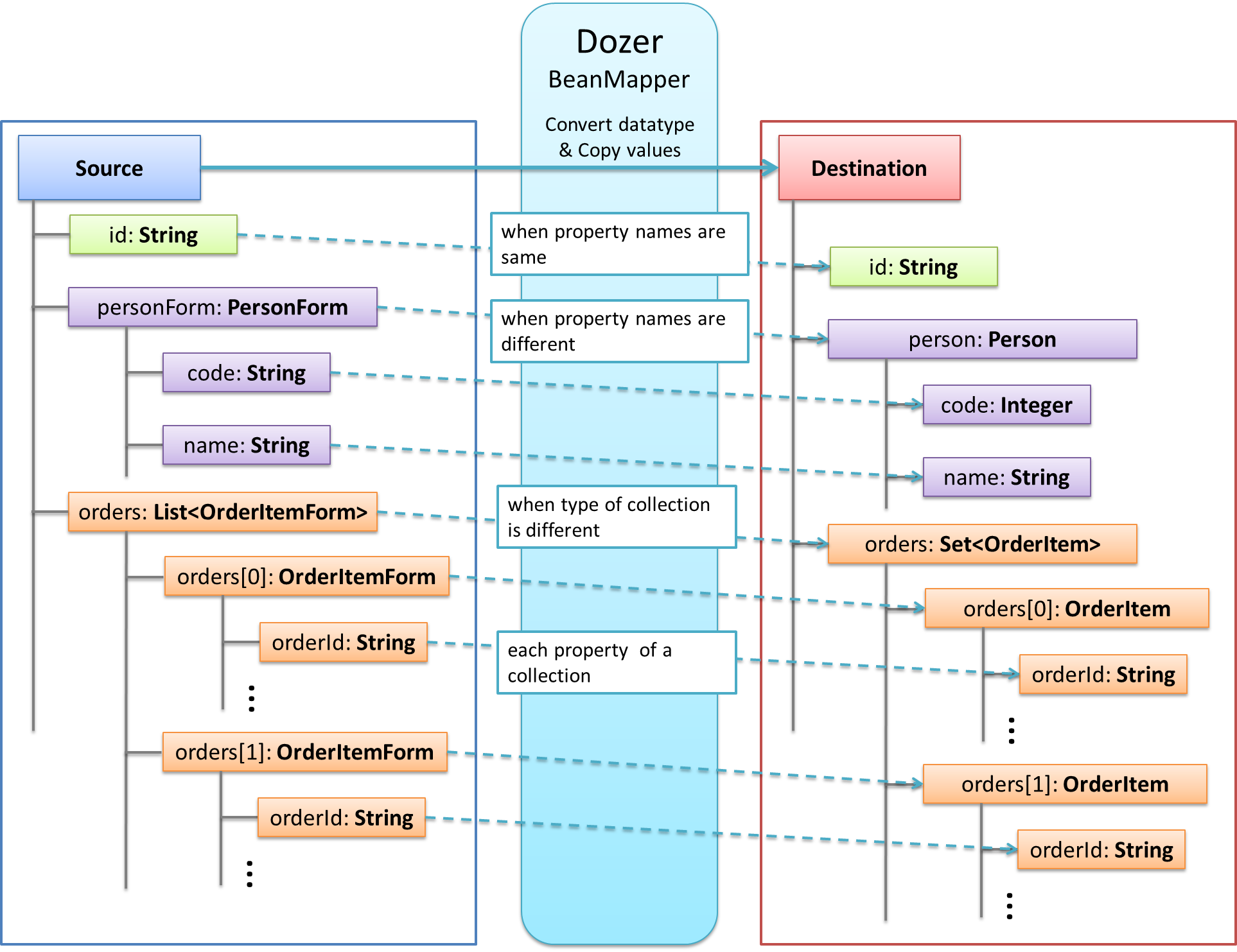
**Bean-Mapping**

**Why use Bean-Mapping**

A mapping framework is useful in a layered architecture, where you can create layers of abstraction by encapsulating changes to particular data objects vs. propagating these objects to other layers (i.e. External service data objects, domain objects, data transfer objects, internal service data objects). A mapping framework is an ideal and can be used within Mapper type classes that are responsible for mapping data from one data object to another.

Mapping between the data objects has been traditionally addressed by hand coding value object assemblers (or converters) that copy data between the objects. Most programmers will develop some sort of custom mapping framework and spend countless hours and thousands of lines of code mapping to and from their different data object.

A generic mapping framework solves these problems. Dozer (which is configured and used in devonfw) is an open source mapping framework that is robust, generic, flexible, reusable, and configurable.

Typically, Dozer works as shown below:

For decoupling, you sometimes need to create separate objects (beans) for a different view. For example, for an external service, you will use a [transfer-object](https://github.com/devonfw/devon4j/wiki/guide-transferobject) instead of the [persistence entity](https://github.com/devonfw/devon/wiki/getting-started-bean-mapping-using-dozer#Entity), so internal changes to the entity do not implicitly change or break the service.

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

public PersonTo mapPerson(PersonEntity source) {

PersonTo target = new PersonTo();

target.setFirstName(source.getFirstName());

target.setLastName(source.getLastName());

...

return target;

}

Therefore, BeanMapper is used for this purpose, which indirectly makes this task a lot easier.

**Bean-Mapper Dependency**

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

<dependency>

<groupId>io.oasp.java</groupId>

<artifactId>oasp4j-beanmapping</artifactId>

</dependency

So, (oasp4j -beanmapping) uses Dozer as dependency in its pom.xml file.

### Bean-Mapper Usage

Then, you can get the BeanMapper via [dependency-injection](https://github.com/devonfw/devon4j/wiki/guide-dependency-injection) which is typically provided by an abstract base class (e.g. AbstractUc). Now, your problem can be solved easily:

PersonEntity person = ...;

...

return getBeanMapper().map(person, PersonTo.class);

So, in the above piece of code, getBeanMapper() method provides an mapper (dozer) instance , and when map() method is called, it maps PersonEntity (source object) to PersonTo(DEstination object). Additionally, it supports the mapping of entire collections.

Dozer has been configured as a Spring bean in devonfw, using dependency injection. This is done in BeanDozerConfiguration.java which is present in resources/common/configuration folder of xxx-core project, created using devon4j template server archetype.

In this class, you can give path of mapping file (dozer-mapping.xml), which is generally placed at config/app/common/dozer-mapping.xml.

### Dozer Bean Mapper

### What is Dozer Bean mapper?

### Dozer is a Java Bean to Java Bean mapper that recursively copies data from one object to another, attribute by attribute.

### The mapping file defines all of the relationships between Java classes and their attributes. Dozer has a method called map which takes a source object and either a destination object or destination object class type.

### After mapping the two objects it then returns the destination object with all of its mapped fields.

### Example:

Mapper mapper = new DozerBeanMapper();

DestinationObject destObject = mapper.map(sourceObject, DestinationObject.class);

Or

DestinationObject destObject = new DestinationObject();

mapper.map(sourceObject, destObject);