devonfw

devonfw for Java 3.0.0

The devonfw community

Version 3.0.0, 2018-12-13_13.14.55

Table of Contents

Introduction

The *devonfw* 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 devon4j, the Java stack of devonfw. 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 devonfw. Then we go into details of the the architecture of an application.

1.1. Key Principles

For the devonfw 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 been proven 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 devonfw 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 devonfw 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

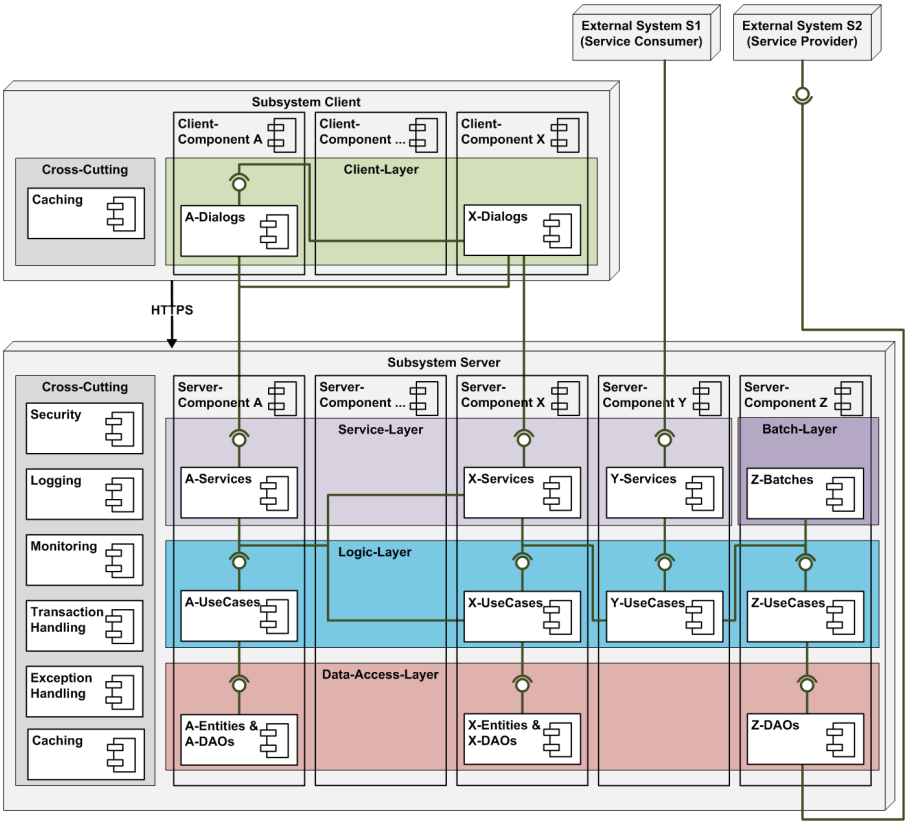


Figure 1. Technical Reference Architecture

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 devonfw 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).

Technology Stack

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

Table 1. Technology Stack of devonfw

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

Topic	Detail	Standard	Suggested implementation
persistence	OR-mapper	JPA	hibernate
batch	framework	JSR352	spring-batch
service	SOAP services	JAX-WS	CXF
service	REST services	JAX-RS	CXF
logging	framework	slf4j	logback
validation	framework	beanvalidation/JSR303	hibernate-validator
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.).
- For package segments and type names prefer singular forms (`CustomerEntity` instead of ~~`CustomersEntity`~~). Only use plural forms when there is no singular or it is really semantically required (e.g. for a container that contains multiple of such objects).
- 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`, `V=Value` 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 devonfw based application we use the following Java-Package schema:

```
«rootpackage».«application».«component».«layer».«scope»[.«detail»]*
```

E.g. in our example application we find the Spring Data repositories for the `ordermanagement` component in the package `com.devonfw.application.mtsj.ordermanagement.dataaccess.api.repo`

Table 2. Segments of package schema

Segment	Description	Example
«rootpackage»	Is the basic Java Package name-space of the organization or IT project owning the code following common Java Package conventions. Consists of multiple segments corresponding to the Internet domain of the organization.	`com.devonfw.application.mtsj`
«application»	The name of the application build in this project.	`devonfw`
«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`

Segment	Description	Example
«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 `com.devonfw.module` as `«basepackage»` and the name of the module as `«component»`. E.g. the API of our `beanmapping` module can be found in the package `com.devonfw.module.beanmapping.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.

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.

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.

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.

2.1.5. Code-Style

This section gives you best practices to write better code and avoid pitfalls and mistakes.

BLOBs

Avoid using `byte[]` for BLOBs as this will load them entirely into your memory. This will cause performance issues or out of memory errors. Instead use streams when dealing with BLOBs. For further details see BLOB support.

Closing Resources

Resources such as streams (`InputStream`, `OutputStream`, `Reader`, `Writer`) or transactions need to be handled properly. Therefore it is important to follow these rules:

- Each resource has to be closed properly, otherwise you will get out of file handles, TX sessions, memory leaks or the like
- Where possible avoid to deal with such resources manually. That is why we are recommending `@Transactional` for transactions in devonfw (see Transaction Handling).
- In case you have to deal with resources manually (e.g. binary streams) ensure to close them properly. See the example below for details.

Closing streams and other such resources is error prone. Have a look at the following example:

```
try {
  InputStream in = new FileInputStream(file);
  readData(in);
  in.close();
} catch (IOException e) {
  throw new RuntimeIoException(e, IoMode.READ);
}
```

The code above is wrong as in case of an `IOException` the `InputStream` is not properly closed. In a server application such mistakes can cause severe errors that typically will only occur in production. As such resources implement the `AutoCloseable` interface you can use the `try-with-resource` syntax to write correct code. The following code shows a correct version of the example:

```
try (InputStream in = new FileInputStream(file)) {
  readData(in);
} catch (IOException e) {
  throw new RuntimeIoException(e, IoMode.READ);
}
```

Lambdas and Streams

With Java8 you have cool new feautres like lambdas and monads like (`Stream`, `CompletableFuture`, `Optional`, etc.). However, these new features can also be misused or lead to code that is hard to read or debug. To avoid pain, we give you the following best practices:

1. Learn how to use the new features properly before using. Often developers are keen on using cool new features. When you do your first experiments in your project code you will cause deep pain and might be ashamed afterwards. Please study the features properly. Even Java8 experts still write for loops to iterate over collections, so only use these features where it really makes sense.
2. Streams shall only be used in fluent API calls as a Stream can not be forked or reused.
3. Each stream has to have exactly one terminal operation.
4. Do not write multiple statements into lambda code:

   ```
   collection.stream().map(x -> {
   Foo foo = doSomething(x);
   ...
   return foo;
   }).collect(Collectors.toList());
   ```

   This style makes the code hard to read and debug. Never do that! Instead extract the lambda body to a private method with a meaningful name:

```
collection.stream().map(this::convertToFoo).collect(Collectors.toList());
```

5. Do not use `parallelStream()` in general code (that will run on server side) unless you know exactly what you are doing and what is going on under the hood. Some developers might think that using parallel streams is a good idea as it will make the code faster. However, if you want to do performance optimizations talk to your technical lead (architect). Many features such as security and transactions will rely on contextual information that is associated with the current thread. Hence, using parallel streams will most probably cause serious bugs. Only use them for standalone (CLI) applications or for code that is just processing large amounts of data.
6. Do not perform operations on a sub-stream inside a lambda:

```
set.stream().flatMap(x -> x.getChildren().stream().filter(this::isSpecial)).
collect(Collectors.toList()); // bad
set.stream().flatMap(x -> x.getChildren().stream()).filter(this::isSpecial).
collect(Collectors.toList()); // fine
```

7. Only use `collect` at the end of the stream:

```
set.stream().collect(Collectors.toList()).forEach(...) // bad
set.stream().peek(...).collect(Collectors.toList()) // fine
```

8. Lambda parameters with Types inference

```
(a,b,c)  -> a.toString() + Float.toString(b) + Arrays.toString(c)  // fine
(String a, Float b, Byte[] c) -> a.toString() + Float.toString(b) + Arrays.
toString(c)  //bad

Collections.sort(personList, (p1, p2) -> p1.getSurName().compareTo(p2.getSurName()
));  //fine
Collections.sort(personList, (Person p1, Person p2) -> p1.getSurName().compareTo(
p2.getSurName()));  //bad
```

9. Avoid Return Braces and Statement

```
 (a) ->  a.toString();   // fine
 (a) ->  { return a.toString(); } //bad
```

10. Avoid Parentheses with Single Parameter

```
 a -> a.toString();  // fine
(a) -> a.toString(); //bad
```

11. Avoid if/else inside foreach method. Use Filter method & comprehension

```
Bad
static public Iterator<String> TwitterHandles(Iterator<Author> authors, string
company) {
    final List result = new ArrayList<String> ();
    foreach (Author a : authors) {
      if (a.Company.equlas(company)) {
        String handle = a.TwitterHandle;
        if (handle != null)
          result.Add(handle);
      }
    }
    return result;
  }
```

```
Fine
public List<String> twitterHandles(List<Author> authors, String company) {
    return authors.stream()
            .filter(a -> null != a && a.getCompany().equals(company))
            .map(a -> a.getTwitterHandle())
            .collect(toList());
  }
```

Optionals

With `Optional` you can wrap values to avoid a `NullPointerException` (NPE). However, it is not a good code-style to use `Optional` for every parameter or result to express that it may be null. For such case use `@Nullable` or even better instead annotate `@NotNull` where `null` is not acceptable.

However, `Optional` can be used to prevent NPEs in fluent calls (due to the lack of the elvis operator):

```
Long id;
id = fooCto.getBar().getBar().getId(); // may cause NPE
id = Optional.ofNullable(fooCto).map(FooCto::getBar).map(BarCto::getBar).map(BarEto:
:getId).orElse(null); // null-safe
```

Encoding

Encoding (esp. Unicode with combining characters and surrogates) is a complex topic. Please study this topic if you have to deal with encodings and processing of special characters. For the basics follow these recommendations:

- When you have explicitly decide for an encoding always prefer Unicode (UTF-8 or better). This especially impacts your databases and has to be defined upfront as it typically can not be changed (easily) afterwards.
- Do not cast from `byte` to `char` (Unicode characters can be composed of multiple bytes, such cast may only work for ASCII characters)

- Never convert the case of a String using the default locale (esp. when writing generic code like in devonfw). E.g. if you do `"HI".toLowerCase()` and your system locale is Turkish, then the output will be "hı" instead of "hi" what can lead to wrong assumptions and serious problems. If you want to do a "universal" case conversion always use explicitly an according western locale (e.g. `toLowerCase(Locale.US)`). Consider using a library (https://github.com/m-m-m/util/blob/master/core/src/main/java/net/sf/mmm/util/lang/api/BasicHelper.java) or create your own little static utility for that in your project.
- Write your code independent form the default encoding (system property `file.encoding`) - this will most likely differ in JUnit from production environment
  - Always provide an encoding when you create a `String` from `byte[]`: `new String(bytes, encoding)`
  - Always provide an encoding when you create a `Reader` or `Writer` : `new InputStreamReader(inStream, encoding)`

Prefer general API

Avoid unnecessary strong bindings:

- Do not bind your code to implementations such as `Vector` or `ArrayList` instead of `List`
- In APIs for input (=parameters) always consider to make little assumptions:
  - prefer `Collection` over `List` or `Set` where the difference does not matter (e.g. only use `Set` when you require uniqueness or highly efficient `contains`)
  - consider prefering `Collection<? extends Foo>` over `Collection<Foo>` when `Foo` is an interface or super-class

3. Layers

3.1. Client Layer

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

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

Our main focus is on web-clients. In our sample application my-thai-star we offer a responsive web-client based on Angular following devon4ng that integrates seamlessly with the backends of my-thai-star available for Java using devon4j as well as .NET/C# using devon4net. For building angular clients read the separate devon4ng guide.

3.1.1. JavaScript for Java Developers

In order to get started with client development as a Java developer we give you some hints to get

started. Also if you are an experienced JavaScript developer and want to learn Java this can be helpful. First, you need to understand that the JavaScript ecosystem is as large as the Java ecosystem and developing a modern web client requires a lot of knowledge. The following table helps you as experienced developer to get an overview of the tools, configuration-files, and other related aspects from the new world to learn. Also it helps you to map concepts between the ecosystems. Please note that we list the tools recommended by devonfw here (and we know that there are alternatives not listed here such as gradle, grunt, bower, etc.).

Table 3. Aspects in JavaScript and Java ecosystem

Topic	Aspect	JavaScript	Java
Programming	Language	TypeScript (extends JavaScript)	Java
Runtime	VM	nodejs (or web-browser)	jvm
Dependency-Management	Tool	yarn (or npm)	maven
	Config	package.json	pom.xml
	Repository	npm repo	maven central (repo search)
Build-Management	Taskrunner	gulp	maven (or more comparable ant)
	Config	gulpfile.js (and `gulp/*`)	pom.xml (or build.xml)
	Clean cmd	gulp clean	mvn clean
	Build cmd	yarn install && gulp build:dist	mvn install (see lifecycle)
	Test cmd	gulp test	mvn test
Testing	Test-Tool	jasmine	junit
	Test-Framework	karma	junit / surefire
	Browser Testing	PhantomJS	Selenium
	Extensions	karma-*, PhantomJs for browser emulation	AssertJ,*Unit and spring-test, etc.)
Code Analysis	Code Coverage	karma-coverage (and remap-istanbul for TypeScript)	JaCoCo/EclEmma
Development	IDE	MS VS Code or IntelliJ	Eclipse or IntelliJ
	Framework	Angular (etc.)	Spring (etc.)

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-end components typically with different release and deployment cycles (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 a native Java client (JavaFx, EclipseRcp, etc.).

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

Table 4. 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 prefix it with `v' (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 create an isolated

version of the service and increment the version (e.g. v2) . In the implementation of different version 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. 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.

3.2.5. Security

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

See REST Security.

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* (see architecture overview) 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 Part

Component Part Interface

A component part is accessed through its component part interface. The API of the component part interface has to be business oriented. This means that all parameters and return types of a method have to be business transfer-objects, datatypes (String, Integer, MyCustomerNumber, etc.), or collections of these. The API may only access objects of other business components listed in the (transitive) dependencies of the declaring business component part.

First we create the interface that contains the method(s) with the business operations documented with JavaDoc.

There are two ways of designing a component part interface at the logic layer. Depending on the application's complexity one of the following approaches should be consistently applied (i.e. you should not use both approaches within the same application).

- Component Part with Simple Interface
- Component Part Interface with Use Case Decomposition

Component Part with Simple Interface

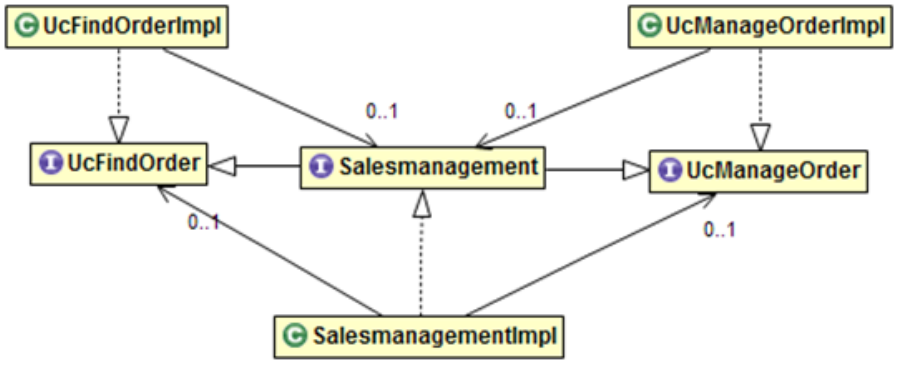
For less complex apps with fairly simple component interfaces (even if it contains many methods, e.g. several find methods), you put all methods to be exposed directly into a single interface. The implementation of the component part interface provides all the corresponding methods in one class.

Here is an example of a simple interface:

```java
/**
 * ... StaffManagement.java
 */
public interface StaffManagement {

  /**
   * @param id the {@link StaffMemberEto#getId() ID} of the requested staff member.
   * @return The {@link StaffMemberEto} with the given <code>id</code> or {@code null}
if no such object exists.
   */
  StaffMemberEto findStaffMember(Long id);

  /**
   * @param login The {@link StaffMemberEto#getName() login} of the requested staff
member.
   * @return The {@link StaffMemberEto} with the given <code>login</code> or {@code
null} if no such object exists.
   */
  StaffMemberEto findStaffMemberByLogin(String login);

  ...
}
```

Component Part Interface with Use Case Decomposition

For complex applications, component part interfaces consisting of many different use cases, it is recommended to further sub-divide it into separate use-case-interfaces to be aggregated in the main component interface. This suits for better maintainability.



The component part interface then extends the available use case interfaces to offer a single interface to the next higher layer, e.g. the service layer. Then, the implementation of the component part interface holds references to all use cases and only delegates method calls. All business logic and data-layer access is performed within the implementations of the use cases. Also, if a use case needs to use functionality of another use case provided by the same layer it will use a reference to the component part interface and *not* to the use case itself.

```
/**
 * ... Salesmanagement.java
 */
public interface Salesmanagement extends UcChangeTable, UcFindBill, UcFindOrder,
UcFindOrderPosition, UcManageBill,
    UcManageOrder, UcManageOrderPosition {

}

// ...

/**
 * ... UcChangeTable.java
 */
public interface UcChangeTable {

  /**
   * UseCase to change from one {@link TableEto table} to another. The people sitting
at a table are identified by their
   * {@link OrderEto order} that has to be provided as argument.
   *
   * @param orderId the {@link OrderEto order}
   * @param newTableId the new {@link TableEto table} to switch to.
   */
  void changeTable(long orderId, long newTableId);

}
```

3.3.2. Component Implementation

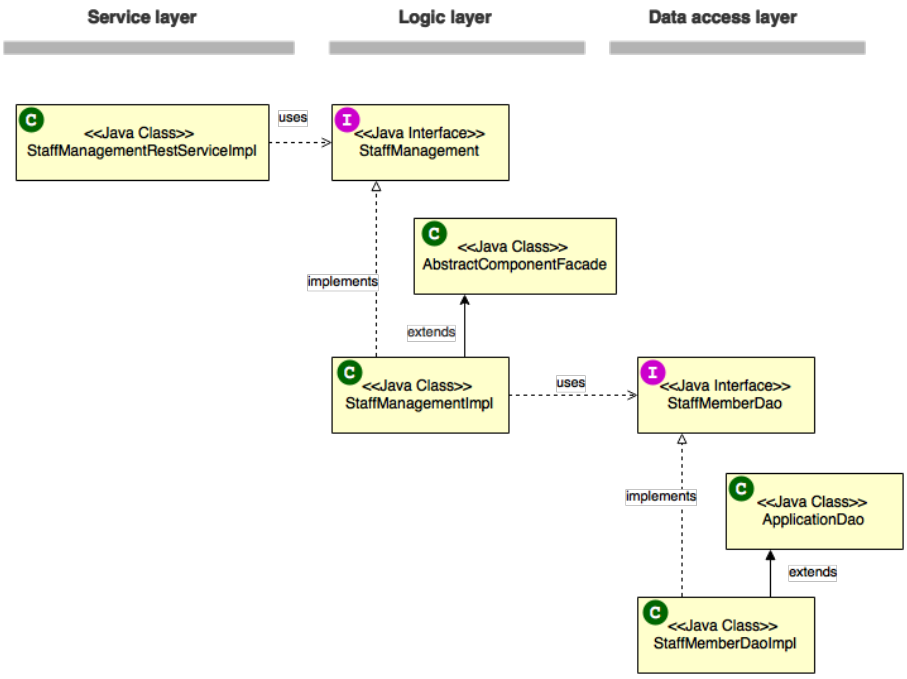
The implementation of the use case typically needs access to the persistent data. This is done by injecting the corresponding DAO. According to the principle *data sovereignty* , only DAOs of the same business component may be accessed directly from the use case. For accessing data from other components 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 Ordermanagement methods:

```
@Named
@Transactional
public class OrdermanagementImpl extends AbstractComponentFacade implements
Ordermanagement {

  @Inject
  private OrderRepository orderRepository;

  @Override
  public OrderCto findOrder(Long id) {

    LOG.debug("Get Order with id {} from database.", id);
    OrderEntity entity = this.orderRepository.find(id);
    OrderCto cto = new OrderCto();
    cto.setBooking(getBeanMapper().map(entity.getBooking(), BookingEto.class));
    cto.setHost(getBeanMapper().map(entity.getHost(), BookingEto.class));
    cto.setOrderLines(getBeanMapper().mapList(entity.getOrderLines(), OrderLineCto
.class));
    cto.setOrder(getBeanMapper().map(entity, OrderEto.class));
    cto.setInvitedGuest(getBeanMapper().map(entity.getInvitedGuest(), InvitedGuestEto
.class));
    return cto;
  }
```

As you can see, provided entities are mapped to corresponding business objects (e.g. `BookingEto.class`). Further details about this can be found in bean-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 shall not be passed to the outside of the dataaccess layer for the reasons of data sovereignty. Therefore we are using transfer-objects (TO) with the same attributes as the persistence entity. The packages are:

Persistence Entities	«rootpackage».«application».«component».dataaccess.api.entity
Transfer Objects(TOs)	«rootpackage».«application».«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. Every method exposed in an interface needs to be annotated with an authorization check, stating what role(s) a caller must provide in order to be allowed to make the call. The authorization concept is described here.

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 authorization 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 is 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 load 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 and 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. RDBMS

The classical approach is to use a Relational Database Management System (RDMS). In such case we strongly recommend to follow our JPA Guide. In case you are using Oracle you should also consider the Oracle guide.

3.4.2. NoSQL

In case of specific demands and requirements you may want to choose for a *Not only SQL* database (NoSQL). There are different categories of such products so you should first be aware what fits your requirements best:

- key/value DB
- document DB
- graph DB
- wide-column DB

As there are many such products and the market is evolving very fast, we do not yet give clear recommendations here. If you are doing a devon project and consider NoSQL please contact us for further details.

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.

devonfw uses Spring Batch as batch framework.

This guide explains how Spring Batch is used in devonfw applications. Please note that it is not yet 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.

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. 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 *billExport* belongs to the salesmanagement business component. So the *billExport* 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 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.

Batch administration and execution

Starting and Stopping Batches

Spring Batch provides a simple command line API for execution and parameterization of batches, the `CommandLineJobRunner`. It is not yet fully compatible with Spring Boot, however. For those using Spring Boot, devonfw provides the `SpringBootBatchCommandLine` with similar functionalities.

Both execute batches 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(s)

The location of the JavaConfig classes (usually annotated with `@Configuration` or `@SpringBootApplication`) and/or XML files that will be used to create an `ApplicationContext`.

The CommandLineJobRunner only accepts one class/file, which must contain everything needed to run a job (potentially by referencing other classes/files), the SpringBootBatchCommandLine, however, expects that there are two paths given: one for the general batch setup and one for the XML file containing the batch job to be executed.

There is an example of a general batch setup for Spring Boot in the my-thai-star batch module. The main class is `SpringBootBatchApp`, which also imports the general configuration class introduced in the chapter on the general configuration. Note that `SpringBootBatchApp` deactivates the evaluation of annotations used for authorization, especially the `@RolesAllowed` annotation. You should of course make sure that only authorized users can start batches, but once the batch is started there is usually no need to check any authorization.

jobName

The name of the job to be run.

All arguments after the job name are considered to be job parameters and must be in the format of `name=value`:

Example for the CommandLineJobRunner:

```
java org.springframework.batch.core.launch.support.CommandLineJobRunner
classpath:config/app/batch/beans-billexport.xml billExportJob -outputFile=file:out.csv
date(date)=2015/12/20
```

Example for the SpringBootBatchCommandLine:

```
java com.devonfw.module.batch.common.base.SpringBootBatchCommandLine
com.devonfw.application.mtsj.SpringBootBatchApp classpath: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 non identifying parameters).

When trying to restart a batch that was already complete, 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: …"`).

Stopping a Job

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

```
java org.springframework.batch.core.launch.support.CommandLineJobRunner
classpath:config/app/batch/beans-billexport.xml –stop billExportJob
```

or

```
java com.devonfw.module.batch.common.base.SpringBootBatchCommandLine
com.devonfw.application.mtsj.SpringBootBatchApp classpath:config/app/batch/beans-
billexport.xml billExportJob –stop
```

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

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 devonfw 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.

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 partly not and there needs to be some sort of plan on 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 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 execute 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 on how to deal with performance problems.

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

For configuring batches, we recommend not to use annotations (would not work very well for batches) or JavaConfig, but XML, because this makes the whole batch configuration more transparent, as its structure and implementing beans are immediately visible. Moreover the Spring Batch documentation focuses rather on XML based configurations than on JavaConfig.

For explanations on how these XML files are build in general, have a look at the spring documentation.

There is, however, some general configuration needed for all batches, for which we use JavaConfig, as it is also used for the setup of all other layers. You can find an example of such a configuration in the `samples/core` project: `BeansBatchConfig`. In this section, we will explain the most important parts of this class.

The `jobRepository` is used to update the meta data tables.

The database type can optionally be set on the `jobRepository` for correctly handling database specific things using the `setDatabaseType` method. Possible values are oracle, mysql, postgres etc.

If the size of all three columns, which by default have a length limitation of 2500, has been increased as proposed here, the property maxVarCharLength should be adjusted accordingly using the corresponding setter method 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 (`JobLauncherWithAdditionalRestartCapabilities`) here, which can be found in the module `modules/batch` (`devon4j-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 of type `JobRegistryBeanPostProcessor` automatically.

A `JobParametersIncremeter` (bean `incrementer`) can be used to generate unique parameters, see restarts and parameters for further details. It should be configured manually for each batch job, see example batch below, otherwise exceptions might occur when starting batches.

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 configures the batch for exporting the bills.

As you can see, there is a job element (`billExportJob`), which contains the two step elements (`createCsvFile` and `deleteBills`). Note that for every step you have to explicitly specify which step comes next (using the next attribute), unless it is the last step.

The step elements always contains a tasklet element, even if chunk processing is used. The transaction-attributes element is especially used to set timeout 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 containing a reference to a tasklet.

In the example above the `ItemReader` named `unprocessedBillsReader` always reads 1000 ids of unprocessed bills (via a DAO) and returns them one after another. The `ItemProcessor` `processedMarker` 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` `csvFileWriter` (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).

The `chunkLoggingListener`, which is also used in the example above, can be utilized for all chunk steps to log exceptions together with the items where these exceptions occurred (see listeners for further details on listeners). It's implementation can be found in the module modules/batch. Note that classes used for items have to have an appropriate `toString()` method in order for this listener to be useful.

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 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. By 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, 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 instructing 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.

By 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` (`JobLauncherWithAdditionalRestartCapabilities`) 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 completed successfully

before executing the next regular 'run'. When scheduling batches, you should make that sure.

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. One has to be careful , while 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 example batch).

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 process 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 which item caused the exception (as it has not been read yet). That is why a reader should contain as little processing logic as possible, minimizing the potential for failures.

Caching

By 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's 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 ready 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` implementations 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 as it does not change (you can e.g. sort the entities by their primary key).

An `ItemReader` 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 access 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` 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 hibernate's 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 devonfw applications, as these mostly require direct access to an `EntityManager` or a JDBC connection for example. You should think twice when using them and not 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. By 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` on 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`/`ItemWriter` and the code needed for properly handling failures (restarts for example) should be encapsulated there.

It is usually a good practice to make `ItemProcessor` implementations stateless, as the process method might be called more than once for one item (see the section on `ItemReader` 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 this 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 there might be changes not written out yet and therefore constraints were not checked yet. 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 contain too many lines (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` and `ItemProcessor`: 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.

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 'fine 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).

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 retries are exhausted (by default, there is no retry) and the exception that occurred cannot be skipped (by 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 yet) 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 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 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 instances of `ItemWriter` 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.

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

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.

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` or `SpringBootBatchCommandLine`, 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 propose 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 adapt the concept of 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.

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.

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 extend the `AbstractSpringBatchIntegrationTest` class. Annotation used to load the job's `ApplicationContext`:

`@SpringBootTest(classes = {…})`: Indicates which JavaConfig classes (attribute `classes`) `@ImportResource("classpath:../sample_BatchContext.xml") : Indicates XML files that contain the ``ApplicationContext`. Use `@ContextConfiguration(…)` if Spring Boot is not used.

```
public abstract class AbstractSpringBatchIntegrationTest extends AbstractComponentTest
{..}
```

```
@SpringBootTest(classes = { SpringBootBatchApp.class }, webEnvironment =
WebEnvironment.RANDOM_PORT)
@ImportResource("classpath:config/app/batch/beans-productimport.xml")
@EnableAutoConfiguration
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`.

```
@SpringBootTest(classes = { SpringBootBatchApp.class }, webEnvironment =
WebEnvironment.RANDOM_PORT)
@ImportResource("classpath:config/app/batch/beans-productimport.xml")
@EnableAutoConfiguration
public class ProductImportJobTest extends AbstractSpringBatchIntegrationTest {

  @Inject
  private Job productImportJob;

  @Test
  public void testJob() throws Exception {
    ......
    ......
    JobExecution jobExecution = getJobLauncherTestUtils(this.productImportJob)
.launchJob(jobParameters);
    assertThat(jobExecution.getStatus()).isEqualTo(BatchStatus.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 devonfw. 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 BatchStatus is still COMPLETED. That is one reason why you should always check whether 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 (also known as 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 (Contexts 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 environment). However, we recommend to use the springframework as container. But we never use proprietary annotations such as `@Autowired` instead of standardized annotations like `@Inject`. Generally we avoid proprietary annotations in business code (`common` and logic layer).
- **Simple Injection-Style**
  In general you can choose between constructor, setter or field injection. For simplicity we recommend to do private field injection as it is very compact and easy to maintain. We believe that constructor injection is bad for maintenance especially in case of inheritance (if you change the dependencies you need to refactor all sub-classes). Private field injection and public setter injection are very similar but setter injection is much more verbose (often you are even forced to have javadoc for all public methods). If you are writing re-usable library code setter injection will make sense as it is more flexible. In a business application you typically do not need that and can save a lot of boiler-plate code if you use private field injection instead. Nowadays you are using container infrastructure also for your tests (see spring integration tests) so there is no need to inject manually (what would require a public setter).
- **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 {
  @Inject
  private 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 field 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
private List<MyConverter> converters;
```

Please note that when writing library code instead of annotating implementation with `@Named` it is better to provide `@Configuration` classes that choose the implementation via `@Bean` methods (see @Bean documentation). This way you can better "export" specific features instead of relying library users to do a component-scan to your library code and loose control on upgrades.

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).

Spring Boot Application

The devonfw recommends using spring-boot to build web applications. For a complete documentation see the Spring Boot Reference Guide.

With spring-boot you provide a simple *main class* (also called starter class) like this: com.devonfw.mtsj.application

```
@SpringBootApplication(exclude = { EndpointAutoConfiguration.class })
@EntityScan(basePackages = { "com.devonfw.mtsj.application" }, basePackageClasses = {
AdvancedRevisionEntity.class })
@EnableGlobalMethodSecurity(jsr250Enabled = true)
@ComponentScan(basePackages = { "com.devonfw.mtsj.application.general",
"com.devonfw.mtsj.application" })
public class SpringBootApp {

  /**
   * Entry point for spring-boot based app
   *
   * @param args - arguments
   */
  public static void main(String[] args) {

    SpringApplication.run(SpringBootApp.class, args);
  }
}
```

In an devonfw application this main class is always located in the `<basepackage>` of the application package namespace (see package-conventions). This is because a spring boot application will automatically do a classpath scan for components (spring-beans) and entities in the package where the application main class is located including all sub-packages. You can use the `@ComponentScan` and `@EntityScan` annotations to customize this behaviour.

Standard beans configuration

For basic bean configuration we rely on spring boot using mainly configuration classes and only 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 non-invasive, 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 (see also `exclude` attribute of `@SpringBootApplication` in example code above).

Beans are configured via annotations in your java code (see dependency-injection).

For technical configuration you will typically write additional spring config classes annotated with `@Configuration` that provide bean implementations via methods annotated with `@Bean`. See spring @Bean documentation for further details. Like in XML you can also use `@Import` to make a `@Configuration` class include other configurations.

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`

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. See batch-layer for further details.

BeanMapper Configuration

In the directory `src/main/resources/config/app/common` we place the configuration for the bean-mapping. See bean-mapper configuration for further details.

Security configuration

The abstract base class `BaseWebSecurityConfig` should be extended to configure web application security thoroughly. A basic and secure configuration is provided which can be overridden or extended by subclasses. Subclasses must use the `@Profile` annotation to further discriminate between beans used in production and testing scenarios. See the following example:

How to extend `BaseWebSecurityConfig` for Production and Test

```
@Configuration
@EnableWebSecurity
@Profile(SpringProfileConstants.JUNIT)
public class TestWebSecurityConfig extends BaseWebSecurityConfig {...}

@Configuration
@EnableWebSecurity
@Profile(SpringProfileConstants.NOT_JUNIT)
public class WebSecurityConfig extends BaseWebSecurityConfig {...}
```

See WebSecurityConfig.

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 your.path.to.the.websocket.config;
...
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig extends AbstractWebSocketMessageBrokerConfigurer {
...
```

Database Configuration

To choose database of your choice , set `spring.profiles.active=XXX` in `src/main/resources/config/application.properties`. Also, one has to set all the active spring profiles in this `application.properties` and not in any of the other `application.properies`.

4.2.2. Externalized Configuration

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

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 (see common application properties for a list of properties defined by the spring framework). These properties are explained in the corresponding configuration sections of the guides for each topic:

- persistence configuration
- service configuration
- logging guide

For a general understanding how spring-boot is loading and boostrapping your `application.properties` see spring-boot external configuration. The following properties files are used in every devonfw application:

- `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 used at development time (for all local deployment scenarios). This property file is excluded from all packaging.
- `src/test/resources/config/application.properties` providing additional properties only used for testing (JUnits based on spring test).

For other environments where the software gets deployed such as `test`, `acceptance` and `production`

you need to provide a tailored copy of `application.properties`. The location depends on the deployment strategy:

- standalone run-able Spring Boot App using embedded tomcat: `config/application.properties` under the installation directory of the spring boot application.
- dedicated tomcat (one tomcat per app): `$CATALINA_BASE/lib/config/application.properties`
- tomcat serving a number of apps (requires expanding the wars): `$CATALINA_BASE/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.

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 hierarchy 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`

4.2.3. Security

Often you need to have passwords (for databases, third-party services, etc.) as part of your configuration. These are typically environment specific (see above). However, with DevOps and continuous-deployment you might be tempted to commit such configurations into your version-control (e.g. `git`). Doing that with plain text passwords is a severe problem especially for production systems. Never do that! Instead we offer some suggestions how to deal with sensible configurations:

Password Encryption

A simple but reasonable approach is to configure the passwords encrypted with a master-password. The master-password should be a strong secret that is specific for each environment. It must never be committed to version-control. In order to support encrypted passwords in spring-boot `application.properties` all you need to do is to add jasypt-spring-boot as dependency in your `pom.xml`(please check for recent version):

```
<dependency>
  <groupId>com.github.ulisesbocchio</groupId>
  <artifactId>jasypt-spring-boot-starter</artifactId>
  <version>1.17</version>
</dependency>
```

This will smoothly integrate jasypt into your spring-boot application. Read this HOWTO to learn how to encrypt and decrypt passwords using jasypt. Here is a simple example output of an enctrypted password (of course you have to use strong passwords instead of `secret` and `postgres` - this is only an example):

```
----ARGUMENTS-------------------

input: postgres
password: secret

----OUTPUT----------------------

jd5ZREpBqxuN9ok0IhnXabgw7V3EoG2p
```

The master-password can be configured on your target environment via the property `jasypt.encryptor.password`. As system properties given on the command-line are visible in the process list, we recommend to use an `config/application.yml` file only for this purpose (as we recommended to use `application.properties` for regular configs):

```
jasypt:
    encryptor:
        password: secret
```

(of course you will replace `secret` with a strong password). In case you happen to have multiple apps on the same machine, you can symlink the `application.yml` from a central place. Now you are able to put encrypted passwords into your `application.properties`

```
spring.datasource.password=ENC(jd5ZREpBqxuN9ok0IhnXabgw7V3EoG2p)
```

To prevent jasypt to throw an exception in dev or test scenarios simply put this in your local config (`src/main/config/application.properties` and same for `test`, see above for details):

```
jasypt.encryptor.password=none
```

Is this Security by Obscurity?

- Yes, from the point of view to protect the passwords on the target environment this is nothing but security by obscurity. If an attacker somehow got full access to the machine this will only cause him to spend some more time.
- No, if someone only gets the configuration file. So all your developers might have access to the version-control where the config is stored. Others might have access to the software releases that include this configs. But without the master-password that should only be known to specific operators none else can decrypt the password (except with brute-force what will take a very long time, see jasypt for details).

4.3. Java Persistence API

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. As we use JPA we abstract from SQL here. However, you will still need a DDL script for your schema and during maintenance also database migrations. Please follow our SQL guide for such artefacts.

4.3.1. 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. A 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 ApplicationPersistenceEntity implements Message{

  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`. ere 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. 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 the application is deployed to production. 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;
}
```

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`.

QueryDSL and Custom Types

Using the Aliases API of QueryDSL might result in an `InvalidDataAccessApiUsageException` when using custom datatypes in entity properties. This can be circumvented in two steps:

1. Ensure you have the following maven dependencies in your project (`core` module) to support custom types via the Aliases API:

```
<dependency>
  <groupId>org.ow2.asm</groupId>
  <artifactId>asm</artifactId>
</dependency>
<dependency>
  <groupId>cglib</groupId>
  <artifactId>cglib</artifactId>
</dependency>
```

2. Make sure, that all your custom types used in entities provide a non-argument constructor with at least visibility level `protected`.

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 `com.devonfw.<projectName>.general.dataaccess.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)`). Be sure to include "AUTO_INCREMENT" in your sql table field ID to be able to persist data (or similar for other databases).

4.3.2. 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 devon4j 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 devon4j 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 Orders 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 devon4j. Instead a third entity has to be introduced, which "represents" the relationship (it has two n:1 relationships).

Eager vs. Lazy Loading

Using JPA 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 devon are strongly advised to always use lazy loading. The JPA defaults are:

- `@OneToMany`: LAZY
- `@ManyToMany`: LAZY
- `@ManyToOne`: EAGER
- `@OneToOne`: EAGER

So at least for `@ManyToOne` and `@OneToOne` you always need to override the default by providing `fetch = FetchType.LAZY`. IMPORTANT: Please read the performance guide.

Cascading Relationships

For relations it is also possible to define whether operations are cascaded (like a recursion) to the related entity. By default, nothing is done in these situations. This can be changed by using the `cascade` property of the annotation that specifies the relation type (`@OneToOne`, `@ManyToOne`, `@OneToMany`, `@ManyToOne`). This property accepts a `CascadeType` that offers the following options:

- PERSIST (for `EntityManager.persist`, relevant to inserted transient entities into DB)
- REMOVE (for `EntityManager.remove` to delete entity from DB)
- MERGE (for `EntityManager.merge`)
- REFRESH (for `EntityManager.refresh`)
- DETACH (for `EntityManager.detach`)
- ALL (cascade all of the above operations)

See here for more information.

4.3.3. Embeddable

An embeddable Object is a way to group properties of an entity into a separate Java (child) object. Unlike with implement relationships the embeddable is not a separate entity and its properties are stored (embedded) in the same table together with the entity. This is helpful to structure and reuse groups of properties.

The following example shows an `Address` implemented as an embeddable 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
}
```

As you can see an embeddable is similar to an entity class, but with an `@Embeddable` annotation instead of the `@Entity` annotation and without primary key or modification counter.

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 this `Address` inside an entity class can be done like this:

```
  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 of the embeddable (here `Address`) are `null`, then the embeddable object itself is also `null` inside the entity. This has to be considered to avoid NullPointerException's. Further this causes some issues with primitive types in embeddable classes that can be avoided by only using object types instead.

4.3.4. 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 class hierarchy to database tables. Generic abstract super-classes for entities can simply be annotated with `@MappedSuperclass`.

For all other cases the JPA offers the annotation `@Inheritance` with the property `strategy` talking an `InheritanceType` that has the following options:

- `SINGLE_TABLE`: This strategy uses a single table that contains all columns needed to store all entity-types of the entire inheritance 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`).
- `TABLE_PER_CLASS`: 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 super type, joins are needed.
- `JOINED`: In this case there is a table for every entity class including abstract classes, which contains only the columns for the persistent properties of that particular class. 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.

Each 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, searching or persisting entities. 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 inheritance strategy has to be annotated to the top-most entity of the class hierarchy (where `@MappedSuperclass`es are not considered) like in the following example:

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public abstract class MyParentEntity extends ApplicationPersistenceEntity implements
MyParent {
  ...
}

@Entity
public class MyChildEntity extends MyParentEntity implements MyChild {
  ...
}

@Entity
public class MyOtherEntity extends MyParentEntity implements MyChild {
  ...
}
```

As a best practice we advise you to avoid entity hierarchies at all where possible and otherwise to keep the hierarchy as small as possible. In order to just ensure reuse or establish a common API you can consider a shared interface, a `@MappedSuperclass` or an `@Embeddable` instead of an entity hierarchy.

4.3.5. Repositories and DAOs

For each entity a code unit is created that groups all database operations for that entity. We recommend to use spring-data respositories for that as it is most efficient for developers. As an alternative there is still the classic approach using DAOs.

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 access a database, anomalies may happen, e.g. a transaction is able to see changes from another transaction although that one did, not yet 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).

Another 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).

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

Optimistic Locking

The class `com.devonfw.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 address this , 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 released. 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.

4.3.6. Database Auditing

See auditing guide.

4.3.7. Testing Entities and DAOs

See testing guide.

4.3.8. 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.

4.3.9. Database Configuration

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

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. In case you want to use Oracle RDBMS you can find additional hints here.

Database Migration

See database migration.

Database Logging

Add the following properties to `application.properties` to enable logging of database queries for debugging purposes.

```
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.use_sql_comments=true
spring.jpa.properties.hibernate.format_sql=true
```

Pooling

You typically want to pool JDBC connections to boost performance by recycling previous connections. There are many libraries available to do connection pooling. We recommend to use HikariCP. For Oracle RDBMS see here.

4.3.10. Security

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.

Limited Permissions for Application

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

4.3.11. Queries

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

The simplest CRUD-Queries (e.g. find an entity by its ID) are already build in the devonfw CRUD functionality (via Repository or DAO). For other cases you need to write your own query. We distinguish between *static* and *dynamic* queries. Static queries have a fixed JPQL query string that may only use parameters to customize the query at runtime. Instead, dynamic queries can change their clauses (`WHERE`, `ORDER BY`, `JOIN`, etc.) at runtime depending on the given search criteria.

Static Queries

E.g. to find all DishEntries (from MTS sample app) that have a price not exceeding a given `maxPrice` we write the following JPQL query:

```
SELECT dish FROM DishEntity dish WHERE dish.price <= :maxPrice
```

Here `dish` is used as alias (variable name) for our selected `DishEntity` (what refers to the simple name of the Java entity class). With `dish.price` we are referring to the Java property `price` (`getPrice()`/`setPrice(…)`) in `DishEntity`. A named variable provided from outside (the search criteria at runtime) is specified with a colon (`:`) as prefix. Here with `:maxPrice` we reference to a variable that needs to be set via `query.setParameter("maxPrice", maxPriceValue)`. JPQL also supports indexed parameters (`?`) but they are discouraged because they easily cause confusion and mistakes.

Using Queries to Avoid Bidirectional Relationships

With the usage of queries it is possible to avoid exposing relationships or modelling bidirectional relationships, which have some disadvantages (see relationships). This is especially desired for relationships between entities of different business components. So for example to get all OrderLineEntities for a specific OrderEntity without using the `orderLines` relation from `OrderEntity` the following query could be used:

```
SELECT line FROM OrderLineEntity line WHERE line.order.id = :orderId
```

Dynamic Queries

For dynamic queries we 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<DishEntity> findOrders(DishSearchCriteriaTo criteria) {
    DishEntity dish = Alias.alias(DishEntity.class);
    JPAQuery<OrderEntity> query = newDslQuery(alias); // new
JPAQuery<>(getEntityManager()).from(Alias.$(dish));
    Range<BigDecimal> priceRange = criteria.getPriceRange();
    if (priceRange != null) {
      BigDecimal min = priceRange.getMin();
      if (min != null) {
        query.where(Alias.$(order.getPrice()).ge(min));
      }
      BigDecimal max = priceRange.getMax();
      if (max != null) {
        query.where(Alias.$(order.getPrice()).le(max));
      }
    }
    String name = criteria.getName();
    if ((name != null) && (!name.isEmpty())) {
      // query.where(Alias.$(alias.getName()).eq(name));
      QueryUtil.get().whereString(query, Alias.$(alias.getName()), name, criteria
.getNameOption());
    }
    return query.fetch();
  }
```

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. devonfw provides this on a lower level by LikePatternSyntax and on a high level by QueryUtil (see QueryHelper.newStringClause(…)).

Pagination

devonfw provides pagination support. If you are using spring-data repositories you will get that directly from spring for static queries. Otherwise for dynamic or generally handwritten queries we provide this via QueryUtil.findPaginated(…):

```
boolean determineTotalHitCount = ...;
return QueryUtil.get().findPaginated(criteria.getPageable(), query,
determineTotalHitCount);
```

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 mythaistar/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"
        }
    ]
}
```

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 mythaistar/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 that are provided via `SearchCriteriaTo`. Besides paging (see above) we also get timeout support.

Advanced Queries

Writing queries can sometimes get rather complex. The current examples given above only showed very simple basics. Within this topic a lot of advanced features need to be considered like:

- Joins
- Constructor queries
- Order By (Sorting)

- Grouping
- Having
- Unions
- Sub-Queries
- Aggregation functions like e.g. count/avg/sum
- Distinct selections
- SQL Hints (see e.g. Oracle hints or SQL-Server hints) - only when required for ultimate performance tuning

This list is just containing the most important aspects. As we can not cover all these topics here, they are linked to external documentation that can help and guide you.

4.3.12. Spring-Data

If you are using the springframework and have no restrictions regarding that, we recommend to use spring-data-jpa via devon4j-starter-spring-data-jpa that brings advanced integration (esp. for QueryDSL).

Motivation

The benefits of spring-data are (for examples and explanations see next sections):

- All you need is one single repository interface for each entity. No need for a separate implementation or other code artefacts like XML descriptors, `NamedQueries` class, etc.
- You have all information together in one place (the repository interface) that actually belong together (where as in the classic approach you have the static queries in an XML file, constants to them in `NamedQueries` class and referencing usages in DAO implementation classes).
- Static queries are most simple to realize as you do not need to write any method body. This means you can develop faster.
- Support for paging is already build-in. Again for static query method the is nothing you have to do except using the paging objects in the signature.
- Still you have the freedom to write custom implementations via default methods within the repository interface (e.g. for dynamic queries).

Repository

For each entity `«Entity»Entity` an interface is created with the name `«Entity»Repository` extending `DefaultRepository`. Such repository is the analogy to a Data-Access-Object (DAO) used in the classic approach or when spring-data is not an option.

Example

The following example shows how to write such a repository:

```
public interface ExampleRepository extends DefaultRepository<ExampleEntity> {

  @Query("SELECT example FROM ExampleEntity example" //
      + " WHERE example.name = :name")
  List<ExampleEntity> findByName(@Param("name") String name);

  @Query("SELECT example FROM ExampleEntity example" //
      + " WHERE example.name = :name")
  Page<ExampleEntity> findByNamePaginated(@Param("name") String name, Pageable
pageable);

  default Page<ExampleEntity> findByCriteria(ExampleSearchCriteriaTo criteria) {
    ExampleEntity alias = newDslAlias();
    JPAQuery<ExampleEntity> query = newDslQuery(alias);
    String name = criteria.getName();
    if ((name != null) && !name.isEmpty()) {
      QueryUtil.get().whereString(query, $(alias.getName()), name, criteria
.getNameOption());
    }
    return QueryUtil.get().findPaginated(criteria.getPageable(), query, false);
  }

}
```

This `ExampleRepository` has the following features:

- CRUD support from spring-data (see JavaDoc for details).
- Support for QueryDSL integration, paging and more as well as locking via GenericRepository
- A static query method `findByName` to find all `ExampleEntity` instances from DB that have the given name. Please note the `@Param` annotation that links the method parameter with the variable inside the query (`:name`).
- The same with pagination support via `findByNamePaginated` method.
- A dynamic query method `findByCriteria` showing the QueryDSL and paging integration into spring-data provided by devon.

Further examples

You can also read the JUnit test-case DefaultRepositoryTest that is testing an example FooRepository.

Auditing

In case you need auditing, you only need to extend `DefaultRevisionedRepository` instead of `DefaultRepository`. The auditing methods can be found in GenericRevisionedRepository.

Dependency

In case you want to switch to or add spring-data support to your devon application all you need is

this maven dependency:

```
<!-- Starter for consuming REST services -->
<dependency>
  <groupId>com.devonfw.java.starters</groupId>
  <artifactId>devon4j-starter-spring-data-jpa</artifactId>
</dependency>
```

Drawbacks

Spring-data also has some drawbacks:

- Some kind of magic behind the scenes that are not so easy to understand. So in case you want to extend all your repositories without providing the implementation via a default method in a parent repository interface you need to deep-dive into spring-data. We assume that you do not need that and hope what spring-data and devon already provides out-of-the-box is already sufficient.
- The spring-data magic also includes guessing the query from the method name. This is not easy to understand and especially to debug. Our suggestion is not to use this feature at all and either provide a `@Query` annotation or an implementation via default method.

4.3.13. Data Access Object

The *Data Acccess Objects* (DAOs) are part of the persistence layer. They are responsible for a specific entity and should be named `«Entity»Dao` and `«Entity»DaoImpl`. 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 `ApplicationDao` interface that comes with the devon application template:

```
public interface MyEntityDao extends ApplicationDao<MyEntity> {
  List<MyEntity> findByCriteria(MyEntitySearchCriteria criteria);
}
```

All CRUD operations are inherited from `ApplicationDao` so you only have to declare the additional methods.

DAO Implementation

Implementing a DAO is quite simple. We create a class named `«Entity»DaoImpl` that extends `ApplicationDaoImpl` and implements your `«Entity»Dao` interface:

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

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

Again you only need to implement the additional non-CRUD 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.

Static queries for DAO Implementation

All static queries are declared in the file src\main\resources\META-INF\orm.xml:

```
<?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="find.dish.with.max.price">
    <query><![SELECT dish FROM DishEntity dish WHERE dish.price <= :maxPrice]]
></query>
  </named-query>
  ...
</hibernate-mapping>
```

When your application is started, all these static queries will be created as prepared statements. This allows better performance and also ensures that you get errors for invalid JPQL queries when you start your app rather than later when the query is used.

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

```
public class NamedQueries {
  public static final String FIND_DISH_WITH_MAX_PRICE = "find.dish.with.max.price";
}
```

Note that changing the name of the java constant (FIND_DISH_WITH_MAX_PRICE) 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:

```
public List<DishEntity> findDishByMaxPrice(BigDecimal maxPrice) {
  Query query = getEntityManager().createNamedQuery(NamedQueries
.FIND_DISH_WITH_MAX_PRICE);
  query.setParameter("maxPrice", maxPrice);
  return query.getResultList();
}
```

Via `EntityManager.createNamedQuery(String)` we create an instance of `Query` for our predefined static query. Next we use `setParameter(String, Object)` to provide a parameter (`maxPrice`) to the query. This has to be done for all parameters of the query.

Note that using the `createQuery(String)` method, which takes the entire query as string (that may already contain the parameter) is not allowed to avoid SQL injection vulnerabilities. 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.

4.3.14. JPA Performance

When using JPA the developer sometimes does not see or understand where and when statements to the database are triggered.

> Establishing expectations Developers shouldn't expect to sprinkle magic pixie dust on POJOs in hopes they will become persistent.
>
> — Dan Allen, https://epdf.tips/seam-in-action.html

So in case you do not understand what is going on under the hood of JPA, you will easily run into performance issues due to lazy loading and other effects.

N plus 1 Problem

The most prominent phenomena is call the *N+1 Problem*. We use entities from our MTS demo app as an example to explain the problem. There is a DishEntity that has a `@ManyToMany` relation to IngredientEntity. Now we assume that we want to iterate all ingredients for a dish like this:

```
DishEntity dish = dao.findDishById(dishId);
BigDecimal priceWithAllExtras = dish.getPrice();
for (IngredientEntity ingredient : dish.getExtras()) {
  priceWithAllExtras = priceWithAllExtras.add(ingredient.getPrice());
}
```

Now `dish.getExtras()` is loaded lazy. Therefore the JPA vendor will provide a list with lazy initialized instances of `IngredientEntity` that only contain the ID of that entity. Now with every call of `ingredient.getPrice()` we technically trigger an SQL query statement to load the specific `IngredientEntity` by its ID from the database. Now `findDishById` caused 1 initial query statement and for any number `N` of ingredients we are causing an additional query statement. This makes a

total of `N+1` statements. As causing statements to the database is an expensive operation with a lot of overhead (creating connection, etc.) this ends in bad performance and is therefore a problem (the N+1 Problem).

Solving N plus 1 Problem

To solve the N+1 Problem you need to change your code to only trigger a single statement instead. This can be archived in various ways. The most universal solution is to use `FETCH JOIN`s in order to pre-load the nested `N` child entities into the first level cache of the JPA vendor implementation. This will behave as if the `@ManyToMany` relation to `IngredientEntity` was having `FetchType.EAGER` but only for the that specific query and not in general. Because changing `@ManyToMany` to `FetchType.EAGER` would cause bad performance for other usecases where only the dish but not its extra ingredients are needed. For this reason all relations, including `@OneToOne` should always be `FetchType.LAZY`. Back to our example we simply replace `dao.findDishById(dishId)` with `dao.findDishWithExtrasById(dishId)` that we implement by the following JPQL query:

```
SELECT dish FROM DishEntity dish
  LEFT JOIN FETCH dish.extras
  WHERE dish.id = :dishId
```

The rest of the code does not have to be changed but now `dish.getExtras()` will get the `IngredientEntity` from the first level cache where is was fetched by the initial query above.

Please note that if you only need the sum of the prices from the extras you can also create a query using an aggregator function:

```
SELECT sum(dish.extras.price) FROM DishEntity dish
```

As you can see you need to understand the concepts in order to get good performance.

There are many advanced topics such as creating database indexes or calculating statistics for the query optimizer to get the best performance. For such advanced topics we recommend to have a database expert in your team that cares about such things. However, understanding the *N+1 Problem* and its solutions is something that every Java developer in the team needs to understand.

4.4. 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>com.devonfw.java.modules</groupId>
  <artifactId>devon4j-jpa-envers</artifactId>
</dependency>
```

Make sure that entity manager also scans the package from the devon4j-jpa[-envers] module in order to work properly.

```
@EntityScan(basePackages = { "«my.base.package»" }, basePackageClasses = {
AdvancedRevisionEntity.class })
...
public class SpringBootApp {
  ...
}
```

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
);
```

4.5. 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.

We recommend using the `@Transactional` annotation (the JEE standard `javax.transaction.Transactional` rather than `org.springframework.transaction.annotation.Transactional`). We use this annotation in the logic layer to annotate business methods that participate in transactions (what typically applies to most up to all business components):

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

In case a service operation should invoke multiple use-cases, you would end up with multiple transactions what is undesired (what if the first TX succeeds and then the second TX fails?). Therefore you would then also annotate the service operation. This is not proposed as a pattern in any case as in some rare cases you need to handle constraint-violations from the database to create a specific business exception (with specified message). In such case you have to surround the transaction with a `try {} catch` statement what is not working if that method itself is `@Transactional`.

4.5.1. Batches

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

4.6. SQL

For general guides on dealing or avoiding SQL, preventing SQL-injection, etc. you should study data-access layer.

4.6.1. Naming Conventions

Here we define naming conventions that you should follow whenever you write SQL files:

- All SQL-Keywords in UPPER CASE
- Table names in upper CamlCase (e.g. `RestaurantOrder`)
- Column names in camlCase (e.g. `drinkState`)
- Indentation should be 2 spaces as suggested by devonfw for every format.

DDL

For DDLs follow these additional guidelines:

- ID column names without underscore (e.g. `tableId`)
- Define columns and constraints inline in the statement to create the table
- Indent column types so they all start in the same text column
- Constraints should be named explicitly (to get a reasonable hint error messages) with:
  - `PK_{table}` for primary key (name optional here as PK constraint are fundamental)
  - `FK_{table}_{property}` for foreign keys (`{table}` and `{property}` are both on the source where the foreign key is defined)
  - `UC_{table}_{property}[_{propertyN}]*` for unique constraints
  - `CK_{table}_{check}` for check constraints (`{check}` describes the check, if it is defined on a single property it should start with the property).
- Databases have hard limitations for names (e.g. 30 characters). If you have to shorten names try to define common abbreviations in your project for according (business) terms. Especially do not just truncate the names at the limit.
- If possible add comments on table and columns to help DBAs understanding your schema. This is also honored by many tools (not only DBA-tools).

Here is a brief example of a DDL:

```
CREATE SEQUENCE HIBERNATE_SEQUENCE START WITH 1000000;

-- *** Table ***
CREATE TABLE Table (
  id BIGINT NOT NULL AUTO_INCREMENT,
  modificationCounter INTEGER NOT NULL,
  seatsNumber INTEGER NOT NULL,
  CONSTRAINT PK_Table PRIMARY KEY(id)
);

-- *** UserRole ***
CREATE TABLE UserRole (
  id BIGINT NOT NULL AUTO_INCREMENT,
  modificationCounter INTEGER NOT NULL,
  name VARCHAR (255),
  active BOOLEAN,
  CONSTRAINT PK_UserRole PRIMARY KEY(id)
-- *** User ***
CREATE TABLE User (
  id BIGINT NOT NULL AUTO_INCREMENT,
  modificationCounter INTEGER NOT NULL,
  username VARCHAR (255) NULL,
  password VARCHAR (255) NULL,
  email VARCHAR (120) NULL,
  idRole BIGINT NOT NULL,
  CONSTRAINT PK_User PRIMARY KEY(id),
  CONSTRAINT PK_User_idRole FOREIGN KEY(idRole) REFERENCES UserRole(id) NOCHECK
);
COMMENT ON TABLE User is 'The users of the restaurant site';
...
```

Data

For insert, update, delete, etc. of data SQL scripts should additionally follow these guidelines:

- Inserts always with the same order of columns in blocks for each table.
- Insert column values always starting with id, modificationCounter, [dtype,] ...
- List columns with fixed length values (boolean, number, enums, etc.) before columns with free text to support alignment of multiple insert statements
- Pro Tip: Get familiar with column mode of `notepad++` when editing large blocks of similar insert statements.

```
INSERT INTO UserRole(id, modificationCounter, name, active) VALUES (0, 1, 'Customer',
true);
INSERT INTO UserRole(id, modificationCounter, name, active) VALUES (1, 1, 'Waiter',
true);
INSERT INTO User(id, modificationCounter, username, password, email, idRole) VALUES
(0, 1, 'user0', 'password', 'user0@mail.com', 0);
INSERT INTO User(id, modificationCounter, username, password, email, idRole) VALUES
(1, 1, 'waiter', 'waiter', 'waiter@mail.com', 1);

INSERT INTO Table(id, modificationCounter, seatsNumber) VALUES (0, 1, 4);
INSERT INTO Table(id, modificationCounter, seatsNumber) VALUES (1, 1, 4);
INSERT INTO Table(id, modificationCounter, seatsNumber) VALUES (2, 1, 4);
INSERT INTO Table(id, modificationCounter, seatsNumber) VALUES (3, 1, 4);
INSERT INTO Table(id, modificationCounter, seatsNumber) VALUES (4, 1, 6);
INSERT INTO Table(id, modificationCounter, seatsNumber) VALUES (5, 1, 6);
INSERT INTO Table(id, modificationCounter, seatsNumber) VALUES (6, 1, 6);
INSERT INTO Table(id, modificationCounter, seatsNumber) VALUES (7, 1, 8);
INSERT INTO Table(id, modificationCounter, seatsNumber) VALUES (8, 1, 8);
...
```

See also Database Migrations.

4.7. Database Migration

For database migrations we use Flyway. As illustrated here database migrations have three advantages:

1. Recreate a database from scratch
2. Make it clear at all times what state a database is in
3. Migrate in a deterministic way from your current version of the database to a newer one

Flyway can be used standalone or can be integrated via its API to make sure the database migration takes place on startup.

4.7.1. Organizational Advice

A few considerations with respect to project organization will help to implement maintainable Flyway migrations.

At first, testing and production environments must be clearly and consistently distinguished. Use the following directory structure to achieve this distinction:

```
src/main/resources/db
src/test/resources/db
```

Although this structure introduces redundancies, the benefit overweights this disadvantage. An even more fine-grained production directory structure which contains one subfolder per release should be implemented:

```
src/main/resources/db/migration/releases/X.Y/x.sql
```

Emphasizing that migration scripts below the current version must never be changed will aid the second advantage of migrations: it will always be clearly reproducible in which state the database currently is. Here, it is important to mention that, if test data is required, it must be managed separately from the migration data in the following directory:

```
src/test/resources/db/migration/
```

The `migration` directory is added to aid easy usage of Flyway defaults. Of course, test data should also be managed per release as like production data.

With regard to content, separation of concerns (SoC) is an important goal. SoC can be achieved by distinguishing and writing multiple scripts with respect to business components/use cases (or database tables in case of large volumes of master data [1: "Stammdaten" in German.]. Comprehensible file names aid this separation.

It is important to have clear responsibilities regarding the database, the persistence layer (JPA), and

migrations. Therefore a dedicated database expert should be in charge of any migrations performed or she should at least be informed before any change to any of the mentioned parts is applied.

4.7.2. Technical Configuration

Database migrations can be SQL based or Java based.

To enable auto migration on startup (not recommended for productive environment) set the following property in the `application.properties` file for an environment.

```
flyway.enabled=true
flyway.clean-on-validation-error=false
```

For development environment it is helpful to set both properties to `true` in order to simplify development. For regular environments `flyway.clean-on-validation-error` should be `false`.

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

```
spring.jpa.hibernate.ddl-auto=validate
```

The setting must be communicated to and coordinated with the customer and their needs. In acceptance testing the same configuration as for the production environment should be enabled.

Since migration scripts will also be versioned the end-of-line (EOL) style must be fixated according to this issue. This is however solved in flyway 4.0+ and the latest devonfw release. Also, the version numbers of migration scripts should not consist of simple ascending integer numbers like V0001..., *V0002*..., ... This naming may lead to problems when merging branches. Instead the usage of timestamps as version numbers will help to avoid such problems.

4.7.3. Naming Conventions

Database migrations should follow this naming convention: V<version>__<description> (e.g.: V12345__Add_new_table.sql).

It is also possible to use Flyway for test data. To do so place your test data migrations in src/main/resources/db/testdata/ and set property

```
flyway.locations=classpath:db/migration/releases,classpath:db/migration/testdata
```

Then Flyway scans the additional location for migrations and applies all in the order specified by their version. If migrations V0001__... and V0002__... exist and a test data migration should be applied in between you can name it V0001_1__....

4.7.4. Database Integration

To integrate devon4j with various databases namely Microsoft SQL Server 2008, PostGres 9.5.4 , Oracle 11g and MariaDB 10.0.27 , refer the following sections

Database Integration with Microsoft SQL Server

To integrate devon4j with Microsoft SQL Server 2008 , please refer instructions here

Database Integration with PostGres

To integrate devon4j with PostGres 9.5.4, please refer instructions here

Database Integration with Oracle

To integrate devon4j with Oracle 11g, please refer instructions here

Database Integration with MariaDB

To integrate devon4j with MariaDB 10.0.27, please refer instructions here

4.8. Oracle RDBMS

This section contains hints for those who use Oracle RDBMS. If you use a different persistence technology you can simply ignore it. Besides general hints about the driver there are tips for more tight integration with other Oracle features or products. However, if you work for a project where Oracle RDBMS is settled and not going to be replaced (you are in a vendor lock-in anyway), you might want to use even more from Oracle technology to take advantage from a closer integration.

4.8.1. Driver

The oracle JDBC driver is not available in maven central. Depending on the Oracle DB version and the Java version, you can use either the 11g/ojdbc6, 12c/ojdbc7, or 12c/ojdbc8 version of the driver. Oracle JDBC drivers usually are backward and forward compatible so you should be able to use the 12c/ojdbc8 driver with an 11g DB etc. As a rule of thumb, use the 12c/ojdbc8 driver unless you must use Java7. All JDBC drivers can be downloaded without registration: 11g/ojdbc6, 12c/ojdbc7, and 12c/ojdbc8. Your project should use a maven repository server such as nexus or artifactory. Your dependency for the oracle driver should look as follows (use artifactId "ojdbc6" or "ojdbc7" for the older drivers):

```
<dependency>
  <groupId>com.oracle</groupId>
  <artifactId>ojdbc8</artifactId>
  <version>${oracle.driver.version}</version>
</dependency>
```

oracle.driver.version being 11.2.0.4 for 11g/ojdbc6, or 12.1.0.1 for 12c/ojdbc7, or 12.2.0.1 for 12c/ojdbc8 or newer

4.8.2. Pooling

In order to boost performance JDBC connections should be pooled and reused. If you are using Oracle RDBMS and do not plan to change that you can use the Oracle specific connection pool "Universal Connection Pool (UCP)" that is perfectly integrated with the Oracle driver. According to the documentation, UCP can even be used to manage third party data sources. The 11g version of UCP can be downloaded without registration here, the 12c version of UCP is available at the same download locations as the 12c JDBC driver (see above). As a rule of thumb, use the version that is the same as the JDBC driver version. Again, you have to upload the artefact manually to your maven repository. The dependency should look like this:

```
<dependency>
  <groupId>com.oracle</groupId>
  <artifactId>ucp</artifactId>
  <version>${oracle.ucp.version}</version>
</dependency>
```

with oracle.ucp.version being 11.2.0.4 or 12.2.0.1 or newer.

Configuration is done via application.properties like this (example):

```
#Oracle UCP
# Datasource for accessing the database
spring.datasource.url=jdbc:oracle:thin:@192.168.58.2:1521:xe
spring.jpa.database-platform=org.hibernate.dialect.Oracle12cDialect
spring.datasource.user=MyUser
spring.datasource.password=ThisIsMyPassword
spring.datasource.driver-class-name=oracle.jdbc.OracleDriver
spring.datasource.schema=MySchema

spring.datasource.type=oracle.ucp.jdbc.PoolDataSourceImpl
spring.datasource.factory=oracle.ucp.jdbc.PoolDataSourceFactory
spring.datasource.factory-method=getPoolDataSource
spring.datasource.connectionFactoryClassName=oracle.jdbc.pool.OracleDataSource
spring.datasource.validateConnectionOnBorrow=true
spring.datasource.connectionPoolName=MyPool
spring.datasource.jmx-enabled=true

# Optional: Set the log level to INTERNAL_ERROR, SEVERE, WARNING, INFO, CONFIG, FINE,
TRACE_10, FINER, TRACE_20, TRACE_30, or FINEST
# logging.level.oracle.ucp=INTERNAL_ERROR
# Optional: activate tracing
# logging.level.oracle.ucp.jdbc.oracle.OracleUniversalPooledConnection=TRACE

#Optional: Configures pool size manually
#spring.datasource.minPoolSize=10
#spring.datasource.maxPoolSize=40
#spring.datasource.initialPoolSize=20
```

Resources: FAQ, developer's guide, Java API Reference. For an in-depth discussion on how to use JDBC and UCP, see the Oracle documentation Connection Management Strategies for Java Applications using JDBC and UCP.

Note: there is a bug in UCP 12.1.0.2 that results in the creation of thousands of java.lang.Timer threads over hours or days of system uptime (see article on stackoverflow). Also, Oracle has a strange bug fixing / patching policy: instead of producing a fixed version 12.1.0.3 or 12.1.0.2.x, Oracle publishes collections of *.class files that must be manually patched into the ucp.jar! Therefore, use the newest versions only.

4.8.3. Messaging

In case you want to do messaging based on JMS you might consider the Oracle JMS also called Oracle Streams Advanced Queuing, or Oracle Advanced Queuing, or OAQ or AQ for short. OAQ is a JMS provider based on the Oracle RDBMS and included in the DB product for no extra fee. OAQ has some features that exceede the JMS standard like a retention time (i.e. a built-in backup mechanism that allows to make messages "unread" within a configurable period of time so that these messages do not have to be resent by the sending application). Also, OAQ messages are stored in relational tables so they can easily be observed by a test driver in a system test scenario. Capgemini has used

the Spring Data JDBC Extension in order to process OAQ messages within **the same technical transaction** as the resulting Oracle RDBMS data changes **without** using 2PC and an XA-compliant transaction manager - which is not available out of the box in Tomcat. This is possible only due to the fact that OAQ queues and RDBMS tables actually reside in the same database. However, this is higher magic and should only be tried if high transaction rates must be achieved by avoiding 2PC.

4.8.4. General Notes on the use of Oracle products

Oracle sells commercial products and receives licence fees for them. This includes access to a support organization. Therefore, at an early stage of your project, prepare for contacting oracle support in case of technical problems. You will need the Oracle support ID **of your customer** [i.e. the legal entity who pays the licence fee and runs the RDBMS] and your customer must grant you permission to use it in a service request - it is not legal to use a your own support ID in a customer-related project. Your customer pays for that service anyway, so use it in case of a problem!

Software components like the JDBC driver or the UCP may be available without a registration or fee but they are protected by the Oracle Technology Network (OTN) License Agreement. The most important aspect of this licence agreement is the fact that an IT service provider is not allowed to simply download the Oracle software component, bundle it in a software artefact and deliver it to the customer. Instead, the Oracle software component must be (from a legal point of view) provided by the owner of the Oracle DB licence (i.e. your customer). This can be achieved in two ways: Advise your customer to install the Oracle software component in the application server as a library that can be used by your custom built system. Or, in cases where this is not feasible, e.g. in a OpenShift environment where the IT service provider delivers complete Docker images, you must advise your customer to (legally, i.e. documented in a written form) provide the Oracle software component to you, i.e. you don't download the software component from the Oracle site but receive it from your customer.

4.9. 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.

4.9.1. Usage

Maven Integration

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

```
<dependency>
  <groupId>com.devonfw.java</groupId>
  <artifactId>devon4j-logging</artifactId>
  <version>2.4.0</version>
</dependency>
```

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. devon4j 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 devon4j-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/
```

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.

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 5. Log-levels

Log-level	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.

4.9.2. Operations

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 6. 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.

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»] [C: «NDC»][T: «thread»][L: «logger»]-[M:
«message»]
```

- **D**: Date (Timestamp in ISO8601 format e.g. 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...]
```

4.9.3. Security

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

4.9.4. Correlation ID

In order to correlate separate HTTP requests to services belonging to the same user / session, we provide a servlet filter called `DiagnosticContextFilter`. This filter takes a provided correlation ID from the HTTP header `X-Correlation-Id`. If none was found, it will generate a new correlation id as `UUID`. This correlation ID is added as MDC to the logger. Therefore, it will then be included to any log message of the current request (thread). Further concepts such as service invocations will pass this correlation ID to subsequent calls in the application landscape. Hence you can find all log messages related to an initial request simply via the correlation ID even in highly distributed systems.

4.9.5. Monitoring

In highly distributed systems (from clustering up to microservices) it might get tedious to search for problems and details in log files. Therefore, we recommend to setup a central log and analysis server for your application landscape. Then you feed the logs from all your applications (using logstash) into that central server that adds them to a search index to allow fast searches (using elasticsearch). This should be completed with a UI that allows dashboards and reports based on data aggregated from all the logs. This is addressed by ELK or Graylog.

4.10. 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 guide will not only automatically prevent you from any harm, but will provide you hints and best practices already used in different software products.

An important aspect of security is proper authentication and authorization as described in access-control. In the following we discuss about potential vulnerabilities and protection to prevent them.

4.10.1. 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 devonfw:

Table 7. Security threats and protection-mechanisms

Threat	Protection	Link to details
A1 Injection	validate input, escape output, use proper frameworks	SQL Injection
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 devonfw application template and guides to avoid	application template and sensitive configuration
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` as default to enforce	Method authorization

Threat	Protection	Link to details
A8 CSRF	secure mutable service operations with an explicit CSRF security token sent in HTTP header and verified on the server	service-layer security
A9 Using Components with Known Vulnerabilities	subscribe to security newsletters, recheck products and their versions continuously, use devonfw dependency management	CVE newsletter and dependency check
A10 Unvalidated Redirects and Forwards	Avoid using redirects and forwards, in case you need them do a security audit on the solution.	devonfw 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

4.10.2. Tools

Dependency Check

To address A9 Using Components with Known Vulnerabilities we integrated OWASP dependency check into the devonfw maven build. If you build an devonfw application (sample or any app created from our app-template) you can activate dependency check with the `security` profile:

```
mvn clean install -P security
```

This does not run by default as it causes a huge overhead for the build performance. However , consider to build this in your CI at least nightly. After the dependency check is performed , you will find the results in `target/dependency-check-report.html` of each module. The report will also always be generated when the site is build (`mvn site`).

Penetration Testing

For penetration testing (testing for vulnerabilities) of your web application, we recommend the following tools:

- ZAP (OWASP Zed Attack Proxy Project)
- sqlmap (or HQLmap)
- nmap
- See the marvellous presentation Toolbox of a security professional from Christian Schneider.

4.11. Access-Control

Access-Control is a central and important aspect of Security. It consists of two major aspects:

- (Who tries to access?)
- (Is the one accessing allowed to do what he wants to do?)

4.11.1. 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.

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). Otherwise you should consider establishing such a system e.g. using keycloak.

We use spring-security as a framework for authentication purposes.

Therefore you need to provide an implementation of WebSecurityConfigurerAdapter:

```
@Configuration
@EnableWebSecurity
public class MyWebSecurityConfig extends WebSecurityConfigurerAdapter {

  @Inject
  private UserDetailsService userDetailsService;
  ...
  public void configure(HttpSecurity http) throws Exception {
    http.userDetailsService(this.userDetailsService)
        .authorizeRequests().antMatchers("/public/**").permitAll()
        .anyRequest().authenticated().and()
        ...
  }
}
```

As you can see spring-security offers a fluent API for easy configuration. You can simply add invocations like `formLogin().loginPage("/public/login")` or `httpBasic().realmName("MyApp")`. Also CSRF protection can be configured by invoking `csrf()`. For further details see spring Java-config for HTTP security.

Further, you need to provide an implementation of the UserDetailsService interface. A good starting point comes with our application template.

Preserve original request anchors after form login redirect

Spring Security will automatically redirect any unauthorized access to the defined login-page. After successful 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 `devon4j-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

You can easily configure this as code in your `WebSecurityConfig` as following:

```
RetainAnchorFilter filter = new RetainAnchorFilter();
filter.setStoreUrlPattern("http://[^/]+/[^/]+/login.*");
filter.setRestoreUrlPattern("http://[^/]+/[^/]+/.*");
filter.setCookieName("TARGETANCHOR");
http.addFilterBefore(filter, UsernamePasswordAuthenticationFilter.class);
```

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.

4.11.2. Authorization

Definition:

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

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 8. Security terms related to authorization

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

Suggestions on the access model

For the access model we give the following 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 avoid negative permissions, that is a user has no permission by default and only those granted to him explicitly give him additional permission for specific things. Permissions granted can not be reduced by other permissions.
- 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.

Naming conventions

As stated above each Access Control is uniquely identified by a human readable string. This string should follow the naming convention:

```
«app-id».«local-name»
```

For Access Control Permissions the `«local-name»` again follows the convention:

```
«verb»«object»
```

The segments are defined by the following table:

Table 9. Segments of Access Control Permission ID

Segment	Description	Example
«app-id»	Is a unqiue technical but human readable string of the application (or microservice). It shall not contain special characters and especially no dot or whitespace. We recommend to use `lower-train-case-ascii-syntax`. The identity and access management should be organized on enterprise level rather than application level. Therefore permissions of different apps might easily clash (e.g. two apps might both define a group `ReadMasterData` but some user shall get this group for only one of these two apps). Using the `«app-id».` prefix is a simple but powerful namespacing concept that allows you to scale and grow. You may also reserve specific «app-id»s for cross-cutting concerns that do not actually reflect a single app e.g to grant access to a geographic region.	`shop`
«verb»	The action that is to be performed on «object». We use `Find` for searching and reading data. `Save` shall be used both for create and update. Only if you really have demands to separate these two you may use `Create` in addition to `Save`. Finally, `Delete` is used for deletions. For non CRUD actions you are free to use additional verbs such as `Approve` or `Reject`.	`Find`
«object»	The affected object or entity. Shall be named according to your data-model	`Product`

So as an example `shop.FindProduct` will reflect the permission to search and retrieve a `Product` in the `shop` application. The group `shop.ReadMasterData` may combine all permissions to read master-data from the `shop`. However, also a group `shop.Admin` may exist for the `Admin` role of the `shop` application. Here the `«local-name»` is `Admin` that does not follow the `«verb»«object»` schema.

devon4j-security

The devonfw provides a ready to use module `devon4j-security` that is based on spring-security and makes your life a lot easier.

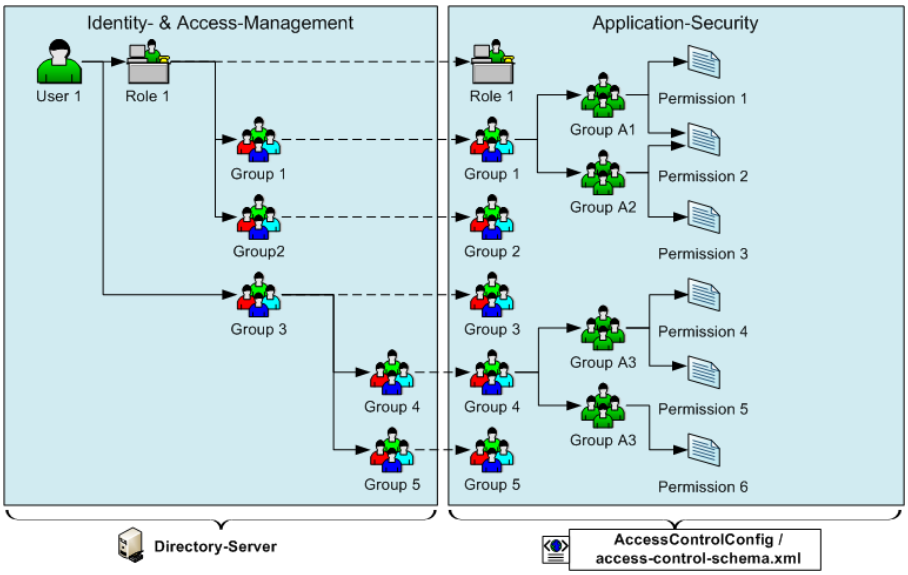


Figure 2. devon4j Security Model

The diagram shows the model of `devon4j-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 defines a hierarchy of *secondary access control objects* (groups and permissions). This is done by configuration owned by the application (see following section). The "API" is defined by the IDs public access control objects (groups) that may be referenced from the *Indentity- and Access-Management*.

Access Control Config

In your application simply extend `AccessControlConfig` to configure your access control objects as code and reference it from your use-cases. An example config may look like this:

```
@Named
public class MyAccessControlConfig extends AccessControlConfig {

  public static final String APP_ID = "MyApp";

  private static final String PREFIX = APP_ID + ".";

  public static final String PERMISSION_FIND_OFFER = PREFIX + "FindOffer";

  public static final String PERMISSION_SAVE_OFFER = PREFIX + "SaveOffer";

  public static final String PERMISSION_DELETE_OFFER = PREFIX + "DeleteOffer";

  public static final String PERMISSION_FIND_PRODUCT = PREFIX + "FindProduct";

  public static final String PERMISSION_SAVE_PRODUCT = PREFIX + "SaveProduct";

  public static final String PERMISSION_DELETE_PRODUCT = PREFIX + "DeleteProduct";

  public static final String GROUP_READ_MASTER_DATA = PREFIX + "ReadMasterData";

  public static final String GROUP_MANAGER = PREFIX + "Manager";

  public static final String GROUP_ADMIN = PREFIX + "Admin";

  public MyAccessControlConfig() {

    super();
    AccessControlGroup readMasterData = group(GROUP_READ_MASTER_DATA,
PERMISSION_FIND_OFFER, PERMISSION_FIND_PRODUCT);
    AccessControlGroup manager = group(GROUP_MANAGER, readMasterData,
PERMISSION_SAVE_OFFER, PERMISSION_SAVE_PRODUCT);
    AccessControlGroup admin = group(GROUP_ADMIN, manager, PERMISSION_DELETE_OFFER,
PERMISSION_DELETE_PRODUCT);
  }
}
```

In your use-case you can now reference a permission like this:

```
@Named
public class UcSafeOfferImpl extends ApplicationUc implements UcSafeOffer {

  @Override
  @RolesAllowed(MyAccessControlConfig.PERMISSION_SAVE_OFFER)
  public OfferEto save(OfferEto offer) { ... }
  ...
}
```

Access Control Schema (deprecated)

The devon4j-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 (see example from sample app). The general terms discussed above can be mapped to the implementation as follows:

Table 10. General security terms related to devon4j access control schema

Term	devon4j-security implementation	Comment
Permission	AccessControlPermission	
Group	AccessControlGroup	When considering different levels of groups of different meanings, declare type attribute, e.g. as "group".
Role	AccessControlGroup	With type="role".
Access Control	AccessControl	Super type that represents a tree of AccessControlGroups and AccessControlPermissions. If a principal "has" a AccessControl he also "has" all AccessControls with according permissions in the spanned sub-tree.

Example access-control-schema.xml

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

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

This example access-control-schema.xml declares

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

`OfferManagement_GetOffer`

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

The devon4j-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 devon4j-security provided access control schema, you can simply inherit your own implementation from the devon4j-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 devon4j-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=
"com.devonfw.module.security.common.base.accesscontrol.AccessControlProviderImpl"/>
<bean id="AccessControlSchemaProvider" class=
"com.devonfw.module.security.common.base.accesscontrol.AccessControlSchemaProviderImpl
"/>
```

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 devon4j 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 OrdermanagementImpl extends AbstractComponentFacade implements
Ordermanagement {

  @RolesAllowed(Roles.WAITER)
  public PaginatedListTo<OrderCto> findOrdersByPost(OrderSearchCriteriaTo criteria) {

    return findOrderCtos(criteria);
  }
}
```

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

Check Data-based Permissions

See data permissions

4.12. Data-permissions

In some projects there are demands for permissions and authorization that is dependent on the processed data. E.g. a user may only be allowed to read or write data for a specific region. This is adding some additional complexity to your authorization. If you can avoid this it is always best to keep things simple. However, in various cases this is a requirement. Therefore the following sections give you guidance and patterns how to solve this properly.

4.12.1. Structuring your data

For all your business objects (entities) that have to be secured regarding to data permissions we recommend that you create a separate interface that provides access to the relevant data required to decide about the permission. Here is a simple example:

```
public interface SecurityDataPermissionCountry {

  /**
   * @return the 2-letter ISO code of the country this object is associated with.
Users need
   *         a data-permission for this country in order to read and write this
object.
   */
  String getCountry();
}
```

Now related business objects (entities) can implement this interface. Often such data-permissions have to be applied to an entire object-hierarchy. For security reasons we recommend that also all child-objects implement this interface. For performance reasons we recommend that the child-objects redundantly store the data-permission properties (such as `country` in the example above) and this gets simply propagated from the parent, when a child object is created.

4.12.2. Permissions for processing data

When saving or processing objects with a data-permission, we recommend to provide dedicated methods to verify the permission in an abstract base-class such as `AbstractUc` and simply call this explicitly from your business code. This makes it easy to understand and debug the code. Here is a simple example:

```
protected void verifyPermission(SecurityDataPermissionCountry entity) throws
AccessDeniedException;
```

Beware of AOP

For simple but cross-cutting data-permissions you may also use AOP. This leads to programming aspects that reflectively scan method arguments and magically decide what to do. Be aware that this quickly gets tricky:

- What if multiple of your method arguments have data-permissions (e.g. implement `SecurityDataPermission*`)?
- What if the object to authorize is only provided as reference (e.g. `Long` or `IdRef`) and only loaded and processed inside the implementation where the AOP aspect does not apply?
- How to express advanced data-permissions in annotations?

What we have learned is that annotations like `@PreAuthorize` from `spring-security` easily lead to the "programming in string literals" anti-pattern. We strongly discourage to use this anti-pattern. In such case writing your own `verifyPermission` methods that you manually call in the right places of your business-logic is much better to understand, debug and maintain.

4.12.3. Permissions for reading data

When it comes to restrictions on the data to read it becomes even more tricky. In the context of a user only entities shall be loaded from the database he is permitted to read. This is simple for loading a single entity (e.g. by its ID) as you can load it and then if not permitted throw an exception to secure your code. But what if the user is performing a search query to find many entities? For performance reasons we should only find data the user is permitted to read and filter all the rest already via the database query. But what if this is not a requirement for a single query but needs to be applied cross-cutting to tons of queries? Therefore we have the following pattern that solves your problem:

For each data-permission attribute (or set of such) we create an abstract base entity:

```
@MappedSuperclass
@EntityListeners(PermissionCheckListener.class)
@FilterDef(name = "country", parameters = {@ParamDef(name = "countries", type =
"string")})
@Filter(name = "country", condition = "country in (:countries)")
public abstract class SecurityDataPermissionCountryEntity extends
ApplicationPersistenceEntity
    implements SecurityDataPermissionCountry {

  private String country;

  @Override
  public String getCountry() {
    return this.country;
  }

  public void setCountry(String country) {
    this.country = country;
  }
}
```

There are some special hibernate annotations `@EntityListeners`, `@FilterDef`, and `@Filter` used here allowing to apply a filter on the `country` for any (non-native) query performed by hibernate. The entity listener may look like this:

```
public class PermissionCheckListener {

  @PostLoad
  public void read(SecurityDataPermissionCountryEntity entity) {
    PermissionChecker.getInstance().requireReadPermission(entity);
  }

  @PrePersist
  @PreUpdate
  public void write(SecurityDataPermissionCountryEntity entity) {
    PermissionChecker.getInstance().requireWritePermission(entity);
  }
}
```

This will ensure that hibernate implicitly will call these checks for every such entity when it is read from or written to the database. Further to avoid reading entities from the database the user is not permitted to (and ending up with exceptions), we create an AOP aspect that automatically activates the above declared hibernate filter:

```
@Named
public class PermissionCheckerAdvice implements MethodBeforeAdvice {

  @Inject
  private PermissionChecker permissionChecker;

  @PersistenceContext
  private EntityManager entityManager;

  @Override
  public void before(Method method, Object[] args, Object target) {

    Collection<String> permittedCountries = this.permissionChecker
.getPermittedCountriesForReading();
    if (permittedCountries != null) { // null is returned for admins that may access
all countries
      if (permittedCountries.isEmpty()) {
        throw new AccessDeniedException("Not permitted for any country!");
      }
      Session session = this.entityManager.unwrap(Session.class);
      session.enableFilter("country").setParameterList("countries",
permittedCountries.toArray());
    }
  }
}
```

Finally to apply this aspect to all Repositories (can easily be changed to DAOs) implement the following advisor:

```
```

```
@Named
public class PermissionCheckerAdvisor implements PointcutAdvisor, Pointcut,
ClassFilter, MethodMatcher {

  @Inject
  private PermissionCheckerAdvice advice;

  @Override
  public Advice getAdvice() {
    return this.advice;
  }

  @Override
  public boolean isPerInstance() {
    return false;
  }

  @Override
  public Pointcut getPointcut() {
    return this;
  }

  @Override
  public ClassFilter getClassFilter() {
    return this;
  }

  @Override
  public MethodMatcher getMethodMatcher() {
    return this;
  }

  @Override
  public boolean matches(Method method, Class<?> targetClass) {
    return true; // apply to all methods
  }

  @Override
  public boolean isRuntime() {
    return false;
  }

  @Override
  public boolean matches(Method method, Class<?> targetClass, Object... args) {
    throw new IllegalStateException("isRuntime()==false");
  }

  @Override
  public boolean matches(Class<?> clazz) {
    // when using DAOs simply change to some class like ApplicationDao
    return DefaultRepository.class.isAssignableFrom(clazz);
```

```
  }
}
```

4.12.4. Managing and granting the data-permissions

Following our authorization guide we can simply create a permission for each country. We might simply reserve a prefix (as virtual `«app-id»`) for each data-permission to allow granting data-permissions to end-users across all applications of the IT landscape. In our example we could create access controls `country.DE`, `country.US`, `country.ES`, etc. and assign those to the users. The method `permissionChecker.getPermittedCountriesForReading()` would then scan for these access controls and only return the 2-letter country code from it.

Before you make your decisions how to design your access controls please clarify the following questions:

- Do you need to separate data-permissions independent of the functional permissions? E.g. may it be required to express that a user can read data from the countries `ES` and `PL` but is only permitted to modify data from `PL`? In such case a single assignment of "country-permissions" to users is insufficient.
- Do you want to grant data-permissions individually for each application (higher flexibility and complexity) or for the entire application landscape (simplicity, better maintenance for administrators)? In case of the first approach you would rather have access controls like `app1.country.GB` and `app2.country.GB`.
- Do your data-permissions depend on objects that can be created dynamically inside your application?
- If you want to grant data-permissions on other business objects (entities), how do you want to reference them (primary keys, business keys, etc.)? What reference is most stable? Which is most readable?

4.13. 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.

4.13.1. 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.

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>
```

- Add the @Validated annotation to the implementation (spring bean) to be validated. The standard use case is to annotate the logic layer implementation, i.e. the use case implementation or component facade in case of simple logic layer pattern. Thus, the validation will be executed for service requests as well as batch processing. 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). .**BookingmanagementRestServiceImpl.java**

```
@Validated
public class BookingmanagementRestServiceImpl implements BookingmanagementRestService
{
  ...
  public BookingEto saveBooking(@Valid BookingCto booking) {
  ...
```

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

```
  @Valid
  private BookingEto booking;
```

BookingEto.java

```
  @NotNull
  @Future
  private Timestamp bookingDate;
```

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.

Bean Validation in Wildfly >v8: Wildfly v8 is the first version of Wildfly implementing the JEE7 specification. It comes with bean validation based on hibernate-validator out of the box. In case someone is running Spring in Wildfly for whatever reasons, the spring based annotation @Validated would duplicate bean validation at runtime and thus should be omitted.

GUI-Integration

TODO

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() ...
```

4.13.2. 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: **BookingmanagementImpl.java**

```
  private void sendConfirmationEmails(BookingEntity booking) {

    if (!booking.getInvitedGuests().isEmpty()) {
      for (InvitedGuestEntity guest : booking.getInvitedGuests()) {
        sendInviteEmailToGuest(guest, booking);
      }
    }

    sendConfirmationEmailToHost(booking);
  }
```

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

4.14. 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.

4.14.1. 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).

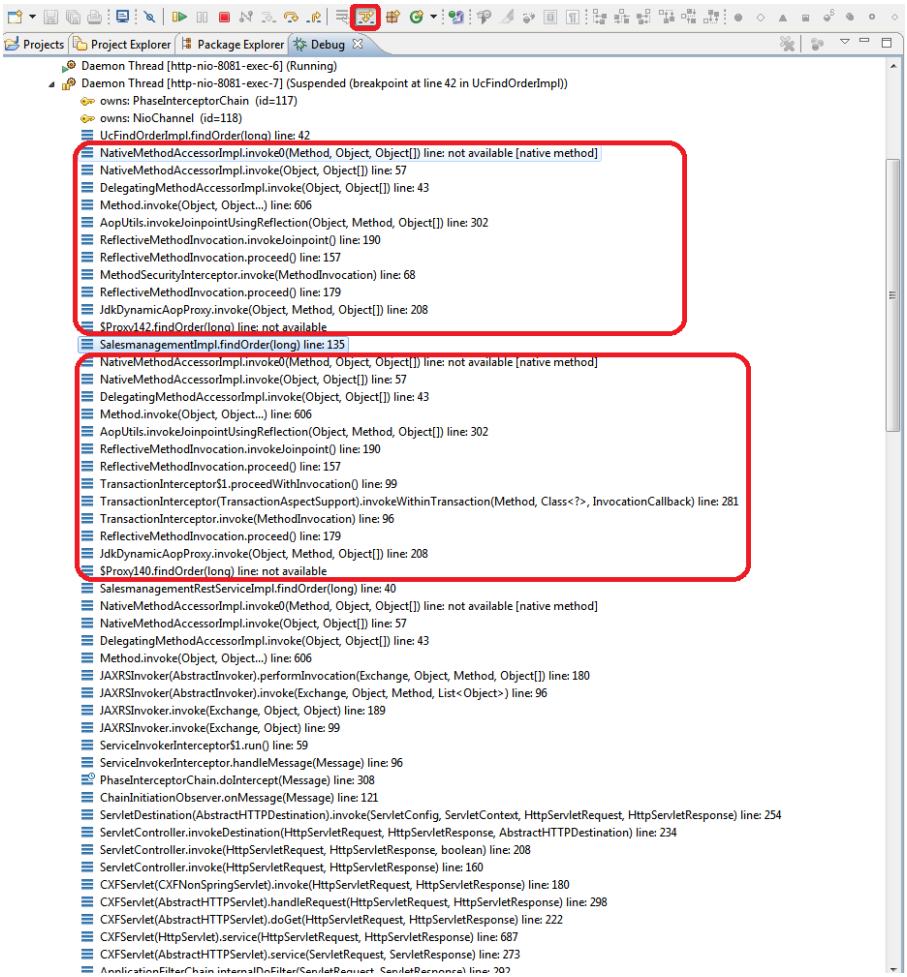
4.14.2. AOP Usage

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

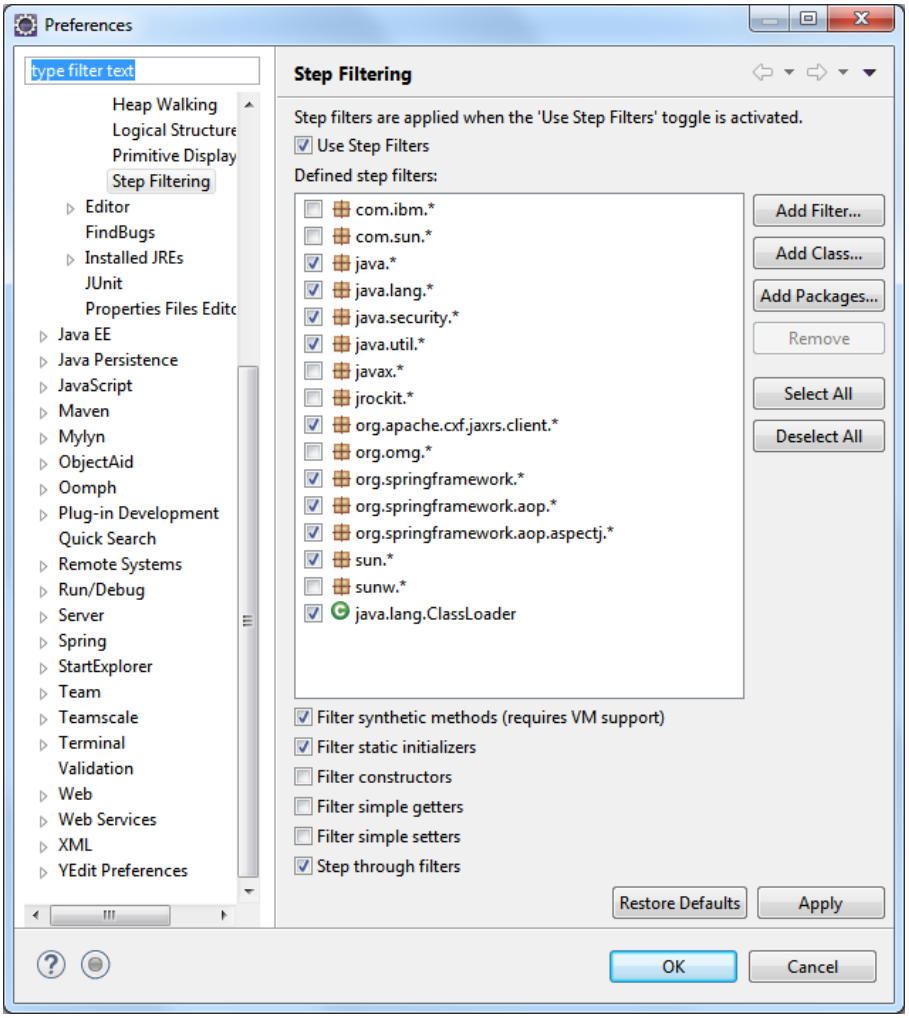
- Transaction-Handling
- Authorization
- Validation
- Trace-Logging (for testing and debugging)
- Exception facades for services but only if no other solution is possible (use alternatives such as JAX-RS provider instead).

4.14.3. AOP Debugging

When using AOP with dynamic proxies the debugging of your code can get nasty. As you can see by the red boxes in the call stack in the debugger there is a lot of magic happening while you often just want to step directly into the implementation skipping all the AOP clutter. When using Eclipse this can easily be archived by enabling *step filters*. Therefore you have to enable the feature in the Eclipse tool bar (highlighted in read).



In order to properly make this work you need to ensure that the step filters are properly configured:



Ensure you have at least the following step-filters configured and active:

```
ch.qos.logback.*
com.devonfw.module.security.*
java.lang.reflect.*
java.security.*
javax.persistence.*
org.apache.commons.logging.*
org.apache.cxf.jaxrs.client.*
org.apache.tomcat.*
org.h2.*
org.springframework.*
```

4.15. Exception Handling

4.15.1. 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.

4.15.2. Exception Example

com.devonfw.application.mtsj.ordermanagement.common.api.exception Here is an exception class from our sample application:

```
public class IllegalEntityStateException extends ApplicationBusinessException {

  private static final long serialVersionUID = 1L;

  public IllegalEntityStateException(Object entity, Object state) {

    this((Throwable) null, entity, state);
  }


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

    this(null, entity, currentState, newState);
  }

  public IllegalEntityStateException(Throwable cause, Object entity, Object state) {

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

  public IllegalEntityStateException(Throwable cause, Object entity, Object
currentState, Object newState) {

    super(cause, createBundle(NlsBundleApplicationRoot.class)
.errorIllegalEntityStateChange(entity, currentState,
        newState));
  }

}
```

The message templates are defined in the interface NlsBundleRestaurantRoot as following: com.devonfw.application.mtsj.ordermanagement.common.api

```
public interface NlsBundleApplicationRoot 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);


  @NlsBundleMessage("The property {property} of object {object} can not be changed!")
  NlsMessage errorIllegalPropertyChange(@Named("object") Object object, @Named(
"property") Object property);

  @NlsBundleMessage("There is currently no user logged in")
  NlsMessage errorNoActiveUser();
```

4.15.3. 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 devon4j 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.

4.16. Internationalization

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

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

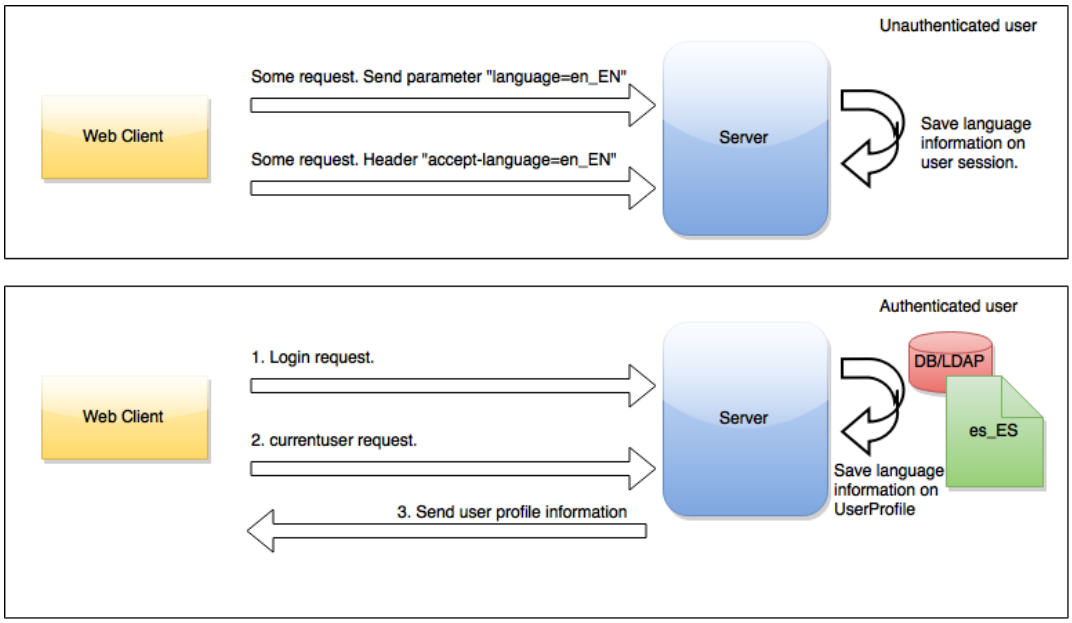
- Bind locale information to the user.
- Get the messages in the current user locale.

4.16.1. 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: devonfw 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 devonfw sample we get the locale information from database.

This image shows the entire process:



4.16.2. Getting internationalizated messages

devonfw 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

devonfw sample user roles:

ApplicationMessages_en_US.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");
```

4.17. 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 devonfw 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.

4.17.1. JAXB

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

JAXB and Inheritance

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

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

4.18. 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.

4.18.1. Configure JSON Mapping

In order to avoid polluting business objects with proprietary Jackson annotations (e.g. `@JsonTypeInfo`, `@JsonSubTypes`, `@JsonProperty`) we propose to create a separate configuration class. Every devonfw application (sample or any app created from our app-template) therefore has a class called `ApplicationObjectMapperFactory` that extends ObjectMapperFactory from the devon4j-rest module. It looks like this:

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

  public RestaurantObjectMapperFactory() {
    super();
    // JSON configuration code goes here
  }
}
```

4.18.2. 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:

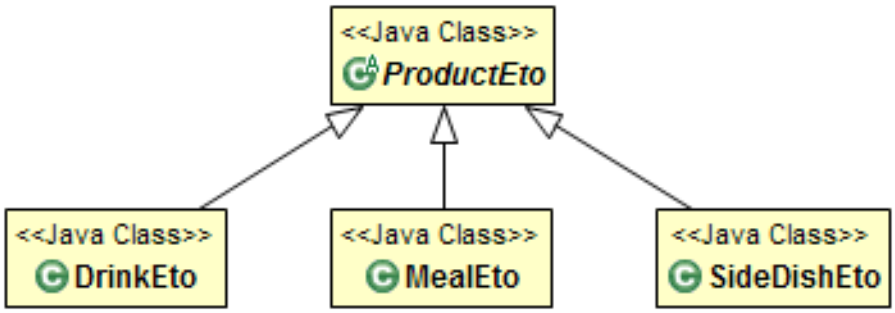


Figure 3. Transfer-Objects using Inheritance

Now assume you have a REST service operation as Java method that takes a ProductEto 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", ... }
```

Therefore you add configuration code to the constructor of ApplicationObjectMapperFactory. Here you can see an example from the sample application:

```
setBaseClasses(ProductEto.class);
addSubtypes(new NamedType(MealEto.class, "Meal"), new NamedType(DrinkEto.class, "
Drink"),
  new NamedType(SideDishEto.class, "SideDish"));
```

We use `setBaseClasses` to register all top-level classes of polymorphic objects. Further we declare all concrete polymorphic sub-classes together with their symbolic name for the JSON format via `addSubtypes`.

4.18.3. 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.

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 `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 with the following configuration code in the constructor of ApplicationObjectMapperFactory:+

```
SimpleModule module = getExtensionModule();
module.addDeserializer(MonetaryAmount.class, new MonetaryAmountJsonDeserializer());
module.addSerializer(MonetaryAmount.class, new MonetaryAmountJsonSerializer());
```

Now we can read and write `MonetaryAmount` from and to JSON as expected.

4.19. REST

REST (REpresentational State Transfer) is an inter-operable protocol for services 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. **ATTENTION:** REST and RESTful often implies very strict and specific rules and conventions. However different people will often have different opinions of such rules. We learned that this leads to "religious discussions" (starting from `PUT` vs. `POST` and IDs in path vs. payload up to Hypermedia and HATEOAS). These "religious discussions" waste a lot of time and money without adding real value in case of common business applications (if you publish your API on the internet to billions of users this is a different story). Therefore we give best practices that lead to simple, easy and pragmatic "HTTP APIs" (to avoid the term "REST services" and end "religious discussions"). Please also note that we do not want to assault anybody nor force anyone to follow our guidelines. Please read the following best practices carefully and be aware that they might slightly differ from what your first hit on the web will say about REST (see e.g. RESTful cookbook).

4.19.1. URLs

URLs are not case sensitive. Hence, we follow the best practice to use only lower-case-letters-with-hypen-to-separate-words. For operations in REST we distinguish the following types of URLs:

- A *collection URL* is build from the rest service URL 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 URL* is build from a collection URL 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`
- A *search URL* is build from a collection URL by appending the segment `search`. The search criteria is send as `POST`. Example: `https://mydomain.com/myapp/services/rest/mycomponent/v1/myentity/search`

This fits perfect for CRUD operations. For business operations (processing, calculation, etc.) we simply create a collection URL with the name of the business operation instead of the entity name (use a clear naming convention to avoid collisions). Then we can `POST` the input for the business operation and get the result back.

If you want to provide an entity with a different structure do not append further details to an element URL but create a separate collection URL as base. So use `https://mydomain.com/myapp/services/rest/mycomponent/v1/myentity-with-details/42` instead of `https://mydomain.com/myapp/services/rest/mycomponent/v1/myentity/42/with-details`. For offering a CTO simply append `-cto` to the collection URL (e.g. `⋯/myentity-cto/`).

4.19.2. HTTP Methods

While REST was designed as a pragmatical approach it sometimes leads to "religious discussions" e.g. about using `PUT` vs. `POST` (see ATTENTION notice above). As the devonfw has a string focus on usual business applications it 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 devonfw proposal.

Table 11. Usage of HTTP methods

HTTP Method	RESTful Meaning	devonfw
`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
`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 entire collection (typically not supported)	Delete an entity. Delete an entire collection (typically not supported)

Please consider these guidelines and rationales:

- We use `POST` on the collection URL to save an entity (`create` if no ID provided in payload otherwise `update`). This avoids pointless discussions in distinctions between `PUT` and `POST` and what to do if a `create` contains an ID in the payload or if an `update` is missing the ID property or contains a different ID in payload than in URL.
- 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.
- Please also note that for (large) bulk deletions you may be forced to used `POST` instead of `DELETE` as according to the HTTP standard `DELETE` must not have payload and URLs are limited in length.

4.19.3. 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 12. Usage of HTTP status codes

HTTP Code	Meaning	Response	Comment
200	OK	requested result	Result of successful GET

HTTP Code	Meaning	Response	Comment
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)

4.19.4. Metadata

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

Name	Description	Further information
X-Correlation-Id	HTTP header for a *correlation ID* that is 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.

4.19.5. 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 write an interface with JAX-RS annotations for the API and a regular implementation class annotated with `@Named` to make it a spring-bean. Here is a simple example: com.devonfw.application.mtsj.dishmanagement.service.impl.rest

```
@Path("/imagemanagement/v1")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public interface ImagemanagementRestService {

  @GET
  @Path("/image/{id}/")
  public ImageEto getImage(@PathParam("id") long id);

}

@Named("ImagemanagementRestService")
public class ImagemanagementRestServiceImpl implements ImagemanagementRestService {

  @Inject
  private Imagemanagement imagemanagement;

  @Override
  public ImageEto getImage(long id) {

    return this.imagemanagement.findImage(id);
  }

}
```

Here we can see a REST service for the business component `imagemanagement`. The method `getImage` can be accessed via HTTP GET (see `@GET`) under the URL path `imagemanagement/image/{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 `getImage`. It will return its result (`ImageEto`) as JSON (see `@Produces` - should already be defined as defaults in `RestService` marker interface). As you can see it delegates to the logic component `imagemanagement` that contains the actual business logic while the service itself only exposes this logic via HTTP. The REST service implementation is a regular CDI bean that can use dependency injection. The separation of the API as a Java interface allows to use it for service client calls.

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.

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.

4.19.6. REST Exception Handling

For exceptions a service needs to have an exception façade 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 devonfw provides a generic solution via `RestServiceExceptionFacade`. You need to follow the exception guide so that it works out of the box because the façade needs to be able to distinguish between business and technical exceptions. Now your service may throw exceptions but the façade with automatically handle them for you.

4.19.7. Recommendations for REST requests and responses

The devonfw proposes, for simplicity, a deviation from the common REST 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.

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

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"
        }
    ]
}
```

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: com.devonfw.application.mtsj.dishmanagement.service.api.rest

```
@Path("/category/search")
  @POST
  public PaginatedListTo<CategoryEto> findCategorysByPost(CategorySearchCriteriaTo
searchCriteriaTo) {
    return this.dishmanagement.findCategoryEtos(searchCriteriaTo);
 }
```

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

```
 @Path("/category/search")
  @POST @Path("/order")
  @GET
  public PaginatedListTo<CategoryEto> findCategorysByPost( @Context UriInfo info) {

    RequestParameters parameters = RequestParameters.fromQuery(info);
    CategorySearchCriteriaTo criteria = new CategorySearchCriteriaTo();
    criteria.setName(parameters.get("name", Long.class, false));
    criteria.setDescription(parameters.get("description", OrderState.class, false));
    criteria.setShowOrder(parameters.get("showOrder", OrderState.class, false));
    return this.dishmanagement.findCategoryEtos(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"
        ],
        // ....
    }
}
```

4.19.8. 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 devonfw 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.

4.19.9. 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)

4.19.10. Security

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

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 devon4j 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.

JSON top-level arrays

OWASP suggests to prevent returning JSON arrays at the top-level, to prevent attacks (see https://www.owasp.org/index.php/OWASP_AJAX_Security_Guidelines). However, no rationale is given at OWASP. We digged deep and found anatomy-of-a-subtle-json-vulnerability. To sum it up the attack is many years old and does not work in any recent or relevant browser. Hence it is fine to use arrays as top-level result in a JSON REST service (means you can return `List<Foo>` in a Java JAX-RS service).

4.20. SOAP

SOAP is a common protocol for services that is rather complex and heavy. It allows to build interoperable 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).

4.20.1. 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 =
"com.devonfw.application.mtsj.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.

4.20.2. 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).

4.20.3. SOAP Testing

For testing SOAP services in general consult the testing guide.

For testing SOAP services manually we strongly recommend SoapUI.

4.21. Service Client

This guide is about consuming (calling) services from other applications (micro-services). For providing services see the Service-Layer Guide. Services can be consumed in the client or the server. As the client is typically not written in Java you should consult the according guide for your client technology. In case you want to call a service within your Java code this guide is the right place to get help.

4.21.1. Motivation

Various solutions already exist for calling services such as `RestTemplate` from spring or the JAX-RS client API. Further each and every service framework offers its own API as well. These solutions might be suitable for very small and simple projects (with one or two such invocations). However, with the trend of microservices the invocation of a service becomes a very common use-case that occurs all over the place. Have a look at the following features to get an idea why you want to use the solution offered here.

4.21.2. Requirements

You need to add (at least one of) these dependencies to your application:

```
<!-- Starter for consuming REST services -->
<dependency>
  <groupId>com.devonfw.java.modules</groupId>
  <artifactId>devon4j-starter-cxf-client-rest</artifactId>
</dependency>
<!-- Starter for consuming SOAP services -->
<dependency>
  <groupId>com.devonfw.java.modules</groupId>
  <artifactId>devon4j-starter-cxf-client-ws</artifactId>
</dependency>
```

4.21.3. Features

When invoking a service you need to consider many cross-cutting aspects. You might not think about them in the very first place and you do not want to implement them multiple times redundantly. Therefore you should consider using this approach. The following sub-sections list the covered features and aspects:

Simple usage

Assuming you already have a Java interface `MyService` of the service you want to invoke:

```
package com.company.department.foo.mycomponent.service.api.rest;
...

@Path("/myservice")
public interface MyService extends RestService {

  @POST
  @Path("/getresult")
  MyResult getResult(MyArgs myArgs);
}
```

Then all you need to do is this:

```
@Named
public class UcMyUseCaseImpl extends MyUseCaseBase implements UcMyUseCase {
  @Inject
  private ServiceClientFactory serviceClientFactory;

  ...
  private MyResult callMyServiceMethod(MyArgs myArgs) {
    MyService myService = this.serviceClientFactory.create(MyService.class);
    MyResult myResult = myService.myMethod(myArgs); // synchronous call of service
over the wire
    return myResult;
  }
```

As you can see the synchronous invocation of a service is very simple. Still it is very flexible and powerful (see following features). The actual call of `myMethod` will technically call the remote service over the wire (e.g. via HTTP) including marshaling the arguments (e.g. converting `myArgs` to JSON) and unmarshalling the result (e.g. converting the received JSON to `myResult`).

Configuration

This solution allows a very flexible configuration on the following levels:

1. Global configuration (defaults)
2. Configuration per remote service application (microservice)
3. Configuration per invocation.

A configuration on a deeper level (e.g. 3) overrides the configuration from a higher level (e.g. 1).

The configuration on Level 1 and 2 are configured via `application.properties` (see configuration guide). For Level 1 the prefix `service.client.default.` is used for properties. Further, for level 2. the prefix `service.client.app.«application».` is used where `«application»` is the technical name of the application providing the service. This name will automatically be derived from the java package of the service interface (e.g. `foo` in `MyService` interface before) following our packaging conventions. In case these conventions are not met it will fallback to the fully qualified name of the service

interface.

Configuration on Level 3 has to be provided as `Map` argument to the method `ServiceClientFactory.create(Class<S> serviceInterface, Map<String, String> config)`. The keys of this `Map` will not use prefixes (such as the ones above). For common configuration parameters a type-safe builder is offered to create such map via `ServiceClientConfigBuilder`. E.g. for testing you may want to do:

```
this.serviceClientFactory.create(MyService.class,
  new ServiceClientConfigBuilder().authBasic().userLogin(login).userPassword(password
).buildMap());
```

TODO add configuration properties for external/mock service from Sneha

Service Discovery

You do not want to hardwire service URLs in your code, right? Therefore different strategies might apply to *discover* the URL of the invoked service. This is done internally by an implementation of the interface `ServiceDiscoverer`. The default implementation simply reads the base URL from the configuration (see above). So you can simply add this to your `application.properties`:

```
service.client.app.foo.url=https://foo.company.com:8443/services/rest
service.client.app.bar.url=http://bar.company.com:8080/services/rest
service.client.default.url=https://api.company.com/services/rest
```

Assuming your service interface would have the fully qualified name `com.company.department.foo.mycomponent.service.api.rest.MyService` then the URL would be resolved to `https://foo.company.com:8443/services/rest` as the `«application»` is `foo`.

Additionally, the URL might use the following variables that will automatically be resolved:

- `${app}` to `«application»` (useful for default URL)
- `${type}` to the type of the service. E.g. `rest` in case of a REST service and `ws` for a SOAP service.
- `${local.server.port}` for the port of your current Java servlet container running the JVM. Should only used for testing with spring-boot random port mechanism (technically spring can not resolve this variable but we do it for you here).

Therefore, the default URL may also be configured as:

```
service.client.default.url=https://api.company.com/${app}/services/${type}
```

As you can use any implementation of `ServiceDiscoverer`, you can also easily use eureka (or anything else) instead to discover your services.

Headers

A very common demand is to tweak (HTTP) headers in the request to invoke the service. May it be for security (authentication data) or for other cross-cutting concerns (such as the Correlation ID). This is done internally by implementations of the interface `ServiceHeaderCustomizer`. We already provide several implementations such as:

- `ServiceHeaderCustomizerBasicAuth` for basic authentication (mainly for testing).
- `ServiceHeaderCustomizerOAuth` for OAuth (passes a security token from security context such as a JWT via OAuth).
- `ServiceHeaderCustomizerCorrelationId` passed the Correlation ID to the service request.

Additionally, you can add further custom implementations of `ServiceHeaderCustomizer` for your individual requirements and additional headers.

Timeouts

You can configure timeouts in a very flexible way. First of all you can configure timeouts to establish the connection (`timeout.connection`) and to wait for the response (`timeout.response`) separately. These timeouts can be configured on all three levels as described in the configuration section above.

Error Handling

Whilst invoking a remote service an error may occur. This solution will automatically handle such errors and map them to a higher level `ServiceInvocationFailedException`. In general we separate two different types of errors:

- **Network error**
  In such case (host not found, connection refused, time out, etc.) there is not even a response from the server. However, in advance to a low-level exception you will get a wrapped `ServiceInvocationFailedException` (with code `ServiceInvoke`) with a readable message containing the service that could not be invoked.
- **Service error**
  In case the service failed on the server-side the error result will be parsed and thrown as a `ServiceInvocationFailedException` with the received message and code.

Logging

By default this solution will log all invocations including the URL of the invoked service, success or error status flag and the duration in seconds (with decimal nano precision as available). Therefore you can easily monitor the status and performance of the service invocations.

Asynchronous Support

An important aspect is also asynchronous (and reactive) support. So far we only propose this as an enhancement for a future release with an API like this:

```
public interface ServiceClientAsyncFactory extends ServiceClientFactory {
  AsyncClient<S> createAsync(Class<S> serviceInterface);
}

public interface AsyncClient<S> {
  S getClient();
  <R> Mono<R> call(R result);
  <T> Flux<T> call(Collection<? extends T> result);
  <R> void call(R result, Consumer<R> callback);
  <R> CompletableFuture<R> callFuture(R result);
}
```

This API would allow typesafe usage like this:

```
@Named
public class UcMyUseCaseImpl extends MyUseCaseBase implements UcMyUseCase {
  @Inject private ServiceClientFactoryAsync clientFactory;

  @Override @RolesAllowed(...)
  public void doSomething(Bar bar) {
    AsyncClient<MyExternalServiceApi> client = this.clientFactory.createAsync
(MyExternalServiceApi.class);
    Mono<Some> result = client.call(client.getService().doSomething(convert(bar)));
    // client.call(client.getService().doSomething(convert(bar)), x ->
processSync(x));
    return process(result);
  }
}
```

How can this work? The ServiceClientAsyncFactory implementation would create its own dynamic proxy for the given service interface. That proxy would only track the last call that was invoked internally and always return a dummy result (`null` for Object types, `false` for boolean, `0` for primitive numbers). The actual implementation of the `call` methods can access the internal invocation that has been recorded from the last service call. It will then trigger the actual service call internally according to the desired style (using a `Consumer` callback, `Mono`, `Flux`, `Future`...).

Resilience

Resilience adds a lot of complexity and that typically means that addressing this here would most probably result in not being up-to-date and not meeting all requirements. Therefore we recommend something completely different: the *sidecar* approach (based on sidecar pattern). This means that you use a generic proxy app that runs as a separate process on the same host, VM, or container of your actual application. Then in your app you are calling the service via the sidecar proxy on `localhost` (service discovery URL is e.g. `http://localhost:8081/${app}/services/${type}`) that then acts as proxy to the actual remote service. Now aspects such as resilience with circuit breaking and the actual service discovery can be configured in the sidecar proxy app and independent of your actual application. Therefore, you can even share and reuse configuration and

experience with such a sidecar proxy app even across different technologies (Java, .NET/C#, Node.JS, etc.).

Various implementations of such sidecar proxy apps are available as free open source software. **TODO**: Decide for the best available solution and suggest here as default.

- Netflix Sidecar - see Spring Cloud Netflix docs
- Envoy - see Microservices Patterns With Envoy Sidecar Proxy
- Prana - see Prana: A Sidecar for your Netflix PaaS based Applications and Services ← **Not updated as it's not used internally by Netflix**
- Keycloak - see Protecting Jaeger UI with a sidecar security proxy

4.22. Testing

4.22.1. 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 `assertThat(service.getAllEntities()).hasSize(42)` or even worse tests that have no assertion at all.
- 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 AssertJ to write good readable and maintainable tests that also provide valuable feedback in case a test fails. Do not use legacy JUnit methods like `assertEquals` anymore!
- For easy understanding divide your test in three commented sections:
  - `//given`
  - `//when`
  - `//then`
- 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.
- Prefer delegation over inheritance for cross-cutting testing functionality. Good places to put this kind of code can be realized and reused via the JUnit @Rule mechanism.

4.22.2. 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 sub-systems we recommend to use spring-boot-starter-test as lightweight and fast testing infrastructure that is already shipped with `devon4j-test`.

In case you have to use a full blown JEE application server, we recommend to use arquillian. To get started with arquillian, look here.

4.22.3. Test Doubles

We use test doubles as generic term for mocks, stubs, fakes, dummies, or spys to avoid confusion. Here is a short summary from stubs VS mocks:

- **Dummy** objects specifying no logic at all. May declare data in a POJO style to be used as boiler plate code to parameter lists or even influence the control flow towards the test's needs.
- **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.

We try to give some examples, which should make it somehow clearer:

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 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.

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.

Wiremock

If you need to mock remote connections such as HTTP-Servers, wiremock offers easy to use functionality. For a full description see the homepage or the github repository. Wiremock can be used either as a JUnit Rule, in Java outside of JUnit or as a standalone process. The mocked server can be configured to respond to specific requests in a given way via a fluent Java API, JSON files and JSON over HTTP. An example as an integration to JUnit can look as follows.

```
import static
com.github.tomakehurst.wiremock.core.WireMockConfiguration.wireMockConfig;
import com.github.tomakehurst.wiremock.junit.WireMockRule;

public class WireMockOfferImport{

  @Rule
  public WireMockRule mockServer = new WireMockRule(wireMockConfig().dynamicPort());

  @Test
  public void requestDataTest() throws Exception {
  int port = this.mockServer.port();
  ...}
```

This creates a server on a randomly chosen free port on the running machine. You can also specify the port to be used if wanted. Other than that there are several options to further configure the server. This includes HTTPs, proxy settings, file locations, logging and extensions.

```
  @Test
  public void requestDataTest() throws Exception {
      this.mockServer.stubFor(get(urlEqualTo("/new/offers")).withHeader("Accept",
equalTo("application/json"))
      .withHeader("Authorization", containing("Basic")).willReturn(aResponse()
.withStatus(200).withFixedDelay(1000)
      .withHeader("Content-Type", "application/json").withBodyFile(
"/wireMockTest/jsonBodyFile.json")));
  }
```

This will stub the URL `localhost:port/new/offers` to respond with a status 200 message containing a header (`Content-Type: application/json`) and a body with content given in `jsonBodyFile.json` if the request matches several conditions. It has to be a GET request to `../new/offers` with the two given header properties.

Note that by default files are located in `src/test/resources/__files/`. When using only one WireMock server one can omit the `this.mockServer` in before the `stubFor` call (static method). You can also add a fixed delay to the response or processing delay with `WireMock.addRequestProcessingDelay(time)` in order to test for timeouts.

WireMock can also respond with different corrupted messages to simulate faulty behaviour.

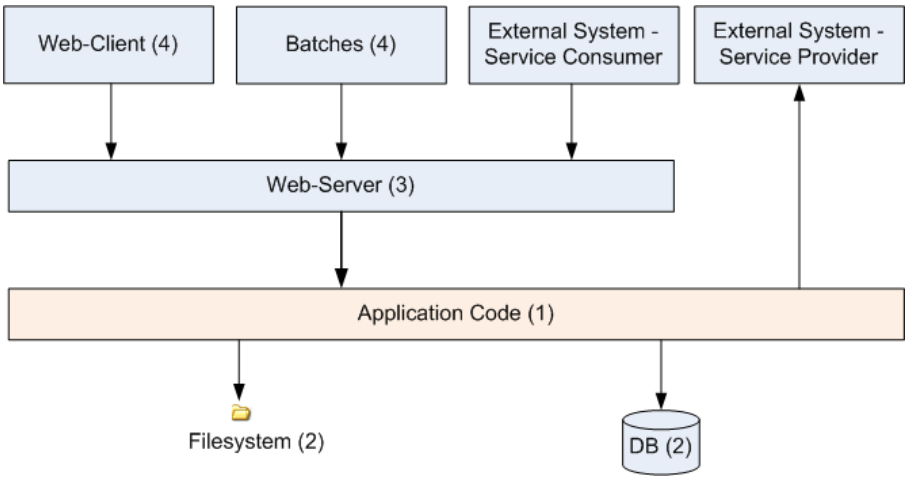
```
@Test(expected = ResourceAccessException.class)
public void faultTest() {

    this.mockServer.stubFor(get(urlEqualTo("/fault")).willReturn(aResponse()
    .withFault(Fault.MALFORMED_RESPONSE_CHUNK)));
...}
```

A GET request to `../fault` returns an OK status header, then garbage, and then closes the connection.

4.22.4. 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, devonfw 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 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 devonfw 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. devonfw 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.

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.

```
public class MyClassTest extends ModuleTest {

  @Test
  public void testMyClass() {

    // given
    MyClass myClass = new MyClass();
    // when
    String value = myClass.doSomething();
    // then
    assertThat(value).isEqualTo("expected value");
  }

}
```

For an advanced example see here.

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-boot-starter-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.

With devon4j and spring you can write a component-test as easy as illustrated in the following example:

```
@SpringBootTest(classes = { MySpringBootApp.class }, webEnvironment = WebEnvironment
.NONE)
public class UcFindCountryTest extends ComponentTest {
  @Inject
  private UcFindCountry ucFindCountry;

  @Test
  public void testFindCountry() {

    // given
    String countryCode = "de";

    // when
    TestUtil.login("user", MyAccessControlConfig.FIND_COUNTRY);
    CountryEto country = this.ucFindCountry.findCountry(countryCode);

    // then
    assertThat(country).isNotNull();
    assertThat(country.getCountryCode()).isEqualTo(countryCode);
    assertThat(country.getName()).isEqualTo("Germany");
  }
}
```

This test will start the entire spring-context of your app (`MySpringBootApp`). Within the test spring will inject according spring-beans into all your fields annotated with `@Inject`. In the test methods you can use these spring-beans and perform your actual tests. This pattern can be used for testing DAOs/Repositories, Use-Cases, or any other spring-bean with its entire configuration including database and transactions.

When you are testing use-cases your authorization will also be in place. Therefore, you have to simulate a logon in advance what is done via the `login` method in the above example. The test-infrastructure will automatically do a `logout` for you after each test method in `doTearDown`.

Level 3 Subsystem Test

A *subsystem test* runs against the external interfaces (e.g. HTTP service) of the integrated subsystem. Subsystem tests of the client subsystem are described in the <<,devon4ng-Wiki>>. In devon4j 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.

With devon4j and spring you can write a subsystem-test as easy as illustrated in the following example:

```
@SpringBootTest(classes = { MySpringBootApp.class }, webEnvironment = WebEnvironment
.RANDOM_PORT)
public class CountryRestServiceTest extends SubsystemTest {

  @Inject
  private ServiceClientFactory serviceClientFactory;

  @Test
  public void testFindCountry() {

    // given
    String countryCode = "de";

    // when
    CountryRestService service = this.serviceClientFactory.create(CountryRestService
.class);
    CountryEto country = service.findCountry(countryCode);

    // then
    assertThat(country).isNotNull();
    assertThat(country.getCountryCode()).isEqualTo(countryCode);
    assertThat(country.getName()).isEqualTo("Germany");
  }
}
```

Even though not obvious on the first look this test will start your entire application as a server on a free random port (so that it works in CI with parallel builds for different branches) and tests the invocation of a (REST) service including (un)marshalling of data (e.g. as JSON) and transport via HTTP (all in the invocation of the `findCountry` method).

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.

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 devonfw only gives advice for automated system test (TODO see allure testing framework). In nearly every project there must be manual system tests, too. This manual system tests are out of scope here.

Classifying Integration-Levels

devon4j defines Category-Interfaces that shall be used as JUnit Categories. Also devon4j provides abstract base classes that you may extend in your test-cases if you like.

devon4j 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.

4.22.5. Implementation

This section introduces how to implement tests on the different levels with the given devonfw infrastructure and the proposed frameworks.

Module Test

In devon4j you can extend the abstract class ModuleTest to basically get access to assertions. In order to test classes embedded in dependencies and external services one needs to provide mocks for that. As the technology stack recommends we use the Mockito framework to offer this functionality. The following example shows how to implement Mockito into a JUnit test.

```
import static org.mockito.Mockito.when;
import static org.mockito.Mockito.mock;
...

public class StaffmanagementImplTest extends ModuleTest {
  @Rule
  public MockitoRule rule = MockitoJUnit.rule();

  @Test
  public void testFindStaffMember() {
  ...}
}
```

Note that the test class does not use the `@SpringApplicationConfiguration` annotation. In a module test one does not use the whole application. The JUnit rule is the best solution to use in order to get all needed functionality of Mockito. Static imports are a convenient option to enhance readability within Mockito tests. You can define mocks with the `@Mock` annotation or the `mock(*.class)` call. To inject the mocked objects into your class under test you can use the `@InjectMocks` annotation. This automatically uses the setters of `StaffmanagementImpl` to inject the defined mocks into the *class under test (CUT)* when there is a setter available. In this case the `beanMapper` and the `staffMemberDao` are injected. Of course it is possible to do this manually if you need more control.

```
  @Mock
  private BeanMapper beanMapper;
  @Mock
  private StaffMemberEntity staffMemberEntity;
  @Mock
  private StaffMemberEto staffMemberEto;
  @Mock
  private StaffMemberDao staffMemberDao;
  @InjectMocks
  StaffmanagementImpl staffmanagementImpl = new StaffmanagementImpl();
```

The mocked objects do not provide any functionality at the time being. To define what happens on a method call on a mocked dependency in the CUT one can use `when(condition).thenReturn(result)`. In this case we want to test `findStaffMember(Long id)` in the StaffmanagementImpl.

```
public StaffMemberEto findStaffMember(Long id) {
  return getBeanMapper().map(getStaffMemberDao().find(id), StaffMemberEto.class);
}
```

In this simple example one has to stub two calls on the CUT as you can see below. For example the method call of the CUT `staffMemberDao.find(id)` is stubbed for returning a mock object `staffMemberEntity` that is also defined as mock.

Subsystem Test

devon4j provides a simple test infrastructure to aid with the implementation of subsystem tests. It becomes available by simply subclassing AbstractRestServiceTest.java.

```
//given
long id = 1L;
Class<StaffMemberEto> targetClass = StaffMemberEto.class;
when(this.staffMemberDao.find(id)).thenReturn(this.staffMemberEntity);
when(this.beanMapper.map(this.staffMemberEntity, targetClass)).thenReturn(this
.staffMemberEto);

//when
StaffMemberEto resultEto = this.staffmanagementImpl.findStaffMember(id);

//then
assertThat(resultEto).isNotNull();
assertThat(resultEto).isEqualTo(this.staffMemberEto);
```

After the test method call one can verify the expected results. Mockito can check whether a mocked method call was indeed called. This can be done using Mockito `verify`. Note that it does not generate any value if you check for method calls that are needed to reach the asserted result anyway. Call verification can be useful e.g. when you want to assure that statistics are written out without actually testing them.

TODO

As an example let us go to the class Tablemanagement. When testing the method *deleteTable()* there are several scenarios that can happen and thus should be covered by tests.

First let us see the valid conditions to delete a table:

- One needs permission to delete a table PermissionConstants.DELETE_TABLE
- The table to delete needs to exist (the table with the given id has to be in the database) and
- The table to delete is required to be TableState.FREE

Invalid conditions are: No credentials, table does not exist or table is not free. If you combine one invalid condition with valid conditions this yields the following test cases. Note that not working actions yield exceptions that can be expected in a test method.

- The caller of the method does not have the required credentials

```
@Test(expected = AccessDeniedException.class)
public void testDeleteTableWithoutCredentials() {...}
```

- The caller has the required credentials but the table to be deleted is occupied

```
@Test(expected = IllegalEntityStateException.class)
public void testDeleteTableWithCredentialsButNotDeletable() {...}
```

- The caller has the required credentials but the table to be deleted does not exist

```
@Test(expected = ObjectNotFoundUserException.class)
public void testDeleteTableWithCredentialsNotExisting() {...}
```

- The caller has the required credentials and the table to be deleted exists and is free

```
@Test
public void testDeleteTableWithCredentials() {...}
```

This type of testing is known as equivalence class analysis. Note that this is a general practice and can be applied to every level of tests.

4.22.6. 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 devon4j 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 devonfw.

4.22.7. 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.

4.22.8. Test Configuration

This section covers test configuration in general without focusing on integration levels as in the first chapter.

Configure Test Specific Beans

Sometimes it can become handy to provide other or differently configured bean implementations via CDI than those available in production. For example, when creating beans using `@Bean` -annotated methods they are usually configured within those methods. WebSecurityBeansConfig shows an example of such methods.

```
@Configuration
public class WebSecurityBeansConfig {
  //...
  @Bean
  public AccessControlSchemaProvider accessControlSchemaProvider() {
    // actually no additional configuration is shown here
    return new AccessControlSchemaProviderImpl();
  }
  //...
}
```

`AccessControlSchemaProvider` allows to programmatically access data defined in some XML file, e.g. `access-control-schema.xml`. Now, one can imagine that it would be helpful if `AccessControlSchemaProvider` would point to some other file than the default within a test class. That file could provide content that differs from the default. The question is: how can I change resource path of `AccessControlSchemaProviderImpl` wihtin a test?

One very helpful solution is to use **static inner classes**. Static inner classes can contain `@Bean` -annotated methods, and by placing them in the `classes` parameter in `@SpringBootTest(classes = { /* place class here*/ })` annotation the beans returned by these methods are placed in the application context during test execution. Combining this feature with inheritance allows to override methods defined in other configuration classes as shown in the following listing where `TempWebSecurityConfig` extends `WebSecurityBeansConfig`. This relationship allows to override `public AccessControlSchemaProvider accessControlSchemaProvider()`. Here we are able to configure the instance of type `AccessControlSchemaProviderImpl` before returning it (and, of course, we could also have used a completely different implementation of the `AccessControlSchemaProvider` interface). By overriding the method the implementation of the super class is ignored, hence, only the new implementation is called at runtime. Other methods defined in `WebSecurityBeansConfig` which are not overridden by the subclass are still dispatched to `WebSecurityBeansConfig`.

```
//... Other testing related annotations
@SpringBootTest(classes = { TempWebSecurityConfig.class })
public class SomeTestClass {

  public static class TempWebSecurityConfig extends WebSecurityBeansConfig {

    @Override
    @Bean
    public AccessControlSchemaProvider accessControlSchemaProvider() {

      ClassPathResource resource = new ClassPathResource(locationPrefix + "access-
control-schema3.xml");
      AccessControlSchemaProviderImpl accessControlSchemaProvider = new
AccessControlSchemaProviderImpl();
      accessControlSchemaProvider.setAccessControlSchema(resource);
      return accessControlSchemaProvider;
    }
  }
}
```

The following chapter of the Spring framework documentation explains issue, but uses a slightly different way to obtain the configuration.

Test Data

It is possible to obtain test data in two different ways depending on your test's integration level.

4.22.9. Debugging Tests

The following two sections describe two debugging approaches for tests. Tests are either run from within the IDE or from the command line using Maven.

Debugging with the IDE

Debugging with the IDE is as easy as always. Even if you want to execute a `SubsystemTest` which needs a Spring context and a server infrastructure to run properly, you just set your breakpoints and click on Debug As → JUnit Test. The test infrastructure will take care of initializing the necessary infrastructure - if everything is configured properly.

Debugging with Maven

Please refer to the following two links to find a guide for debugging tests when running them from Maven.

- http://maven.apache.org/surefire/maven-surefire-plugin/examples/debugging.html
- https://www.eclipse.org/jetty/documentation/9.3.x/debugging-with-eclipse.html

In essence, you first have to start execute a test using the command line. Maven will halt just before the test execution and wait for your IDE to connect to the process. When receiving a connection the

test will start and then pause at any breakpoint set in advance. The first link states that tests are started through the following command:

```
mvn -Dmaven.surefire.debug test
```

Although this is correct, it will run *every* test class in your project and - which is time consuming and mostly unnecessary - halt before each of these tests. To counter this problem you can simply execute a single test class through the following command (here we execute the `TablemanagementRestServiceTest` from the restaurant sample application):

```
mvn test -Dmaven.surefire.debug test -Dtest=TablemanagementRestServiceTest
```

It is important to notice that you first have to execute the Maven command in the according submodule, e.g. to execute the `TablemanagementRestServiceTest` you have first to navigate to the core module's directory.

4.23. 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.

4.23.1. 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.

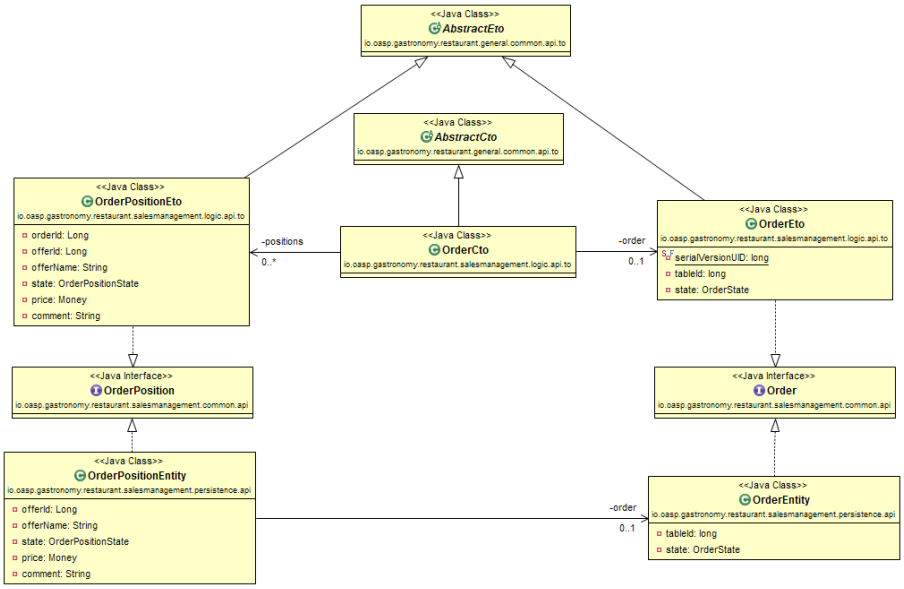


Figure 4. ETOs and CTOs

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.

4.23.2. 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).

4.24. 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 but not to mapping code):

```
public UserEto mapUser(UserEntity source) {
  UserEto target = new UserEto();
  target.setUsername(source.getUsername());
  target.setEmail(source.getEmail());
  ...
  return target;
}
```

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

4.24.1. Bean-Mapper Dependency

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

```
    <dependency>
      <groupId>com.devonfw.java</groupId>
      <artifactId>devon4j-beanmapping</artifactId>
    </dependency>
```

4.24.2. Bean-Mapper Configuration

The `BeanMapper` implementation is based on an existing open-source bean mapping framework. In case of Dozer the mapping is configured `src/main/resources/config/app/common/dozer-mapping.xml`.

See the my-thai-star dozer-mapping.xml as an example. Important is that you configure all your custom datatypes as `<copy-by-reference>` tags and have the mapping from `PersistenceEntity` (`ApplicationPersistenceEntity`) to `AbstractEto` configured properly:

```
 <mapping type="one-way">
    <class-a>com.devonfw.module.basic.common.api.entity.PersistenceEntity</class-a>
    <class-b>com.devonfw.module.basic.common.api.to.AbstractEto</class-b>
    <field custom-converter=
"com.devonfw.module.beanmapping.common.impl.dozer.IdentityConverter">
      <a>this</a>
      <b is-accessible="true">persistentEntity</b>
    </field>
</mapping>
```

4.24.3. 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:

```
...
UserEntity resultEntity = ...;
...
return getBeanMapper().map(resultEntity, UserEto.class);
```

There is also additional support for mapping entire collections.

4.25. 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.

4.25.1. Datatype Packaging

For the devonfw 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 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.

4.25.2. Technical Concerns

It has been exposed, that multiple technologies like Dozer and QueryDSL's (alias API) are heavily based on reflection to make the programmers world easier. However, to let them work properly with custom datatypes, the frameworks have to be able to instantiate custom datatypes by non-argument constructors. Therefore, we propose to implement a non-argument constructor for each datatype of at least protected visibility.

4.25.3. Datatypes in Entities

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

4.25.4. Datatypes in Transfer-Objects

XML

For mapping datatypes with JAXB see XML guide.

JSON

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

4.26. 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

4.27. CORS support

When you are developing Javascript client and server application separately, you have to deal with cross domain issues. We have to request from a origin domain distinct to target domain and browser does not allow this.

So , we need to prepare server side to accept request from other domains. We need to cover the following points:

- Accept request from other domains.
- Accept devonfw used headers like `X-CSRF-TOKEN` or `correlationId.`
- Be prepared to receive secured request (cookies).

It is important to note that if you are using security in your request (sending cookies) you have to set `withCredentials` flag to `true` in your client side request and deal with special IE8 characteristics.

4.27.1. Configuring CORS support

On the server side we have defined a new filter in Spring security chain filters to support CORS and we have configured devonfw security chain filter to use it.

You only have to change `CORSDisabled` property value in `application-default.properties` properties file.

```
#CORS support
security.cors.enabled=false
```

4.28. BLOB support

4.28.1. Introduction

BLOB stands for **B**inary **L**arge **Ob**ject. A BLOB may be an image, an office document, ZIP archive or any other multimedia object. devon4j supports BLOB via its BinaryObject data type. The devonfw Maven archetype generates the following Java files for dealing with BLOBs:

`general.common.api.BinaryObject`	Interface for a BinaryObject
`general.dataaccess.api.BinaryObjectEntity`	Instance of BinaryObject entity, contains the actual BLOB
`general.dataaccess.api.dao.BinaryObjectDao.java`	DAO for BinaryObject entity
`general.dataaccess.base.dao.BinaryObjectDaoImpl`	Implemenentation of the BinaryObjectDao
`general.logic.api.to.BinaryObjectEto`	ETO for BinaryObject
`general.logic.base.UcManageBinaryObject`	Use case for managing BinaryObject. This use case contains methods for finding, getting, deleting and saving a BLOB.
`general.logic.impl.UCManageBinaryObjectImpl`	Implemenentation of the UcManageBinaryObject

4.28.2. Implementing BLOB support: an example

In the sample application the business component Offermanagement uses BLOBs for product pictures. Feel free to use the following approach as starting point for BLOB support in your application.

Logic Layer

Use the methods declared in `general.logic.base.UcManageBinaryObject` in the implementation of your business component. Let's take a look at an example from the sample application.

The method

```
OffermanagementImpl.updateProductPicture(Long productId, Blob blob, BinaryObjectEto
binaryObjectEto)
```

saves a new picture for a given product.

This is done by calling an appropriate method, declared in the BinaryObject use case.

```
@Override
@RolesAllowed(PermissionConstants.SAVE_PRODUCT_PICTURE)
public void updateProductPicture(Long productId, Blob blob, BinaryObjectEto
binaryObjectEto) {

    ...
      binaryObjectEto = getUcManageBinaryObject().saveBinaryObject(blob,
binaryObjectEto);
    ...
 }
```

Service Layer

Following the devonfw conventions, you must implement a REST service for each business component. There you define, how BLOBs are uploaded/downloaded. According to that, the REST service for the business component Offermanagement is implemented in a class named OffermanagementRestServiceImpl.

The coding examples below are taken from the afore mentioned class.

The sample application uses the content-type "multipart/mixed" to transfer pictures plus additional header data.

Upload

```
@Consumes("multipart/mixed")
@POST
@Path("/product/{id}/picture")
  public void updateProductPicture(@PathParam("id") long productId,
      @Multipart(value = "binaryObjectEto", type = MediaType.APPLICATION_JSON)
BinaryObjectEto binaryObjectEto,
      @Multipart(value = "blob", type = MediaType.APPLICATION_OCTET_STREAM)
InputStream picture)
      throws SerialException, SQLException, IOException {

    Blob blob = new SerialBlob(IOUtils.readBytesFromStream(picture));
    this.offerManagement.updateProductPicture(productId, blob, binaryObjectEto);

}
```

A new Blob object is being created by reading the data (`IOUtils.readBytesFromStream(picture)`).

Download

```
@Produces("multipart/mixed")
@GET
@Path("/product/{id}/picture")
public MultipartBody getProductPicture(@PathParam("id") long productId) throws
SQLException, IOException {

    Blob blob = this.offerManagement.findProductPictureBlob(productId);
    byte[] data = IOUtils.readBytesFromStream(blob.getBinaryStream());

    List<Attachment> atts = new LinkedList<>();
    atts.add(new Attachment("binaryObjectEto", MediaType.APPLICATION_JSON, this
.offerManagement
        .findProductPicture(productId)));
    atts.add(new Attachment("blob", MediaType.APPLICATION_OCTET_STREAM, new
ByteArrayInputStream(data)));
    return new MultipartBody(atts, true);
}
```

As you may have noticed, the data is loaded into the heap before it is added as an Attachement to the MultiPart body.

Caution!	Using a byte array will cause problems, when dealing with large BLOBs.

Why is the sample application using a byte array then?

As of now, there is no universal solid way of streaming a BLOB directly from a database to the client without reading the BLOB's content to memory, when streaming over a RESTful service based on JDBC and JAX RS. Following this approach means: whenever a file is uploaded or downloaded as BLOB it is loaded completely to memory before it is written to the database.

4.28.3. Further Reading

- The multipart content type
- JAX-RS : Support for Multiparts
- Component Implementation
- BLOBs and the Data Access Layer
- Security Vulnerability Unrestricted File Upload

5. Tutorials

5.1. Introduction

This is an step by step tutorial for starting an devonfw server application from setting up the environment to packaging for production.

The tutorial starts by setting up the programmer environment with the aid of the devon-ide project and verifies everything is correct by running the my-thai-start restaurant sample application of the devonfw project.

Afterwards a new blank application is created by using the provided archetypes and all generated files are reviewed to explain what devonfw is providing.

A classical CRUD use case is developed for creating, retrieving updating and deleting an entity. With this entity we introduce cross cutting concerns such as exception handling, validation and securing the access from the web.

Finally the sample will be ready for deployment to a web server so we will package it on a WAR (or EAR) file.

5.2. Creating a new application

5.2.1. Running the archetype

In order to create a new application you must use the archetype provided by devon4j which uses the maven archetype functionality.

To use the archetype provided by devon4j you can choose between 2 alternatives, create it from command line or, in more visual manner, within eclipse.

From command Line

To create a new application from command line, you must execute one of the following commands.

For only war packaging (arguments before `archetype:generate` identifies devonfw artifact):

```
mvn -DarchetypeVersion=3.0.0 -DarchetypeGroupId=com.devonfw.java.templates
-DarchetypeArtifactId=devon4j-template-server archetype:generate
-DgroupId=com.devonfw.application -DartifactId=sampleapp -Dversion=0.1.0-SNAPSHOT
-Dpackage=com.devonfw.application.sampleapp
```

Further providing additional properties (using `-D` parameter) you can customize the generated app:

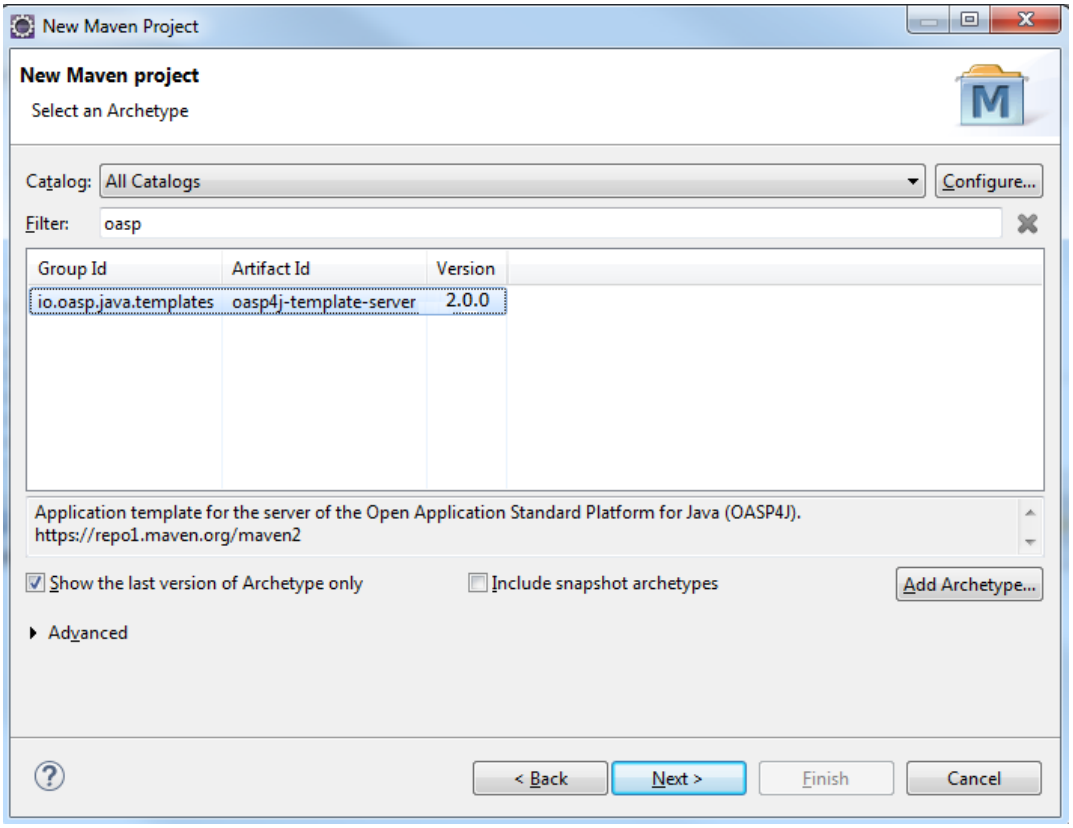
Table 13. Options for app template

property	comment	example
`dbType`	Choose the type of RDBMS to use (`hana`, `oracle`, `mssql`, `postgresql`, `mariadb`, `mysql`, etc.)	`-DdbTpye=postgresql`
`batch`	Option to add an `batch` module	`-Dbatch=batch`
`earProjectName`	Option to add an EAR module with the given name	`-DearProjectName=ear`

From Eclipse

To create a new application, you should have installed devonfw IDE. After that, you should follow this Eclipse steps to create your application:

- Create a new Maven Project.
- Choose the devon4j-template-server archetype, just like the image.



- Fill the Group Id, Artifact Id, Version and Package for your project. If you want to add an EAR generation mechanism to your project, you should fill the property earProjectName with the value Artifact Id + "-ear". For example, "sampleapp-ear". If you only want WAR generation, you can remove the property earProjectName.

[EAR] | *https://cloud.githubusercontent.com/assets/11404205/7963678/43421092-0a14-11e5-86a1-*

ed9202674339.PNG

- Finish the Eclipse assistant and you are ready to start your project.

5.2.2. What is generated

The application template (archetype) generates a Maven multi-module project. It has the following modules:

- `api`: module with the API (REST service interfaces, transferobjects, datatypes, etc.) to be imported by other apps as a maven dependency in order to invoke and consume the offered (micro)services.
- `core`: maven module containing the core of the application.
- `batch`: optional module for batch(es)
- `server`: module that bundles the entire app (`core` with optional `batch`) as a WAR file.
- `ear`: optional maven module is responsible to packaging the application as a EAR file.

The toplevel `pom.xml` of the generated project has the following features:

- Properties definition: Spring-boot version, Java version, etc.
- Modules definition for the modules (described above)
- Dependency management: define versions for dependencies of the technology stack that are recommended and work together in a compatible way.
- Maven plugins with desired versions and configuration
- Profiles for test stages

Core Module

Core module constains the base classes and the base configuration for the application. We are going to describe each Java file and each XML configuration file that archetype has generated.

Java

Those are the different Java files contained in each package:

- general.common

File	Descripcion
api.ApplicationEntity.java	Abstract interface for a MutableGenericEntity of this application.
api.BinaryObject.java	Interface for a BinaryObject.
api.NlsBundleApplicationRoot.java	NlsBundle for this application.
api.Usermanagement.java	Interface to get a user from its login.
api.UserProfile.java	Interface for the profile of a logged user.

File	Descripcion
api.constants.PermissionConstants.java	Constants for AccessControlPermission´s keys.
api.datatype.OrderBy.java	Enum for sort order.
api.datatype.Role.java	Enum for roles.
api.exception.ApplicationBusinessException.java	Abstract business main exception.
api.exception.ApplicationException.java	Abstract main exception.
api.exception.ApplicationTechnicalException.java	Abstract technical main exception.
api.exception.IllegalEntityStateException.java	Manage entities illegal state exceptions.
api.exception.IllegalPropertyChangeException.java	Manage entities illegal property changes exceptions.
api.exception.NoActiveUserException.java	Manage exceptions when user require to be logged in.
api.security.UserData.java	Container class for the profile of a user.
api.to.AbstractCto.java	Abstract class for Composite Transfer Object.
api.to.AbstractEto.java	Abstract class for Entity Transfer Object.
api.to.AbstractTo.java	Abstract class for a plain Transfer Object.
api.to.SearchCriteriaTo.java	Abstract class for a Transfer Object with the criteria for a search query.
api.to.UserDetailsClientTo.java	.
base.AbstractBeanMapperSupport.java	Provides access to the BeanMapper.
impl.security.ApplicationAuthenticationProvider.java	Responsible for the security aspects of authentication.
impl.security.PrincipalAccessControlProviderImpl.java	Implementation of PrincipalAccessControlProvider.

- general.dataaccess

File	Descripcion
api.ApplicationPersistenceEntity.java	Abstract Entity for all Entities with an id and a version field.
api.BinaryObjectEntity.java	BinaryObject entity.
api.dao.ApplicationDao.java	Interface for all DAOs of the application.
api.dao.ApplicationRevisionedDao.java	Interface for all revisioned DAOs of the application.
api.dao.BinaryObjectDao.java	DAO for BinaryObject entity.

- general.gui.api

File	Descripcion
LoginController.java	Controller for login page.

- general.logic

File	Descripcion
api.UseCase.java	Annotation to mark all use-cases.
api.to.BinaryObjectEto.java	ETO for a BinaryObject.
base.AbstractUc.java	Abstract base class for any use case in the application.
base.UcManageBinaryObject.java	Use case for managing BinaryObject.
impl.UcManageBinaryObjectImpl.java	Implementation of the UcManageBinaryObject interface.
impl.UsermanagementDummyImpl.java	Implementation of Usermanagement.

- general.service.impl.rest

File	Descripcion
ApplicationAccessDeniedHandler.java	Class to manage denied access.
ApplicationObjectMapperFactory.java	MappingFactory class to resolve polymorphic conflicts within the application.
SecurityRestServiceImpl.java	Class that represents REST service for security.

Resources

Those are the different XML files contained in resources folder:

- config

File	Descripcion
app.common.beans-common.xml	Constains beans definition for application common beans like propertyConfigurer bean.
app.common.beans-dozer.xml	Beans relationated with Dozer Mappers.
app.common.dozer-mapping.xml	Dozer mapping configuration.
app.dataaccess.beans-dataaccess.xml	Parent from the other data access files.
app.dataaccess.beans-db-plain.xml	Data source configuration for profile `db-plain` (testing).
app.dataaccess.beans-db-server.xml	Data source configuration for profile distinct to `db-plain.`
app.dataaccess.beans-jpa.xml	Contains neccessary beans to configure JPA.
app.dataaccess.NamedQueries.xml	
app.gui.dispatcher-servlet.xml	

File	Descripcion
app.logic.beans-logic.xml	Component scan configuration for classes in `logic` path.
app.security.access-control-schema.xml	
app.security.beans-security-filters.xml	Security filters definition.
app.security.beans-security.xml	Application security configuration.
app.service.beans-monitoring.xml	
app.service.beans-service.xml	Importing configuration files, REST beans definition and configuration.
app.websocket.websocket-context.xml	Scan component package definition for websockects.
app.application.default.properties	Default application properties values.
app	beans-application
Root file configuration. It starts the chain and imports other configuration files.	env
application	Specific application properties values.

- db

File	Descripcion
migration.V0001__Create_schema.slq	Script template to create the database schema and tables definition.

Test

Those are different Java files to serve as base classes in testing:

- general.common

File	Descripcion
AbstractSpringIntegrationTest.java	.
AccessControlSchemaXmlValidationTest.java	Tests if the access-control-schema.xml is valid.
PermissionCheckTest.java	Test to check if all relevant methods in use case implementations have permission checks.

Server Module

This module is constains two files:

- lockback.xml: This file is in the resources folder and it is the responsible to configure the log.
- pom.xml: This file has Maven configuration for packaging the application as a WAR. Also, this file has a profile to package the Javascript client ZIP file into the WAR.

EAR Module

This module only contains a pom.xml file to packaging the application as EAR from the WAR generated.

5.2.3. Database configuration and creation

Including driver installation if oracle or other db is required.

5.2.4. Editing the pom.xml

How to edit the pom.xml file for the project to add dependencies and modules for the application.

5.2.5. Known Issues

- Could not resolve archetype com.devonfw.java.templates:devon4j-template-server:.. from any of the configured repositories. In Eclipse: Open Window > Preferences Open Maven > Archetypes Click 'Add Remote Catalog' and add the following: Catalog File: http://repo1.maven.org/maven2/archetype-catalog.xml Description: maven catalog