessing data access layer

In practice it is not always appropriate to create use cases for every bit of work a batch should do. Instead, the data access layer can be used directly. An example for that is a typical batch for data retention which deletes out-of-time data. Often deleting out-dated data is done by invoking a single SQL statement. It is appropriate to implement that SQL in a DAO method and call this method it directly from the batch. But be careful that this pattern is a simplification which could lead to business logic cluttered in different layers which reduces maintainability of your application. It is a typical design decision you have to take when designing your specific batches.

3.5.1.2 Batch administration and execution

Starting and Stopping Batches

Spring Batch provides a simple command line API for execution and parameterization of batches, the `CommandLineJobRunner`.

Each batch runs as a "simple" standalone process (instantiating a new JVM and creating a new ApplicationContext).

Starting a Batch Job

For starting a batch job, the following parameters are required:

jobPath

The location of the XML file that will be used to create an ApplicationContext. This file should contain everything needed to run the job.

jobName

The name of the job to be run.

These arguments must be passed in with the path first and the name second. All arguments after these are considered to be JobParameters and must be in the format of 'name=value':

```
bash$ java CommandLineJobRunner /config/app/batch/beans-billexport.xml billExportJob -
outputFile=file:/.../out.csv date(date)=2015/12/20
```

The date parameter will be explained in the section on parameters.

Note that when a batch is started with the same parameters as a previous execution of the same batch job, the new execution is considered a restart, see restarts for further details. Parameters starting with a "-" are ignored when deciding whether an execution is a restart or not (so called no identifying parameters).

When trying to restart a batch that was already complete (or has been abandon), there will either be an exception (message: "A job instance already exists and is complete for parameters={…}. If you want to run this job again, change the parameters.") or the batch will simply do nothing (might happen when no or only non identifying parameters are set; in this case the console log contains the following message for every step: "Step already complete or not restartable, so no action to execute: …").

Restarting a Job

As an alternative to starting the batch with the same parameters once more for restarting, the following command can be used, which will automatically determine the parameters of the last failed execution of the batch:

```
bash$ java CommandLineJobRunner /config/app/batch/beans-billexport.xml –restart billExportJob
```

Note that this command might not work as expected when marking a batch job with restartable="false" (see restarts), which is why you should rather use the start operation instead.

Stopping a Job

The command line option to stop a running execution is as follows:

```
bash$ java CommandLineJobRunner /config/app/batch/beans-billexport.xml –stop billExportJob
```

Note that the job is not shutdown immediately, but might actually take some time to stop.

Aborting a Job

To avoid that a batch job is restarted, you can abandon it.

The command line option to abandon all stopped or failed executions of a batch job is as follows:

```
bash$ java CommandLineJobRunner /config/app/batch/beans-billexport.xml – abandon billExportJob
```

Scheduling

In real world scheduling of batches is not as simple as it first might look like.

- Multiple batches have to be executed in order to achieve complex tasks. If one of those batches fails the further execution has to be stopped and operations should be notified for example.
- Input files or those created by batches have to be copied from one node to another.
- Scheduling batch executing could get complex easily (quarterly jobs, run job on first workday of a month, …)

For OASP we propose the batches themselves should not mess around with details of batch administration. Likewise your application should not do so.

Batch administration should be externalized to a dedicated batch administration service or scheduler. This service could be a complex product or a simple tool like cron. We propose Rundeck as an open source job scheduler.

This gives full control to operations to choose the solution which fits best into existing administration procedures.

3.5.2 Implementation

In this chapter we will describe how to properly setup and implement batches.

3.5.2.1 Main Challenges

At a first glimpse, implementing batches is much like implementing a backend for client processing. There are, however, some points at which batches have to be implemented totally different. This is especially true if large data volumes are to be processed.

The most important points are:

Transaction handling

For processing request made by clients there is usually one transaction for each request. If anything goes wrong, the transaction is rolled back and all changes are reverted.

A naive approach for batches would be to execute a whole batch in one single transaction so that if anything goes wrong, all changes are reverted and the batch could start from scratch. For processing large amounts of data, this is technically not feasible, because the database system would have to be able to undo every action made within this transaction. And the space for storing the undo information needed for this (the so called "undo tablespace") is usually quite limited.

So there is a need of short running transactions. To help programmers to do so, Spring Batch offers the so called chunk processing which will be explained here.

Restarting Batches

In client processing mode, when an exception occurs, the transaction is rolled back and there is no need to worry about data inconsistencies.

This is not true for batches however, due to the fact that you usually can't have just one transaction. When an unexpected error occurs and the batch aborts, the system is in a state where the data is partly processed and party not and there needs to be some sort of plan how to continue from there.

Even if a batch was perfectly reliable, there might be errors that are not under the control of the application, e.g. lost connection to the database, so that there is always a need for being able to restart.

The section on restarts describes how to design a batch that is restartable. What's important is that a programmer has to invest some time upfront for a batch to be able restart after aborts.

Exception handling in Batches

The problem with exception handling is that e.g. a single record can cause a whole batch to fail and many records will remain unprocessed. In contrast to this, in client processing mode when processing fails this usually affects only one user.

To prevent this situation, Spring Batch allows to skip data when certain exceptions occur. However, the feature should not be misused in a way that you just skip all exceptions independently of their cause.

So when implementing a batch, you should think about what exceptional situations might occur and how to deal with that and weather it is okay to skip those exceptions or not. When an unexpected exception occurs, the batch should still fail so that this exception is not ignored but its causes are analyzed.

Another way of handling exceptions in batches is retrying: Simply try to process the data once more and hope that everything works well this time. This approach often works for database problems, e.g. timeouts.

The section on exception handling explains skipping and retrying in more detail.

Note that exceptions are another reason why you should not execution a whole batch in one transaction. If anything goes wrong, you could either rollback the transaction and start the batch from scratch or you could manually revert all relevant changes. Both are not very good solutions.

Performance issues

In client processing mode, optimizing throughput (and response times) is an important topic as well, of course.

However, a performance that is still considered okay for client processing might be problematic for batches as these usually have to process large volumes of data and the time for their execution is usually quite limited (batches are often executed at night when no one is using the application).

An example: If processing the data of one person takes a second, this is usually still considered OK for client processing (even though performance could be better). However if a batch has to process the data of 100.000 persons in one night and is not executed with multiple threads, this takes roughly 28 hours, which is by far too much.

The section on performance contains some tips how to deal with performance problems.

3.5.2.2 Setup

Database

Spring Batch needs some meta data tables for monitoring batch executions and for restoring state for restarts. Detailed description about needed tables, sequences and indexes can be found in Spring Batch - Reference Documentation: Appendix B. Meta-Data Schema.

It is not recommended to add additional meta data tables, because this easily leads to inconsistencies with what is stored in those tables maintained by Spring Batch. You should rather try to extract all needed information out of the standard tables in case the standard API (especially JobRepository and JobExplorer, see below) does not fit your needs.

Failure information

BATCH_JOB_EXECUTION.EXIT_MESSAGE and BATCH_STEP_EXECUTION.EXIT_MESSAGE store a detailed description of how the job exited. In the case of failure, this might include as much of the stack trace as is possible. BATCH_STEP_EXECUTION_CONTEXT.SHORT_CONTEXT stores a stringified version of the step's ExecutionContext (see saving and restoring state, the rest is stored in a BLOB if needed). The default length of those columns in the sample schema scripts is 2500.

It is good to increase the length of those columns as far as the database allows it to make it easier to find out which exception failed a batch (not every exception causes a failure, see exception handling). Some JDBC drivers cast CLOBs to string automatically. If this is the case, you can use CLOBs instead.

General Configuration

OASP tries to be bean container independent and encourages application configuration with annotations. OASP batch layer bases on Spring Batch and bean configuration for this is done in class BeansBatchConfig.java. In order to control and monitor batch executions, there are some general beans needed and are provide by this class.This class contains beans like jobLauncher, jobRepository etc.

The jobRepository is used to update the meta data tables.

The transactionManager is needed for transaction management for internal operations of the job repository (i.e. for updating the meta data tables; to find the transaction manager used to do the main processing with, Spring Batch looks for a bean with named "transactionManager" automatically).

The database type needs to be set for correctly handling database specific things. Possible values are oracle, mysql, postgres, …

If the size of all three columns, which per default have a length limitation of 2500, has been increased as proposed here, the property maxVarCharLength should be adjusted accordingly in order to actually utilize the additional space.

The jobExplorer offers methods for reading from the meta data tables in addition to those methods provided by the jobRepository, e.g. getting the last executions of a batch.

The jobLauncher is used to actually start batches.

We use our own implementation here, which can be found in the module oasp4j-batch. It enables a special form of restarting a batch ("restart from scratch", see the section on restarts for further details).

The jobRegistry is basically a map, which contains all batch jobs. It is filled by the bean registerBatchJobs.

A JobParametersIncremeter (bean "incrementer") can be used to generate unique parameters, see restarts and parameters for further details.

The chunkLoggingListener is used to log exceptions together with the items (see chunk processing) where these exceptions occurred. It's implementation can be found in the module oasp4j-batch. Note that classes used for items have to have an appropriate toString() method in order for this listener to be useful.

For every job there is separate xml file defining batch steps (a job consists of steps) based on chunk processing and those based on tasklet processing from which concrete jobs and steps should inherit.

It makes sense to have a JobParametersIncrementer for every job and to use the ChunkLoggingListener for every step based on chunk processing. The abstract batch step for tasklet based processing is empty by now, but might be extended in the future.

3.5.2.3 Example-Batch

As already mentioned, every batch job consists of one or more batch steps, which internally either use chunk processing or tasklet based processing.

Our bill export batch job consists of the following to steps:

1. Read all (not processed) bills from the database, mark them as processed (additional attribute) and write them into a CSV file (to be further processed by other systems). This step is implemented using chunk processing (see chunk processing).
2. Delete all bill from the database which are marked as processed. This step is implemented in a tasklet (see tasklet based processing).

Note that you could also delete the bills directly. However, for being able to demonstrate tasklet based processing, we have created a separate step here.

Also note that in real systems you would usually create a backup of data as important as bills, which is not done here.

The beans-billexport.xml (located in src/main/resources/config/app/batch) has to look like this to implement the batch. Note that you might not fully understand this example by now, but you should after reading the whole chapter on batches.

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

  <batch:job id="billExportJob" incrementer="incrementer">
    <batch:step id="billExportStep">
      <batch:tasklet>
        <batch:chunk reader="billReader" processor="billProcessor" writer="billWriter" commit-
interval="2"/>
      </batch:tasklet>
    </batch:step>
  </batch:job>

  <bean id="billReader" class="org.springframework.batch.item.database.JdbcCursorItemReader">
    <property name="dataSource" ref="dataSource"/>
    <property name="rowMapper">
      <bean class="org.springframework.jdbc.core.SingleColumnRowMapper">
        <property name="requiredType" value="java.lang.Long"/>
      </bean>
    </property>
    <property name="sql">
      <bean class="io.oasp.module.jpa.dataaccess.base.NamedQueryFactoryBean">

  <property name="queryName" value="#{T(io.oasp.gastronomy.restaurant.general.common.api.constants.NamedQueries).GET_ALL_ID
>
      </bean>
    </property>
```

```
  </bean>

  <bean id="billProcessor" class="io.oasp.gastronomy.restaurant.salesmanagement.batch.impl.billexport.processor.BillProcess
>

  <bean id="billWriter" class="io.oasp.gastronomy.restaurant.salesmanagement.batch.impl.billexport.writer.BillCompositeWrit
    <property name="billLineAggregator">
      <bean class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
        <property name="fieldExtractor">
          <bean class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor">
            <property name="names" value="id, payed, total, tip"/>
          </bean>
        </property>
      </bean>
    </property>
    <property name="positionLineAggregator">
      <bean class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
        <property name="fieldExtractor">
          <bean class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor">
            <property name="names" value="id, orderId, cookId, offerId, offerName, state, price,
 comment"/>
          </bean>
        </property>
      </bean>
    </property>
    <property name="delegate">
      <bean class="org.springframework.batch.item.file.FlatFileItemWriter">
        <property name="lineAggregator">
          <bean class="org.springframework.batch.item.file.transform.PassThroughLineAggregator"/>
        </property>
      </bean>
    </property>
    <property name="resource" value="file:#{jobParameters['bills.file']}"/>
  </bean>

</beans>
```

The step elements always contains a tasklet element, even if chunk processing is used. The transaction-attributes element is especially used to set timeouts of transactions (in seconds). Note that there is usually more than one transaction per step (see below).

What follows is either a chunk element with ItemReader, ItemProcessor, ItemWriter and a commit interval (see chunk processing) or the tasklet element contains a reference to a tasklet.

In the example above the ItemReader billsReader always reads 1000 ids of unprocessed bills (via a DAO) and returns them one after another. The ItemProcessor billProcessor reads the corresponding bills from the database (see chunk processing why we do not read them directly in the ItemReader) and marks them as processed. The ItemWriter billWriter (see below on how this writer is configured) writes them to a CSV file. The path of this file is provided as batch parameter ("outputFile").

The tasklet billsDeleter deletes all processed bills (10.000 in one transaction).

3.5.2.4 Restarts

A batch execution is considered a restart, if it was run already (with the same parameters) and there was a (non skippable) failure or the batch has been stopped.

There are basically two ways how to do a restart:

- Undo all changes and restart from scratch.
- Restore the state of that batch at the time the error occurred and continue processing.

The first approach has two major disadvantages: One is that depending on what the batch does, reverting all of its changes can get quite complex. And you easily end up having implemented a batch that is restartable, but not if it fails in the wrong step.

The second disadvantage is that if a batch runs for several hours and then it fails it has to start all over again. And as the time for executing batches is usually quite limited, this can be problematic.

If reverting all changes is as easy as deleting all files in a given directory or something like that and the expected duration for an execution of the batch is rather short, you might consider the option of always starting at the beginning, otherwise you shouldn't.

Spring Batch supports implementing the second option. Per default, if a batch is restarted with the same parameters as a previous execution of this batch, then this new execution continues processing at the step where the last execution was stopped or failed. If the last execution was already complete (or has been abandoned), an exception is raised.

The step itself has to be implemented in a way so that it can restore its internal state, which is the main drawback of this second option.

However, there are 'standard implementations' that are capable of doing so and these can easily be adapted to your needs. They are introduced in the section on chunk processing.

For telling Spring Batch to always restart a batch at the very beginning even though there has been an execution of this batch with the same parameters already, set the restartable attribute of the Job element to false.

Per default, setting this attribute to false means that the batch is not restartable (i.e. it cannot be started with the same parameters once more). It would raise an error if there was attempt to do so, so that it cannot be restarted where it left off.

We use our own JobLauncher as described in the section on the general configuration to modify this behavior so that those batches are always restarted from the first step on by adding an extra parameter (instead of raising an exception), so that you do not have to take care of that yourself. So don't think of a batch marked with restartable="false" as a batch that is not restartable (as most people would probably assume just looking at the attribute) but as a batch that restarts always from the first step on.

Note that if a batch is restartable by restoring its internal state, it might not work correctly if the batch is started with different parameters after it failed, which usually comes down to the same thing as restating it from scratch. So the batch has to be restarted and complete successfully before executing the next regular 'run'. When scheduling batches, you should make that sure.

3.5.2.5 Chunk Processing

Chunk processing is item based processing. Items can be bills, persons or whatever needs to be processed. Those items are grouped into chunks of a fixed size and all items within such a chunk are processed in one transaction. There is not one transaction for every single (small) item because there would be too many commits which degrades performance.

All items of a chunk are read by an ItemReader (e.g. from a file or from database), processed by an ItemProcessor (e.g. modified or converted) and written out as a whole by an ItemWriter (e.g. to a file or to database).

The size of a chunk is also called commit interval. Careful when choosing a large chunk size: When a skip or retry occurs for a single item (see exception handling), the current transaction has to be rolled

back and all items of the chunk have to be reprocessed. This is especially a problem when skips and retries occur more often and results in long runtimes.

The most important advantages of chunk processing are:

- good trade-off between size and number of transactions (configurable via commit size)
- transaction timeouts that do not have to be adapted for larger amounts of data that needs to be processed (as there is always one transaction for a fixed number of items)
- an exception handling that is more fain-grained than aborting/restarting the whole batch (item based skipping and retrying, see exception handling)
- logging items where exceptions occurred (which makes failure analysis much more easy)

Note that you could actually achieve similar results using tasklets as described below. However, you would have to write many lines of additional code whereas you get these advantages out of the box using chunk processing (logging exceptions and items where these exceptions occurred is an extension, see general configuration).

Also note that items should not be too "big". For example, one might consider processing all bills within one month as one item. However, doing so you would not have those advantages any more. For instance, you would have larger transactions, as there are usually quite a lot of bills per month or payment method and if an exception occurs, you would not know which bill actually caused the exception. Additionally you would lose control of commit size, since one commit would comprsie many bills hard coded and you cannot choose smaller chuncks.

Nevertheless, there are sometimes situations where you cannot further "divide" items, e.g. when these are needed for one single call to an external system (e.g. for creating a PDF of all bills within a certain month, if PDFs are created by an external system). In this case you should do as much of the processing as possible on the basis of "small" items and then add an extra step to do what cannot be done based on these "small" items.

ItemReader

A reader has to implement the ItemReader interface, which has the following method:

```
public T read() throws Exception;
```

T is a type parameter of the ItemReader interface to be replaced with the type of items to be read.

The method returns all items (one at a time) that need to be processed or null if there are no more items.

If an exception occurs during read, Spring Batch cannot tell with item caused the exception (as it has not been read jet). That is why a reader should contain as little processing logic as possible, minimizing the potential for failures.

Caching

Per default, all items read by an ItemReader are cached by Spring Batch. This is useful because when a skippable exception occurs during processing of a chunk, all items (or at least those, that did not cause the exception) have to be reprocessed. These items are not read twice but taken from the cache then.

This is often necessary, because if a reader saves it current state in member variables (e.g. the current position within a list of items) or uses some sort of cursor, these will be updated already and the next calls of the read method would deliver the next items already and not those that have to be reprocessed.

However this also means that when the items read by an ItemReader are entities, these might be detached, because these might have been read in a different transaction. In some standard implementations Spring Batch even manually detaches entities in ItemReaders.

In case these entities are to be modified it is a good practice that the ItemReader only reads IDs and the ItemProcessor loads the entities for these IDs to avoid the problem.

Reading from Transactional Queues

In case the reader reads from a transactional queue (e.g. using JMS), you must not use caching, because then an item might get processed twice: Once from cache and once from queue to where it has been returned after the rollback. To achieve this, set reader-transactional-queue="true" in the chunk element in the step definition.

Moreover the equals and hashCode methods of the class used for items have to be appropriately implemented for Spring Batch to be able to identify items that were processed before unsuccessfully (causing a rollback and thereby returning them to the queue). Otherwise the batch might be caught in an infinite loop trying to process the same item over and over again (e.g. when the item is about to be skipped, see exception handling).

Reading from the Database

When selecting data from a database, there is usually some sort of cursor used. One challenge is to make this cursor not participate in the chunk's transaction, because it would be closed after the first chunk.

We will show how to use JDBC based cursors for ItemReader's in later releases of this documentation.

For JPA/JPQL based queries, cursors cannot be used, because JPA does not know of the concept of a cursor. Instead it supports pagination as introduced in the chapter on the data access layer, which can be used for this purpose as well. Note that pagination requires the result set to be sorted in an unambiguous order to work reliably. The order itself is irrelevant as long it does not change (you can e.g. sort the entities by their primary key).

ItemReader's using pagination should inherit from the AbstractPagingItemReader, which already provides most of the needed functionality. It manages the internal state, i.e. the current position, which can be correctly restored after a restart (when using an unambiguous order for the result set).

Classes inheriting from AbstractPagingItemReader must implement two methods.

The method doReadPage() performs the actual read of a page. The result is not returned (return type is void) but used to replace the content of the 'results' instance variable (type: List).

Due to our layering concept and the persistence layer being the only place where accesses to the database should take place, you should not directly execute a query in this method, but call a DAO, which itself executes the query (using pagination).

AbstractPagingItemReader provides methods for finding out the current position: use getPage() for the current page and getPageSize() for the (max.) page size. These values should be passed to the DAO as parameters. Note that the AbstractPagingItemReader starts counting pages from zero, whereas the PaginationTo used for pagination (retrieved by calling SearchCriteriaTo.getPagination()) starts counting from one, which is why you always have to increment the page number by one.

The second method is doJumpToPage(int), which usually only requires an empty implementation.

Furthermore, you need to set the property pageSize, which specifies how many items should be read at once. A page size that is as big as the commit interval usually results in the best performance.

The approach of using pagination for ItemReader's should not be used when items (usually entities) are added or removed or modified by the batch step itself or in parallel with the execution of the batch step so that the order changes, e.g. by other batches or due to operations started by clients (i.e. if the batch is executed in online mode). In this case there might be items processed twice or not processed at all. Be aware that due to hibernates Hi/Lo-Algorithm newer entities could get lower IDs than existing IDs and you probably will not process all entities if you rely on strict ID monotony!

A simple solution for such scenarios would be to introduce a new flag 'processed' for the entities read if that is an option (as it is also done in the example batch). The query should be rewritten then so that only unprocessed items are read (additionally limiting the result set size to the number of items to be processed in the current chunk, but not more).

Note that most of the standard implementations provided by Spring Batch do not fit to the layering approach in OASP applications, as these mostly require direct access to an EntityManager or a JDBC connection for example. You should think twice when using them an break the layering concept.

Reading from Files

For reading simply structured files, e.g. for those in which every line corresponds to an item to be processed by the batch, the FlatFileItemReader can be used. It requires two properties to be set: The first one the LineMapper (property lineMapper), which is used to convert a line (i.e. a String) to an item. It is a very simple interface which will not be discussed in more detail here. The second one is the resource, which is actually the file to be read. When set in the XML, it is sufficient to specify the path with a "file:" in front of it if it is a normal file from the file system.

In addition to that, the property linesToSkip (integer) can be set to skip headers for example. For reading more than one line before for creating an item a RecordSeparatorPolicy can be used, which will not be discussed in more detail here, too. Per default, all lines starting with a '#' will be considered to be a comment, which can be changed by changing the comment property (string array). The encoding property can be used to set the encoding. A FlatFileItemReader can restore its state after restarts.

For reading XML files, you can use the StaxEventItemReader (StAX is an alternative to DOM and SAX), which will not be discussed in further detail here.

In case the standard implementations introduced here do not fit your needs, you will need to implement your own ItemReader. If this ItemReader has some internal state (usually stored in member variables), which needs to be restored in case of restarts, see the section on saving and restoring state for information on how to do this.

ItemProcessor

A processor must implement the ItemProcessor interface, which has the following method:

```
public O process(I item) throws Exception;
```

As you can see, there are two type parameters involved: one for the type of items received from the ItemReader and one for the type of items passed to the ItemWriter. These can be the same.

If an item has been selected by the ItemReader, but there is no need to further process this item (i.e. it should not be passed to the ItemWriter), the ItemProcessor can return null instead of an item.

Strictly interpreting chunk processing, the ItemProcessor should not modify anything but should only give instructions to the ItemWriter how to do modifications. For entities however this is not really practical and as it requires no special logic in case of rollbacks/restarts (as all modifications are transactional), it is usually OK to modify them directly.

In contrast to this, performing accesses to files or calling external systems should only be done in ItemReader's/ItemWriter's and the code needed for properly handling failures (restarts for example) should be encapsulated there.

It is usually a good practice to make ItemProcessor's stateless, as the process method might be called more than once for one item (see the section on ItemReader's why). If your ItemProcessor really needs to have some internal state, see saving and restoring state on how to save and restore the state for restarts.

Do not forget to implement use cases instead of implementing everything directly in the ItemProcessor if the processing logic gets more complex.

ItemWriter

A writer has to implement the ItemWriter interface, which has the following method:

```
public void write(List<? extends T> items) Exception;
```

This method is called at the end of each chunk with a list of all (processed) items. It is not called once for every item, because it is often more efficient doing 'bulk writes', e.g. when writing to files.

Note that is method might also be called more than once for one item (see the section on ItemReader's why).

At the end of the write method, there should always be a flush.

When writing to files, this should be obvious, because when a chunks completes, it is expected that all changes are already there in case of restarts, which is not true if these changes were only buffered but have not been written out.

When modifying the database, the flush method on the EntityManager should be called, too (via a DAO), because otherwise there might be changes not written out jet and therefore constraints were not checked jet. This can be problematic, because Spring Batch considers all exceptions that occur during commit as critical, which is why these exceptions cannot be skipped. You should be careful using deferred constraints for the same reason.

Writing to Database or Transactional Queues

All changes made which are transactional can be conducted directly, there is no special logic needed for restarts, because these changes are applied if and only if the chunk succeeds.

Writing to Files

For writing simply structured files, the FlatFileItemWriter can be used. Similar to the FlatFileItemReader it requires the resource (i.e. the file) and a LineAggregator (property lineAggregator; instead of the lineMapper) to be set.

There are various properties that can be used of which we will only present the most important ones here. As with the FlatFileItemReader, the encoding property is used to set the encoding. A FlatFileHeaderCallback (property headerCallback) can be used to write a header.

The FlatFileItemWriter can restore its state correctly after restarts. In case the files contains too many line (written out in chunks that did not complete successfully), these lines are removed before continuing execution.

For writing XML files, you can use the StaxEventItemWriter, which will not be discussed in further detail here.

Just as with ItemReader's and ItemProcessor's: In case your ItemWriter has some internal state this state is not managed by a standard implementation, see saving and restoring state on how to make your implementation restartable (restart by restoring the internal state).

Saving and Restoring State

For saving and restoring (in case of restarts) state, e.g. saving and restoring values of member variables, the ItemStream interface should be implemented by the ItemReader/ItemProcessor/ItemWriter, which has the following methods:

```
public void open(ExecutionContext executionContext) throws ItemStreamException;
public void update(ExecutionContext executionContext) throws ItemStreamException;
public void close() throws ItemStreamException;
```

The open method is always called before the actual processing starts for the current step and can be used to restore state when restarting.

The ExecutionContext passed in as parameter is basically a map to be used to retrieve values set before the failure. The method containsKey(String) can be used to check if a value for a given key is set. If it is not set, this might be because the current batch execution is no restart or no value has been set before the failure.

There are several getter methods for actually retrieving a value for a given key: get(String) for objects (must be serializable), getInt(String), getLong(String), getDouble(String) and getString(String). These values will be the same as after the subsequent call to the update method after the last chunk that completed successfully. Note that if you update the ExecutionContext outside of the update method (e.g. in the read method of an ItemReader), it might contain values set in chunks that did not finish successfully after restarts, which is why you should not do that.

So the update method is the right place to update the current state. It is called after each chunk (and before and after each step).

For setting values, there are several put methods: put(String, Object), putInt(String, int), putLong(String, long), putDouble(String, double) and putString(String, String). You can choose keys (String) freely as long as these are unique within the current step.

Note that when a skip occurs, the update method is sometimes but not always called, so you should design your code in a way that it can deal with both situations.

The close method is usually not needed.

Do not misuse the ItemStream interface for purposes other than storing/restoring state. For instance, do not use the update method for flushing, because you will not have the chance to properly handle failure (e.g. skipping). For opening or closing a file handle, you should rather use a StepExecutionListener as introduced in the section on listeners. The state can also be restored in the beforeStep(ExecutionListener) method (instead of the open method).

Note that when a batch that always starts from scratch (i.e. the restartable attribute has been set to false for the batch job) is restarted, the ExecutionContext will not contain any state from the previous (failed) execution, so there is no use in storing the state in this case and usually no need to, of course, because the batch will start all over again.

3.5.2.6 Tasklet based Processing

Tasklets are the alternative to chunk processing. In the section on chunk processing we already mentioned the advantages of chunk processing as compared to tasklets. However, if only very few data needs to be processed (within one transaction) or if you need to do some sort of bulk operation (e.g. deleting all records from a database table), where the currently processed item does not matter and it is unlikely that a 'fain grained' exception handling will be needed, tasklets might still be considered an option. Note that for the latter use case you should still use more than one transaction, which is possible when using tasklets, too.

Tasklets have to implement the interface with the same name, which has the following method:

```
public RepeatStatus execute(StepContribution contribution, ChunkContext chunkContext) throws Exception;
```

This method might be called several times. Every call is executed inside a new transaction automatically. If processing is not finished yet and the execute method should be called once more, just use RepeatStatus.CONTINUABLE as return value and RepeatStatus.FINISHED otherwise.

The StepContribution parameter can be used to set how many items have been processed manually (which is done automatically using chunk processing), there is, however, usually no need to do so.

The ChunkContext is similar to the ExecutionContext, but is only used within one chunk. If there is a retry in chunk processing, the same context should be used (with the same state that this context had when the exception occurred).

Note that tasklets serve as the basis for chunk processing internally. For chunk processing there is a Spring Batch internal tasklet, which has an execute method that is called for every chunk and itself calls ItemReader, ItemProcessor and ItemWriter.

That is the reason why a StepContribution and a ChunkContext are passed to tasklets as parameters, even though they are more useful in chunk processing. Moreover this is also the reason why you have to use the tasklet element in the XML even though you want to specify a step that uses chunk processing (see the example batch).

3.5.2.7 Exception Handling

As already mentioned, in chunk processing you can configure a step so that items are skipped or retried when certain exceptions occur.

If reties are exhausted (per default, there is no retry) and the exception that occurred cannot be skipped (per default, no exception can be skipped), the batch will fail (i.e. stop executing).

In tasklet based processing this cannot be done, the only chance is to implement the needed logic yourself.

Skipping

Before skipping items you should think about what to do if a skip occurs. If a skip occurs, the exception will be logged in the server log. However if no one evaluates those logs on a regular basis and informs those who are affected further actions need to take place when implementing the batch.

Implement the SkipListener interface to be informed when a skip occurs. For example, you could store a notification or send a message to someone. For skips that occurred in ItemReader's there is no information available about the item that was skipped (as it has not been read jet) which is why there should be as little processing logic as possible in an ItemReader. It might also be a reason why you might want to forbid to skip exceptions that might occur in readers.

Do not try to catch skipped exceptions and write something into the database in a new transaction (e.g. a notification) instead of using a SkipListener, because a skipped item might be processed more than once before actually being skipped (for example, if a skippable exception is thrown during a call of an ItemWriter, Spring Batch does not know which item of the current chunk actually caused the exception and therefore has to retry each item separately in order to know which item actually caused the exception).

Skippable exception classes can be specified as shown below:

```
<batch:chunk ... skip-limit="10">
   <batch:skippable-exception-classes>
      <batch:include class="..."/>
      <batch:include class="..."/>
      ...
   </batch:skippable-exception-classes>
</batch:chunk>
```

The attribute skip-limit, which has to be set in case there is any skippable exception class configured, is used to set how many items should be skipped at most. It is useful to avoid situations where very many items are skipped but the batch still completes successfully and no one notices this situation.

Skippable exception classes are specified by their fully qualified name (e.g. java.lang.Exception), each of such class set in its own include element as shown above. Subclasses of such classes are also skipped.

To programmatically decide whether to skip an exception or not, you can set a skip policy as shown below:

```
<batch:chunk ... skip-policy="mySkipPolicy">
```

The skip policy (here mySkipPolicy) has to be a bean that implements the interface SkipPolicy with the following method:

```
public boolean shouldSkip(java.lang.Throwable t,
                  int skipCount)
            throws SkipLimitExceededException
```

To skip the exception and continue processing, just return true and otherwise false.

The parameter skipCount can be used for a skip limit. A SkipLimitExceededException should be thrown if there should be thrown if there should be no more skips. Note that this method is sometimes called with a skipCount less than zero to test if an exception is skippable in general.

When a SkipPolicy is set, the attribute skip-limit and element skippable-exception-classes are ignored.

You could of course skip every exception (using java.lang.Exception as skippable exception class). This is, however, not a good practice as it might easily result in an error in the code that is ignored as the batch still completes successfully and everything seems to be fine. Instead, you should think about what kind of exceptions might actually occur, what to do if they occur and if it is OK to skip them. If an unexpected exception occurs, it is usually better to fail the batch execution and analyze the cause of the exception before restarting the batch.

Exceptions that can occur in ItemWriter's that write something to file should not be skipped unless the ItemWriter can properly deal with that. Otherwise there might be data written out even though the according item is skipped, because operations in the file systems are not transactional.

Another situation where skips can be problematic is when calls to external interfaces are being made and these calls change something "on the other side", as these calls are usually not transactional. So be careful using skips here, too.

Retrying

For some types of exceptions, processing should be retried independently of weather the exception can be skipped or would otherwise fail the batch execution.

For example, if there was a database timeout, this might be because there were too many requests at the time the chunk was processed. And it is not unlikely that retrying to successfully complete the chunk would succeed.

There are, of course, also exceptions where retrying does not make much sense. E.g. exceptions caused by the business logic should be deterministic and therefore retrying does not make much sense in this case.

Nevertheless, retrying every exception results in longer runtime but should in general be considered OK if you do not know which exceptions might occur or do not have the time to think about it.

Retryable exception classes can be set similarly to setting skippable exception classes:

```
<batch:chunk ... retry-limit="3">
   <batch:retryable-exception-classes>
      <batch:include class="..."/>
      <batch:include class="..."/>
      ...
   </batch:retryable-exception-classes>
</batch:chunk>
```

The retry-limit attribute specifies how many times one individual item can be retried, as long as the exception thrown is "retryable".

As with skippable exception classes, retryable exception classes are set in include elements and their subclasses are retried, too.

To programmatically decide, whether to retry an exception or not, you can use a RetryPolicy, which is not covered in more detail here.

Note that even if no retry is configured, an item might nevertheless be processed more than once. This is because if a skippable exception occurs in a chunk, all items of the chunk that did not cause the exception have to reprocessed, which is done in a separate transaction for every item, as the transaction in which these items were processed in the first place was rolled back. And even if the exception is not skippable, there is no guarantee that Spring Batch will not attempt to reprocess each item separately.

3.5.2.8 Listeners

Spring Batch provides various listeners for various events to be notified about.

For every listener there is an interface which can either be implemented by an ItemReader, ItemProcessor, ItemWriter or Tasklet or by a separate listener class, which can be registered for a step like this:

```
    <batch:tasklet>
        <batch:chunk .../>
        <batch:listeners merge="true">
            <batch:listener ref="listener1"/>
            <batch:listener ref="listener2"/>
            ....
        </batch:listeners>
    </batch:tasklet>
    <beans:bean id="listener1" class=".."/>
    <beans:bean id="listener2" class=".."/>
    ...
```

Do not forget to set the merge attribute to true as shown above, because otherwise the general listeners configured in general configuration will not be active any more.

The most commonly use listener is probably the StepExecutionListener, which has methods that are called before and after the execution of the step. It can be utilized e.g. for opening and closing files.

The following example shows how to use the listener:

```
public class MyListener implements StepExecutionListener {

 public void beforeStep(StepExecution stepExecution) {
  // take actions before processing of the step starts
 }

 public ExitStatus afterStep(StepExecution stepExecution) {
  try {
   // take actions after processing is finished
  } catch (Exception e) {
   stepExecution.addFailureException(e);
   stepExecution.setStatus(BatchStatus.FAILED);
   return ExitStatus.FAILED.addExitDescription(e);
  }
  return null;
 }

}
```

In the afterStep(StepExecution) method, you can check the outcome of the batch execution (completed, failed, stopped etc.) checking the ExitStatus, which can be accessed via StepExecution#getExitStatus(). You can even modify the ExitStatus by returning a new ExitStatus, which is something we will not discuss in further detail here. If you do not want to modify the ExitStatus, just return null.

Throwing an exception in this method has no effect. If you want to fail the whole batch in case an exception occurs, you have to do an exception handling as shown above. This does not apply to the beforeStep method.

For other types of listeners (among others the SkipListener mentioned already) see Spring Batch Reference Documentation - 5. Configuring a Step - Intercepting Step Execution.

Note that exception handling for listeners is often a problem, because exceptions are mostly ignored, which is not always documented very well. If an important part of a batch is implemented in listener methods, you should always test what happens when exceptions occur. Or you might think about not implementing important things in listeners …

If you want an exception to fail the whole batch, you can always wrap it in a FatalStepExecutionException, which will stop the execution.

3.5.2.9 Parameters

The section on starting and stopping batches already showed how to start a batch with parameters.

One way to get access to the values set is using the StepExecutionListener introduced in the section on listeners like this:

```
public void beforeStep(StepExecution stepExecution) {

 String parameterValue = stepExecution.getJobExecution().getJobParameters().
  getString("parameterKey");
}
```

There are getter methods for strings, doubles, longs and dates. Note that when set via the CommandLineJobRunner, all parameters will be of type string unless the type is specified in brackets after the parameter key, e.g. processUntil(date)=2015/12/31. The parameter key here is "processUntil".

Another way is to inject values. In order for this to work, the bean has to have step scope, which means there is a new object created for every execution of a batch step. It works like this:

```
<bean id="myProcessor" class="...MyItemProcessor" scope="step">
 <property name="parameter" value="#{jobParameters['parameterKey']}" />
<bean>
```

There has to be an appropriate setter method for the parameter of course.

As already mentioned in the section on restarts, a batch that successfully completed with a certain set of parameters cannot be started once more with the same parameters as this would be considered a restart, which is not necessary, because the batch was already finished.

So using no parameters for a batch would mean that it can be started until it completes successfully once, which usually does not make much sense.

As batches are usually not executed more than once a day, we purpose introducing a general "date" parameter (without time) for all batch executions.

It is advisable to add the date parameter automatically in the JobLauncher if it has not been set manually, which can be done as shown below:

```
private static final String DATE_PARAMETER = "date";

...

if (jobParameters.getDate("DATE_PARAMETER") == null) {

 Date dateWithoutTime = new Date();
 Calendar cal = Calendar.getInstance();
 cal.setTime(dateWithoutTime);
 cal.set(Calendar.HOUR_OF_DAY, 0);
 cal.set(Calendar.MINUTE, 0);
 cal.set(Calendar.SECOND, 0);
 cal.set(Calendar.MILLISECOND, 0);
 dateWithoutTime = cal.getTime();

 jobParameters = new JobParametersBuilder(jobParameters).addDate(
  DATE_PARAMETER, dateWithoutTime).toJobParameters();

 ... // using the jobParametersIncrementer as shown above
}
```

Keep in mind that you might need to set the date parameter explicitly for restarts. Also note that automatically setting the date parameter can be problematic if a batch is sometimes started before and sometimes after midnight, which might result in a batch not being executed (as it has already been executed with the same parameters), so at least for productive systems you should always set it explicitly.

The date parameters can also be useful for controlling the business logic, e.g. a batch can process all data that was created until the current date (as set in the date parameter), thereby giving a chance to control how much is actually processed.

If your batch has to run more than once a day you could easily adapte the concept for timestamps. If you are using an external batch scheduler, they often provide a counter for the execution and you might automatically pass this instead of the date parameter.

3.5.2.10 Performance Tuning

Most important for performance are of course the algorithms that you write and how fast (and scalable) these are, which is the same as for client processing. Apart from that, the performance of batches is usually closely related to the performance of the database system.

If you are retrieving information from the database, you can have one complex query executed in the ItemReader (via a DAO) retrieving all the information needed for the current set of items, or you can execute further queries in the ItemProcessor (or ItemWriter) on a per item basis to retrieve further information.

The first approach is usually by far more performant, because there is an overhead for every query being executed and this approach results in less queries being executed. Note that there is a tradeoff between performance and maintainability here. If you put everything into the query executed by an ItemReader, this query can get quite complex.

Using cursors instead of pagination as described in the section on ItemReaders can result in a better performance for the same reason: When using a cursor, the query is only executed once, when using pagination, the query is usually executed once per chunk. You could of course manually cache items, however this easily leads to a high memory consumption.

Further possibilities for optimizations are query (plan) optimization and adding missing database indexes.

3.5.2.11 Testing

The Section Testing covers how to unit and integration test in detail. Therefore we focus here on testing batches.

In order for the unit test to run a batch job the unit test class must extends the AbstractSpringBatchIntegrationTest class. Two annotations are used to load the job's ApplicationContext:

@RunWith(SpringJUnit4ClassRunner.class): Indicates that the class should use Spring's JUnit facilities

@ContextConfiguration(locations = {...}): Indicates which XML files contain the ApplicationContext.

```
@RunWith(SpringJUnit4ClassRunner.class)
@DirtiesContext(classMode = ClassMode.AFTER_CLASS)
@ActiveProfiles("db-plain")
public abstract class AbstractSpringBatchIntegrationTest {..}
```

```
@ContextConfiguration(locations = { ApplicationConfigurationConstants.BEANS_BATCH })
public class ProductImportJobTest extends AbstractSpringBatchIntegrationTest {..}
```

Testing Batch Jobs

For testing the complete run of a batch job from beginning to end involves following steps:

Set up a test condition,

execute the job,

Verify the end result.

The test method below begins by setting up the database with test data. The test then launches the Job using the launchJob() method. The launchJob() method is provided by the JobLauncherTestUtils class.

Also provided by the utils class is launchJob(JobParameters), which allows the test to give particular parameters. The launchJob() method returns the JobExecution object which is useful for asserting particular information about the Job run. In the case below, the test verifies that the Job ended with ExitStatus "COMPLETED".

```
@ContextConfiguration(locations = { ApplicationConfigurationConstants.BEANS_BATCH })
public class ProductImportJobTest extends AbstractSpringBatchIntegrationTest {

  @Inject
  private Job billExportJob;

  @Test
  public void shouldExportBills() throws Exception {
    JobExecution jobExecution = getJobLauncherTestUtils(this.billExportJob).launchJob();
    assertThat(jobExecution.getExitStatus()).isEqualTo(ExitStatus.COMPLETED);
  }
}
```

Note that when using the launchJob() method, the batch execution will never be considered as a restart (i.e. it will always start from scratch). This is achieved by adding a unique (random) parameter.

This is not true for the method launchJob(JobParameters) however, which will result in an exception if the test is executed twice or a batch is executed in two different tests with the same parameters.

We will add methods for appropriately handling this situation in future releases of OASP. Until then you can help yourself by using the method getUniqueJobParameters() and then add all required parameters to those parameters returned by the method (as shown in the section on parameters).

Also note that even if skips occurred, the ExitStatus is still COMPLETED. That is one reason why you should always checks weather the batch did what it was supposed to do or not.

Testing Individual Steps

For complex batch jobs individual steps can be tested. For example to test a createCsvFile, run just that particular Step. This approach allows for more targeted tests by allowing the test to set up data for just that step and to validate its results directly.

```
JobExecution jobExecution = getJobLauncherTestUtils(this.billExportJob).launchStep(“createCsvFile”);
```

Validating Output Files

When a batch job writes to the database, it is easy to query the database to verify the output. To facilitate the verification of output files Spring Batch provides the class AssertFile. The method assertFileEquals takes two File objects and asserts, line by line, that the two files have the same content. Therefore, it is possible to create a file with the expected output and to compare it to the actual result:

```
private static final String EXPECTED_FILE = "classpath:expected.csv";
private static final String OUTPUT_FILE = " file:./temp/output.csv";
AssertFile.assertFileEquals(new FileSystemResource(EXPECTED_FILE), new FileSystemResource(OUTPUT_FILE));
```

Testing Restarts

Simulating an exception at an arbitrary method in the code can be done relatively easy using AspectJ. Afterwards you should restart the batch and check if the outcome is still correct.

Note that when using the launchJob() method, the batch is always started from the beginning (as already mentioned). Use the launchJob(JobParameters) instead with the same parameters for the initial (failing) execution and for the restart.

Test your code thoroughly. There should be at least one restart test for every step of the batch job.

4. Guides

4.1 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 Java EE 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.

4.1.1 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 Java EE 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. It also allows greater flexibility as you can do some processing when injecting the value. 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.

4.1.2 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.

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;
  }
```

4.1.3 Bean configuration

Wiring and Bean configuration can be found in configuration guide.

4.2 Configuration

An application needs to be configurable in order to allow internal setup (like CDI) but also to allow externalized configuration of a deployed package (e.g. integration into runtime environment). Using Spring Boot (must read: Spring Boot reference) we rely on a comprehensive configuration approach following a "convention over configuration" pattern. This guide adds on to this by detailed instructions and best-practices how to deal with configurations.

In general we distinguish the following kinds of configuration that are explained in the following sections:

- Internal Application configuration maintained by developers
- Externalized Environment configuration maintained by operators
- Externalized Business configuration maintained by business administrators

4.2.1 Internal Application Configuration

The application configuration contains all internal settings and wirings of the application (bean wiring, database mappings, etc.) and is maintained by the application developers at development time. There usually is a main configuration registered with main Spring Boot App, but differing configurations to support automated test of the application can be defined using profiles (not detailed in this guide).

Appetizer: The web.xml as the place for all web related configuration is not at all used anymore to configure the web app. It is empty.

4.2.1.1 Standard beans configuration

For basic bean configuration we rely on spring boot using mainly configuration classes and occasionally xml-configuration files. Some key principle to understand Spring Boot auto-configuration features:

- Spring Boot auto-configuration attempts to automatically configure your Spring application based on the jar dependencies and annotated components found in your source code.
- Auto-configuration is noninvasive, at any point you can start to define your own configuration to replace specific parts of the auto-configuration by redefining your identically named bean.

Beans are configured via annotations at the java-class (@Component, @Bean, @Named, etc.). These beans will be known when wiring the application at runtime. The required component scan is already auto-enabled within the main SpringBootApp.

For beans that need separate configuration for any reason, additional Configuration Classes (using annotation @Configuration) can be used and will be automatically evaluated during application startup.

Configuration classes all reside in the folder:

src/main/general/configuration/

4.2.1.2 XML-based beans configuration

It is still possible and allowed to provide (bean-) configurations using xml, though not recommended. These configuration files are no more bundled via a main xml config file but loaded individually from their respective owners, e.g. for unit-tests:

```
@SpringApplicationConfiguration(classes = { SpringBootApp.class }, locations = { "classpath:/config/app/
batch/beans-productimport.xml" })
public class ProductImportJobTest extends AbstractSpringBatchIntegrationTest {
...
```

Configuration XML-files reside in an adequately named subfolder of:

src/main/resources/app

4.2.1.3 Batch configuration

In the directory src/main/resources/config/app/batch we place the configuration for the batch jobs. Each file within this directory represents one batch job.

4.2.1.4 WebSocket configuration

A websocket endpoint is configured within the business package as a Spring configuration class. The annotation @EnableWebSocketMessageBroker makes Spring Boot registering this endpoint.

```
package io.oasp.gastronomy.restaurant.salesmanagement.websocket.config;
...
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig extends AbstractWebSocketMessageBrokerConfigurer {
...
```

4.2.2 Externalized Configuration

Externalized configuration is a configuration that is provided separately to a deployment package and can be maintained undisturbed by redeployments.

4.2.2.1 Environment Configuration

The environment configuration contains configuration parameters (typically 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.

The environment configuration is maintained in `application.properties` files, defining various properties. These properties are explained in the corresponding configuration sections of the guides for each topic:

- persistence configuration
- service configuration
- logging guide

There are two properties files already contained within the example server:

- src/main/resources/application.properties providing a default configuration - bundled and deployed with the application package. It further acts as a template to derive a tailored minimal environment-specific configuration.
- src/main/resources/config/application.properties providing additional properties only required at development time (for all local deployment scenarios). This property file is excluded from all packaging.

Where to actually put a tailored application.properties file after deployment depends on the deployment strategy:

- standalone runnable Spring Boot App using embedded tomcat: put a tailored copy of application.properties into installpath/config/
- dedicated tomcat (one tomcat per app): put a tailored copy of application.properties into tomcat/lib/
- tomcat serving a number of apps (requires expanding the wars): put a tailored copy of application.properties into tomcat/webapps/<app>/WEB-INF/classes/config

In this application.properties you only define the minimum properties that are environment specific and inherit everything else from the bundled src/main/resources/application.properties . In any case, make very sure that the classloader will find the file.

Make sure your properties are thoroughly documented by providing a comment to each property. This inline documentation is most valuable for your operating department.

4.2.2.2 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 administrative 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

4.3 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.

4.3.1 Usage

4.3.1.1 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>
```

4.3.1.2 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).

The provided logback.xml is configured to use variables defined on the config/application.properties file. On our example, the log files path point to ../logs/ in order to log to tomcat log directory when starting tomcat on the bin folder. Change it according to your custom needs.

config/application.properties.

```
log.dir=../logs/
```

4.3.1.3 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);
  ...
}
```

Please note that in this case we are not using injection pattern but use the convenient static alternative. This is already a common solution and also has performance benefits.

4.3.1.4 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 or shutdown/restart required)	Operator has to react immediately	all
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.

4.3.2 Operations

4.3.2.1 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.

4.3.2.2 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...]
```

4.3.3 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.

4.3.4 Correlating separate requests

In order to correlate separate HTTP requests to services belonging to the same user / session, we provide a servlet filter called "DiagnosticContextFilter". This filter first searches for a configurable HTTP

header containing a correlation id. If none was found, it will generate a new correlation id. By default the HTTP header used is called "CorrelationId".

4.4 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.

4.4.1 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>
```

4.4.1.1 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.

4.4.1.2 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.

4.4.1.3 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.

4.4.2 Authorization

Definition:

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

4.4.2.1 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.

Term	Meaning and comment
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.
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.

4.4.2.2 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.

4.4.2.3 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.

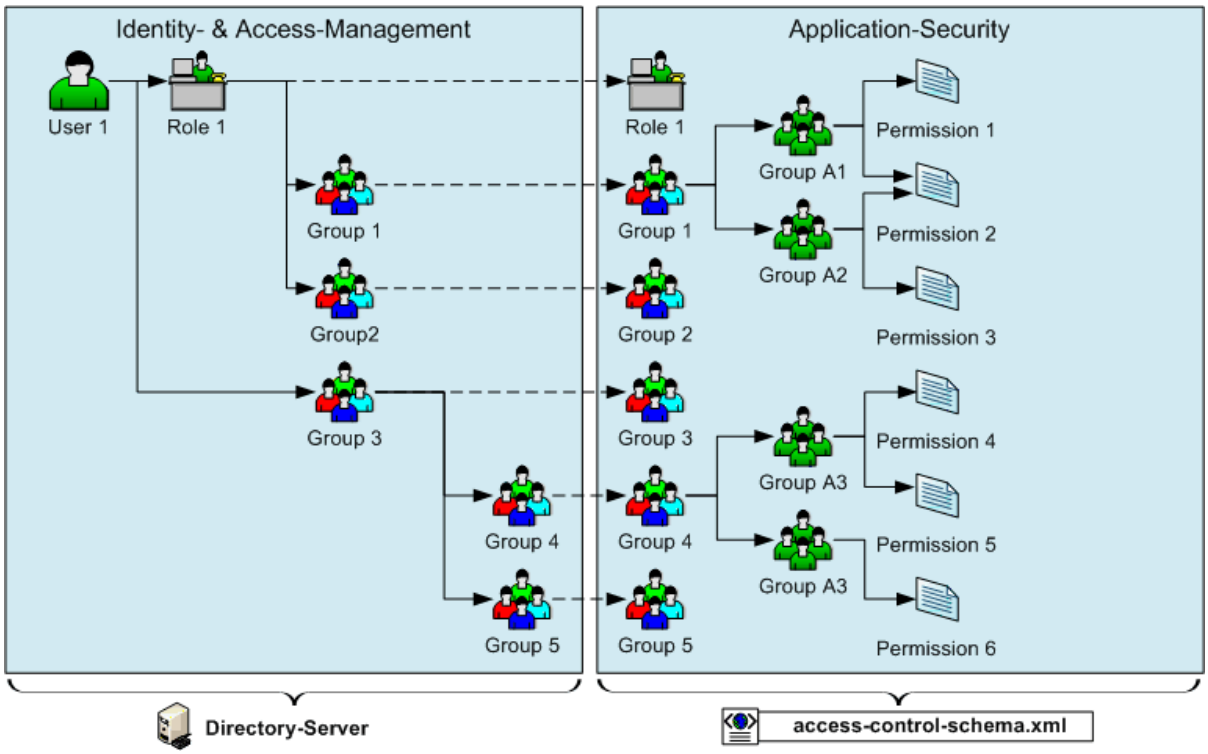


Figure 4.1. OASP4J Security Model

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".

Term	oasp4j-security implementation	Comment
Role	`AccessControlGroup` with `type="role"`	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.

4.4.3 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
A4 Insecure Direct Object References	Using direct object references (IDs) only with appropriate authorization	logic-layer
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	service-layer security

Thread	Protection	Link to details
	HTTP header and verified on the server	
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

4.5 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 Auditing

For database auditing we use hibernate envers. If you want to use auditing ensure you have the following dependency in your pom.xml:

```
<dependency>
  <groupId>io.oasp.java.modules</groupId>
  <artifactId>oasp4j-jpa-envers</artifactId>
</dependency>
```

Make sure that entity manager (configured in beans-jpa.xml) also scans the package from the oasp4j-jpa[-envers] module in order to work properly.

```
...
<property name="packagesToScan">
  <list>
    <value>io.oasp.module.jpa.dataaccess.api</value>
    ...
  </list>
```

Now let your DAO implementation extend from AbstractRevisionedDao instead of AbstractDao and your DAO interface extend from [Application]RevisionedDao instead of [Application]Dao.

The DAO now has a method getRevisionHistory(entity) available to get a list of revisions for a given entity and a method load(id, revision) to load a specific revision of an entity with the given ID.

To enable auditing for a entity simply place the @Audited annotation to your entity and all entity classes it extends from.

```
@Entity(name = "Drink")
@Audited
public class DrinkEntity extends ProductEntity implements Drink {
...
```

When auditing is enabled for an entity an additional database table is used to store all changes to the entity table and a corresponding revision number. This table is called <ENTITY_NAME>_AUD per default. Another table called REVINFO is used to store all revisions. Make sure that these tables are available. They can be generated by hibernate with the following property (only for development environments).

```
  database.hibernate.hbm2ddl.auto=create
```

Another possibility is to put them in your database migration scripts like so.

```
CREATE CACHED TABLE PUBLIC.REVINFO(
  id BIGINT NOT NULL generated by default as identity (start with 1),
  timestamp BIGINT NOT NULL,
  user VARCHAR(255)
);
...
CREATE CACHED TABLE PUBLIC.<TABLE_NAME>_AUD(
    <ALL_TABLE_ATTRIBUTES>,
    revtype TINYINT,
    rev BIGINT NOT NULL
);
```

4.7 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.

4.7.1 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).

4.7.2 AOP Usage

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

- Transaction-Handling
- Authorization
- Validation
- 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).

4.8 Exception Handling

4.8.1 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.

4.8.2 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);
}
```

4.8.3 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.

4.9 Internationalization

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

In OASP we have developed a solution to manage text internationalization. OASP solution comes into two aspects:

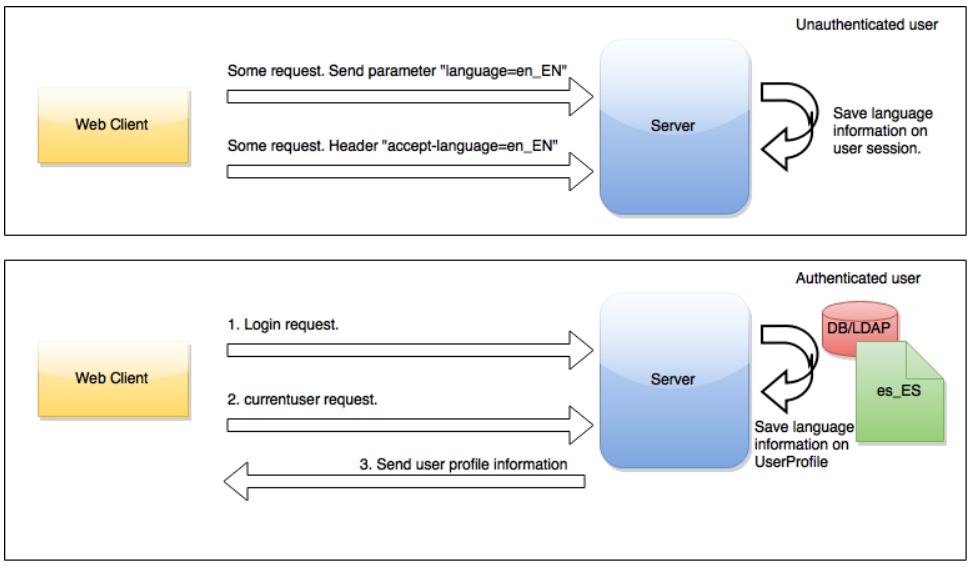
- Bind locale information to the user.
- Get the messages in the current user locale.

4.9.1 Binding locale information to the user

We have defined two different points to bind locale information to user, depending on user is authenticated or not.

- User not authenticated: Oasp intercepts unsecured request and extract locale from it. At first, we try to extract a `language` parameter from the request and if it is not possible, we extract locale from Àccept-language` header.
- User authenticated. During login process, applications developers are responsible to fill `language` parameter in the UserProfile class. This `language` parameter could be obtain from DB, LDAP, request, etc. In OASP sample we get the locale information from database.

This image shows the entire process:



4.9.2 Getting internationalizated messages

OASP has a bean that manage i18n message resolution, the `ApplicationLocaleResolver.` This bean is responsible to get the current user and extract locale information from it and read the correct properties file to get the message.

The i18n properties file must be called `ApplicationMessages_la_CO.properties` where la=language and CO=country. This is an example of a i18n properties file for English language to translate OASP sample user roles:

ApplicationMessages_en_EN.properties

```
waiter=Waiter
chief=Chief
cook=Cook
barkeeper=Barkeeper
```

You should define an ApplicationMessages_la_CO.properties file for every language that your application needs.

`ApplicationLocaleResolver` bean is injected in `AbstractComponentFacade` class so you have available this bean in logic layer so you only need to put this code to get an internationalizated message:

```
String msg = getApplicationLocaleResolver().getMessage("mymessage");
```

4.10 XML

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

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

4.10.1 JAXB

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

4.10.1.1 JAXB and Inheritance

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

4.10.1.2 JAXB Custom Mapping

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

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

4.11 JSON

JSON (JavaScript Object Notation) is a popular format to represent and exchange data especially for modern web-clients. For mapping Java objects to JSON and vice-versa there is no official standard API. We use the established and powerful open-source solution Jackson. Due to problems with the wiki of fasterxml you should try this alternative link: Jackson/AltLink.

4.11.1 JSON and Inheritance

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

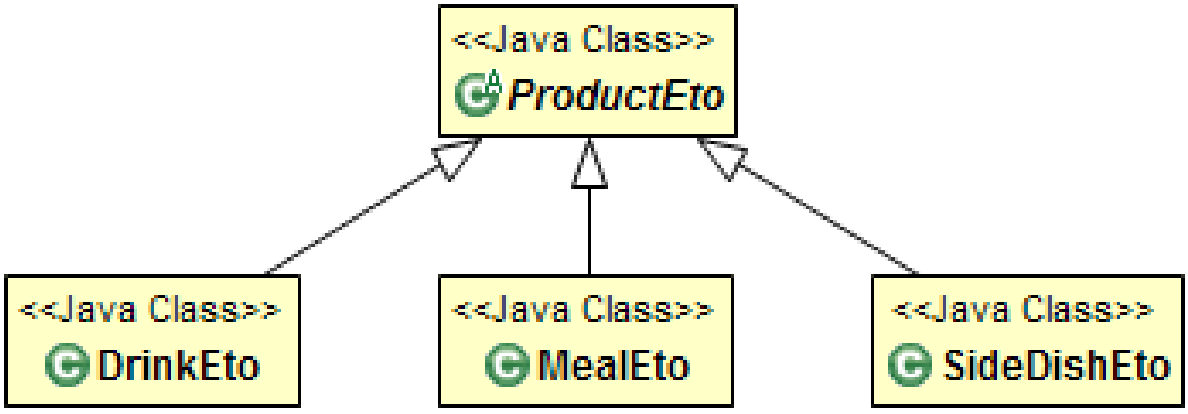


Figure 4.2. Transfer-Objects using Inheritance

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.

4.11.2 JSON Custom Mapping

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.12 Testing

4.12.1 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 integration levels.
- 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.
- **Use assert frameworks** like AssertJ to write good readable and maintainable tests that also provide out-of-the-box good failure reports in case a test fails.
- Plan your tests and test data 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 *automated system 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.
- **Do not use inheritance for cross-cutting testing functionality**: Sometimes cross-cutting functionality like opening/closing a database connection or code to fill a database with test data is put in a common parent class like AbstractTestCase that all test classes need to inherit from. Starting with some functions this classes tend to grow up to the point where they become real maintenance nightmares. Good places to put this needed kind of code can be realized using JUnit @Rule mechanism. In general favor delegation over inheritance. There are reasons why frameworks like JEE or JUnit do not use inheritance for technical features - and for the same reasons also project test frameworks should not do it.

4.12.2 Test Automation Technology Stack

For test automation we use JUnit. However, we are strictly doing all assertions with AssertJ. For mocking we use mockito. In order to mock remote connections we use wiremock. For testing entire components or integrations we recommend to use spring-test.

4.12.3 Test Doubles

Due to the non-consistent use and understanding of mocks/stubs/fakes/dummies for any kind of interface for testing purposes, we shortly want to give a common understanding about the different types of test doubles. Therefore we mainly stick on Gerard Meszaros's definitions, who also introduced the term test doubles as generic term for mocks/stubs/fakes/dummies/spys. Another interesting discussion about stubs VS mocks has been published by Martin Fowler, which focuses more on the differences between stubs and mocks. A short summary (by Martin Fowler):

- **Dummy** objects are passed around but never actually used. Usually they are just used to fill parameter lists.
- **Fake** objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in memory database is a good example).
- **Stubs** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test. Stubs may also record information about calls, such as an email gateway stub that remembers the messages it 'sent', or maybe only how many messages it 'sent'.
- **Mocks** are objects pre-programmed with expectations, which form a specification of the calls they are expected to receive.

What both authors do not cover is the applicability of the different concepts. We try to give some examples, which should make it somehow clearer:

4.12.3.1 Stubs

Best Practices for applications:

- A good way to replace small to medium large boundary systems, whose impact (e.g. latency) should be ignored during performing load and performance tests of the application under development.
- As stub implementation will rely on state-based verification, there is the threat, that test developers will partially reimplement the state transitions based on the replaced code. This will immediately lead to a black maintenance whole, so better use mocks to assure the certain behavior on interface level.
- Do NOT use stubs as basis of a large amount of test cases as due to state-based verification of stubs, test developers will enrich the stub implementation to become a large monster with its own hunger after maintenance efforts.

4.12.3.2 Mocks

Best Practices for applications:

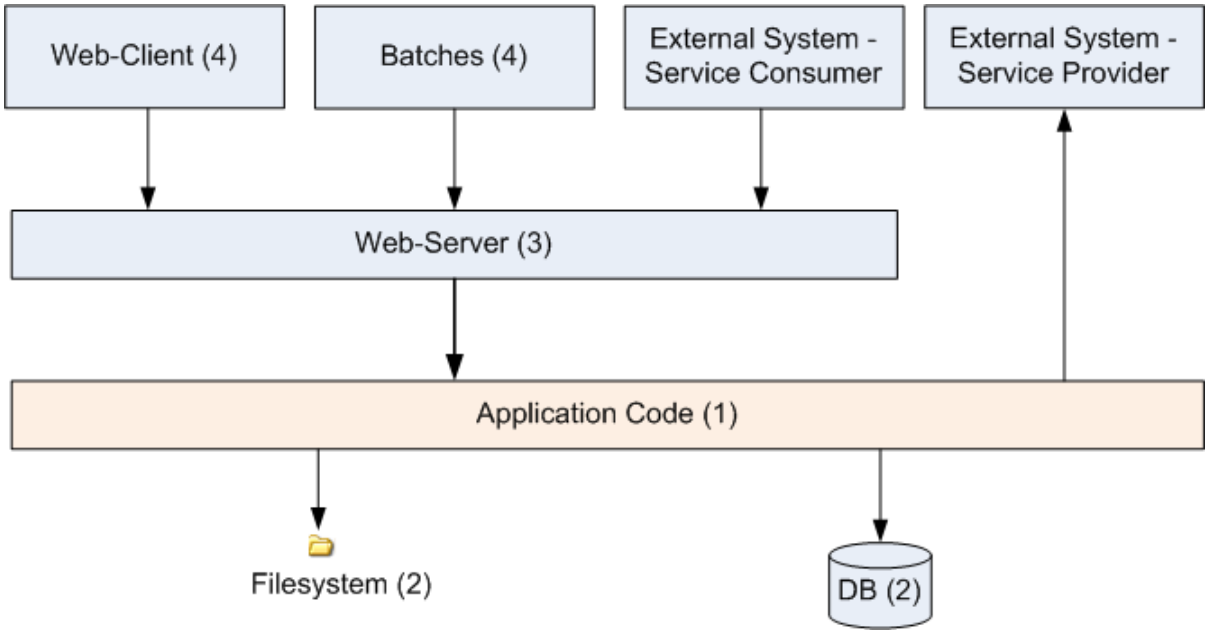
- Replace not-needed dependencies of your system-under-test (SUT) to minimize the application context to start of your component framework.
- Replace dependencies of your SUT to impact the control flow under test without establishing all the context parameters needed to match the control flow.
- Remember: Not everything has to be mocked! Especially on lower levels of tests like isolated module tests you can be betrayed into a mocking delusion, where you end up in a hundred lines of code

mocking the whole context and five lines executing the test and verifying the mocks behavior. Always keep in mind the benefit-cost ratio, when implementing tests using mocks.

4.12.4 Integration Levels

There are many discussions about the right level of integration for test automation. Sometimes it is better to focus on small, isolated modules of the system - whatever a "module" may be. In other cases it makes more sense to test integrated groups of modules. Because there is no universal answer to this question, OASP only defines a common terminology for what could be tested. Each project must make its own decision where to put the focus of test automation. There is no worldwide accepted terminology for the integration levels of testing. In general we we consider ISTQB. However, with a technical focus on test automation we want to get more precise.

The following picture shows a simplified view of an application based on the OASP reference architecture. We define four integration levels that are explained in detail below. The boxes in the picture contain parenthesized numbers. These numbers depict the lowest integration level, a box belongs to. Higher integration levels also contain all boxes of lower integration levels. When writing tests for a given integration level, related boxes with a lower integration level must be replaced by test doubles or drivers.



The main difference between the integration levels is the amount of infrastructure needed to test them. The more infrastructure you need, the more bugs you will find, but the more instable and the slower your tests will be. So each project has to make a trade-off between pros and contras of including much infrastructure in tests and has to select the integration levels that fit best to the project.

Consider, that more infrastructure does not automatically lead to a better bug-detection. There may be bugs in your software that are masked by bugs in the infrastructure. The best way to find those bugs is to test with very few infrastructure.

External systems do not belong to any of the integration levels defined here. OASP does not recommend involving real external systems in test automation. This means, they have to be replaced by test doubles in automated tests. An exception may be external systems that are fully under control of the own development team.

The following chapters describe the four integration levels.

4.12.4.1 Level 1 Module Test

The goal of a *isolated module test* is to provide fast feedback to the developer. Consequently, isolated module tests must not have any interaction with the client, the database, the file system, the network, etc.

An isolated module test is testing a single classes or at least a small set of classes in isolation. If such classes depend on other components or external resources, etc. these shall be replaced with a test double.

For an example see here.

4.12.4.2 Level 2 Component Test

A *component test* aims to test components or component parts as a unit. These tests typically run with a (light-weight) infrastructure such as spring-test and can access resources such as a database (e.g. for DAO tests). Further, no remote communication is intended here. Access to external systems shall be replaced by a test double.

4.12.4.3 Level 3 Subsystem Test

A *subsystem test* runs against the external interfaces (e.g. HTTP service) of the integrated subsystem. In OASP4J the server (JEE application) is the subsystem under test. The tests act as a client (e.g. service consumer) and the server has to be integrated and started in a container.

Subsystem tests of the client subsystem are described in the OASP4JS-Wiki.

If you are using spring-boot, you should use `spring-test` as leightweight and fast testing infrastructure that is already shipped with `oasp4j-test`. In case you have to use a full blown JEE application server, we recommend to use arquillian. To get started look here.

Do not confuse a *subsystem test* with a system integration test. A system integration test validates the interaction of several systems where we do not recommend test automation.

4.12.4.4 Level 4 System Test

A *system test* has the goal to test the system as a whole against its official interfaces such as its UI or batches. The system itself runs as a separate process in a way close to a regular deployment. Only external systems are simulated by test doubles.

The OASP does only give advices for automated system test. In nearly every project there must be manual system tests, too. This manual system tests are out of scope here.

4.12.4.5 Classifying Integration-Levels

OASP4J defines Category-Interfaces that shall be used as JUnit Categories. Also OSAP4J provides abstract base classes that you may extend in your test-cases if you like.

OASP4J further pre-configures the maven build to only run integration levels 1-2 by default (e.g. for fast feedback in continuous integration). It offers the profiles subsystemtest (1-3) and systemtest (1-4). In your nightly build you can simply add -Psystemtest to run all tests.

4.12.5 Deployment Pipeline

A deployment pipeline is a semi-automated process that gets software-changes from version control into production. It contains several validation steps, e.g. automated tests of all integration levels. Because

OASP4J should fit to different project types - from agile to waterfall - it does not define a standard deployment pipeline. But we recommend to define such a deployment pipeline explicitly for each project and to find the right place in it for each type of test.

For that purpose, it is advisable to have fast running test suite that gives as much confidence as possible without needing too much time and too much infrastructure. This test suite should run in an early stage of your deployment pipeline. Maybe the developer should run it even before he/she checked in the code. Usually lower integration levels are more suitable for this test suite than higher integration levels.

Note, that the deployment pipeline always should contain manual validation steps, at least manual acceptance testing. There also may be manual validation steps that have to be executed for special changes only, e.g. usability testing. Management and execution processes of those manual validation steps are currently not in the scope of OASP.

4.12.6 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.

4.13 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.

4.13.1 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.

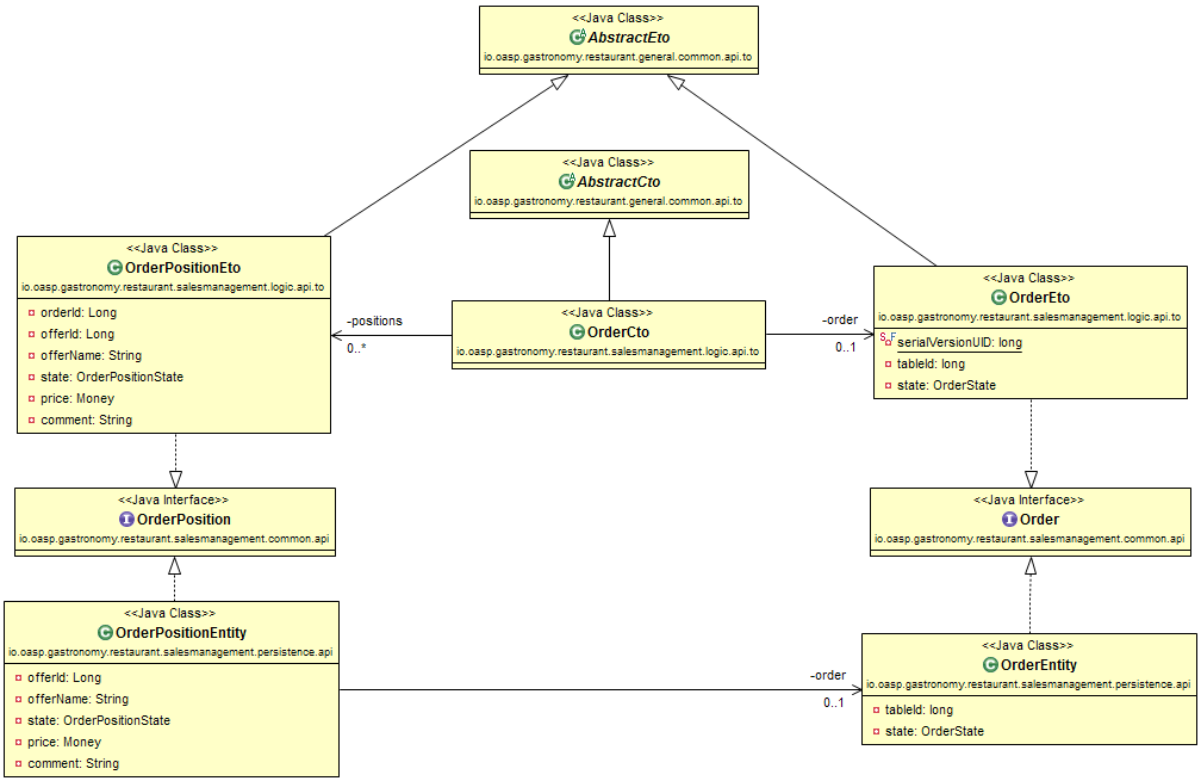


Figure 4.3. ETOs and CTOs

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.

4.13.2 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).

4.14 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.

4.14.1 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>
```

4.14.2 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.

4.15 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.

4.15.1 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.

4.15.2 Datatypes in Entities

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

4.15.3 Datatypes in Transfer-Objects

4.15.3.1 XML

For mapping datatypes with JAXB see XML guide.

4.15.3.2 JSON

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

4.16 Transaction Handling

Transactions are technically processed by the data access 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 logic layer to annotate business methods that participate in transactions (what typically applies to all business components).

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

4.16.1 Batches

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

4.17 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

http://www.de.capgemini.com/public-sector/igov