Cityopt Architecture Documentation

# Application Architecture

CityOpt uses the Model/View/Controller pattern, combined with a layered architecture, were Data Transfer Objects (DTO) are used as a further abstraction from the Data Layer. This has the advantage, that DTOs can be customized for the Web Layer, containing only the necessary fields used by Controller and View and also combine Data from multiple models. Furthermore they could be used to add annotation based validation or other UI specific functionality and can prevent data access related exceptions in the controller layer.

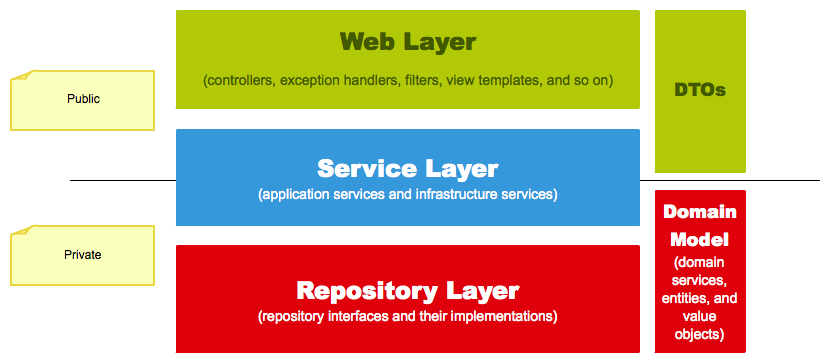


Figure 1: CityOpt Architecture with DTOs

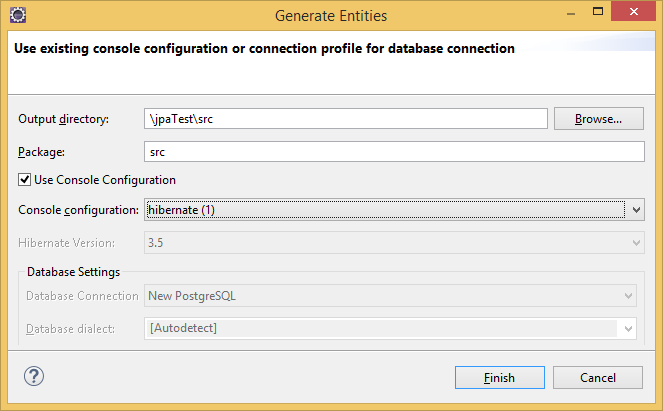
The downside is that they need translation from and too models and cause some redundancy because of the similarity to the model objects.

## Model classes (eu.cityopt.model)

### Generating the model classes

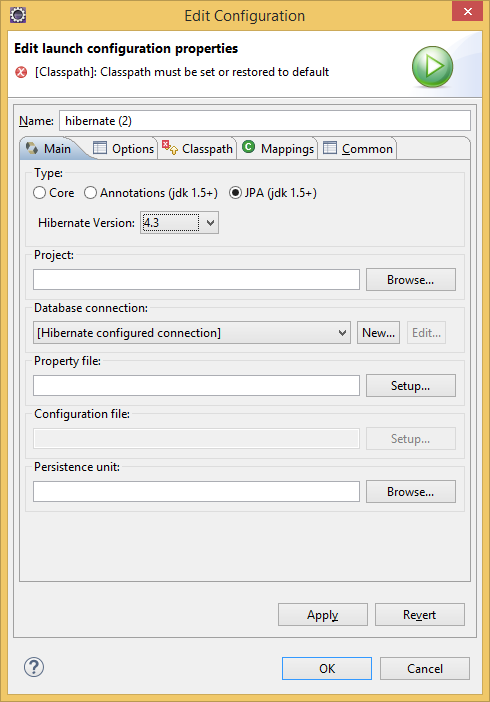
In Eclipse: Create new JPA project, with JPA version 2.1 and Hibernate as provider. Set up connection (see jpaContext.xml for parameters)

Right click on project 🡪 JPA Tools 🡪 Generate Entities from Tables. Select package as desired and use Console configuration:



If no console configuration exists, create it by:

* Open Eclipse Hibernate perspective.
* In the Hibernate Configurations view, click right and select “add configuration”. (If view isn’t visible show it by: Window🡪 show view 🡪 hibernate configuration)



* + Under options, select PostgresSQL as database dialect

### Configuration

Cityopt models use configuration by JPA annotations. Normally Hibernate does a good job configuring the model classes, except for the Id column, where it does not find the sequences as generation strategy. They need to be configured manually:

@SequenceGenerator(name="project\_prjid\_seq",sequenceName="project\_prjid\_seq")

@GeneratedValue(strategy = GenerationType.***SEQUENCE***, generator="project\_prjid\_seq")

@Id

@Column(name = "prjid", unique = **true**, nullable = **false**)

**public** **int** getPrjid() {

**return** **this**.prjid;

}

SequenceGenerator sets a name for the sequence and references the database sequence (=sequenceName). GeneratedValue tells Hibernate to use a sequence for generation of the identity column. The generator attribute has to match the name attribute under SequenceGenerator.

#### Cascade

The generated models don’t contain cascade behaviour by default. The different cascade options are:

1. **CascadeType.PERSIST** : means that save() or persist() operations cascade to related entities.
2. **CascadeType.MERGE** : means that related entities are merged into managed state when the owning entity is merged.
3. **CascadeType.REFRESH** : does the same thing for the refresh() operation.
4. **CascadeType.REMOVE** : removes all related entities association with this setting when the owning entity is deleted.
5. **CascadeType.DETACH** : detaches all related entities if a “manual detach” occurs.
6. **CascadeType.ALL** : is shorthand for all of the above cascade operations.

**CascadeType.DETACH**

Currently Cityopt only uses cascadeType.detach to be able to delete projects and scenarios with all their references. **Note that the deletion of TimeSeries is not cascaded to TimeSeriesVal on the models, as the performance impact is significant. This is currently the only cascade option which is set on the database (25.06.2015)**

(Read more about how cascade delete <https://eddii.wordpress.com/2006/11/16/hibernate-on-deletecascade-performance/> )

**CascadeType.PERSIST**

The advantages of cascade persist are, that children can save their parent objects (e.g. saving a scenario with a new project would need only one save operation) and references can be set bidirectional. When used intensively it gets harder to track errors, and it causes problems with references in the DTO classes.

For example, the components of a project:

@OneToMany(fetch = FetchType.***LAZY***, mappedBy = "project", cascade={CascadeType.***REMOVE***, CascadeType.***PERSIST***})

@OrderBy("componentid")

**public** List<Component> getComponents() {

**return** **this**.components;

}

The ProjectDTO used for saving does not contain the components. Therefore projects.components becomes null at a mapping/save operation of a project. If this was persisted to the components, the reference would always be erased when saving a project. To prevent this, the CascadeType.Persist needs to be removed. If Cascade.Persist should be included, the DTO then needs to include project.components (this could impact performance) or the servicemethod for updating has to keep the original components.

## Repositories

Cityopt uses Spring JpaRepository for the implementation of repository classes. The @Repository annotation marks a class as repository component and allows it to be added to the Spring IoC container automatically (component scan needs to be specified in the configuration).

Cityopt extends the generic JpaRepository interface to automatically implement CRUD operations for repositories. For example:

@Repository

**public** **interface** ScenarioRepository **extends** JpaRepository<Scenario,Integer>{

}

When building its IoC container, Spring automatically creates an implementation of the JpaRepository for the scenario model. When the ScenarioRepository is injected, methods like findAll, save and delete can be used. (see <http://docs.spring.io/spring-data/data-jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html> for the complete definition)

It is possible to add custom functionality to the repository by query methods, JPA Criteria API or third party frameworks like Querydsl. Cityopt only uses Query methods currently (29.06.2015). The 3 different query methods include:

* Queries by method name
* Named queries
* @Query annotation

### Queries by method name

Spring can automatically generate Queries out of the method name, when following naming conventions. Therefore find…/get…/read… will indicate Query creation methods. The start of the actual criteria is indicated by the “By” clause. Examples:

@Repository

**public** **interface** ProjectRepository **extends** JpaRepository<Project,Integer>{ Project findByName(String prjname);

List<Project> findByNameLikeIgnoreCase(String prjname);

List<Project> findByCreatedonBetween(Date from, Date to);

List<Project> findByNameContainingOrDescriptionContaining(String sc1, String sc2);

}

A detailed description of the keywords and their purpose can be found here: <http://docs.spring.io/spring-data/jpa/docs/1.3.0.RELEASE/reference/html/repositories.html#repositories.query-methods>

### Named queries

Named queries can be added by annotations in the model classes or by a XML file. @NamedQuery declares a named JPQL query, while native SQL is supported over @NamedNativeQuery.

@NamedQueries**({**

@NamedQuery**(**name **=** "Project.findByProjectName"**,**

query **=** "select distinct p from Project p where Lower(p.name) like CONCAT('%',Lower(:prjName),'%')"

resultClass **=** Project**.**class**)**

**})**

//call of the named query in ProjectRepository:

public List**<**Project**>** findByProjectName**(**@Param**(**"prjName"**)** String prjname**);**

The search for a named query method is performed before the query generation by method name. In the above example, findByProjectName would also work if no named query is provided. (The query is then generated by method name.)

### @Query annotation

Queries can be directly added to the repository over the @Query annotation. It supports native SQL and JPQL. The disadvantage compared to named queries is that the result is restricted to the model class the repository is created for (therefore ProjectRepository can only return one Project or a collection of Projects).

Examples

@Query("select distinct p from Project p where Lower(p.name) like CONCAT('%',Lower(:prjName),'%')")

List<Project> findByNameContaining(@Param("prjName") String prjname);

@Query("select epv from ExtParamValSetComp e JOIN e.extparamval as epv "

+ "where extParamValSetID = :epvsid")

List<ExtParamVal> findByExtParamValSetId(@Param("epvsid") **int** epvsId);

@Query("select mv from MetricVal mv LEFT JOIN FETCH mv.scenariometrics scm "

+ "where mv.metric.metid = :metId and scm.extparamvalset.extparamvalsetid = :extParamSetId")

**public** List<MetricVal> findByMetricAndEParamSet(@Param("metId") Integer metId, @Param("extParamSetId") Integer epvsId);

**Queries that perform writing operations need to be annotated by @Modifying:**

@Modifying

@Query("delete from ExtParamValSetComp e "

+ "where extParamValSetID = :epvsid and extParamValID = :epvId")

**void** removeExtParamValFromSet(@Param("epvsid") **int** epvsId, @Param("epvId") **int** epvId);

See also <http://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-three-custom-queries-with-query-methods/> or the Spring Data book.

## Services

Service components should contain the business logic of the application. In Cityopt, they are also responsible for the translation of Models🡪DTOs and back. Services are annotated with the @Service stereotype.

### DTOs

In Cityopt we use modelMapper (<http://modelmapper.org/>) for DTO translations. It works convention-based (e.g. naming convention) and needs very little mapping configuration in most cases. When translating a type the first time, it creates an internal map which is cached for further translations of the same types. Therefore it is good to share a ModelMapper instance across the application. ModelMapper provides deep mapping. When mapping a model with its references to a DTO, all references would be loaded eager in the DTO. For this reason, the DTOs should contain only necessary data to keep the application efficient.

For example: When the UI should display a list of inputParameters together with their componentname, it would be most efficient to implement a DTO like:

**public** **class** InputParamWithCompNameDTO **extends** BaseDTO{

@Getter @Setter **private** **int** inputid;

@Getter @Setter **private** String name;

@Getter @Setter **private** String defaultvalue;

@Getter @Setter **private** String componentName;

}

When an InputParameter model is mapped to the InputParamWithCompNameDTO, modelMapper detects that InputParameter contains a flied “component” and the component a “name” field. Therefor it can map this DTO without explicit mapping information. Explicit mapping information can be provided by a map, for the above case for example:

**public** **class** InputParamComponentMap **extends** PropertyMap<InputParameter, InputParamWithCompNameDTO>{

@Override

**protected** **void** configure() {

map().setName(source.getName());

map().setComponentName(source.getComponent().getName());

}

}

For the best performance, it is possible to retrieve objects by omitting the OR mapping framework. This can be done by using the JdbcTemplate and RowMappers. See <http://www.mkyong.com/spring/spring-jdbctemplate-querying-examples/> for examples, or the eu.cityopt.repository. CustomQueryRepositoryImpl

As the DTOs should be kept as slim as possible, most of collections were not included in the DTOs (except if they are always needed by the controller or view). As replacement for the references in the DTOs additional service methods were necessary.

Example - Project.components

Instead of including the components of the project in the projectDTO, there is a service method to retrieve a project’s components by the project’s id:

@Transactional(readOnly = **true**)

**public** List<ComponentDTO> getComponents(**int** prjid) {

Project item = projectRepository.findOne(prjid);

List<Component> components = item.getComponents();

**return** modelMapper.map(components, **new** TypeToken<List<ComponentDTO>>() {}.getType());

}

The project is retrieved from the repository. Then the project’s components are mapped to their respective DTO type and returned. Collections types are specified by creating a guava TypeToken (see here <http://stackoverflow.com/questions/7199916/does-modelmapper-library-support-collections-like-arraylist-or-hashset>).

## Configuration

### Web.xml

The src/main/webapp/WEB-INF/web.xml is the overall configuration for the web application. It ties together the application context and configures the servlet.

### Application Context

The Spring application context under src/main/resources/jpaContext.xml contains the configuration of Spring components. It enables annotation based configuration and sets locations (component scan) for Spring components. Other components can be added and configured by adding a bean tag, with id and its corresponding class definition. transactionManager, dataSource and modelMapper instance are configured this way and can therefore be injected into other classes.

See also:

Autowiring/Dependency Injection:

<http://www.mkyong.com/spring/spring-auto-wiring-beans-with-autowired-annotation/>

Component scan and Spring stereotypes:

<http://www.mkyong.com/spring/spring-auto-scanning-components/>

### Servlet Configuration

The servlet configuration is under eu.cityopt.config. appConfig.java. It contains relevant configurations for the servlet – like component scan for the controller or ViewResolvers. It is important to not include other Spring components, like Repositories and Services in the Servlet configuration, because this interferes with the transaction management.

### Transaction management

Transaction management is set to be annotation driven and configured by this section in jpaContext.xml:

<tx:annotation-driven transaction-manager=*"transactionManager"* proxy-target-class=*"true"*/>

<bean id=*"transactionManager"* class=*"org.springframework.orm.jpa.JpaTransactionManager"*>

<property name=*"entityManagerFactory"* ref=*"entityManagerFactory"*/>

</bean>

The attribute specifies another proxying mechanism than the default, which enables generating proxies for classes that don’t provide interfaces. (see also <http://stackoverflow.com/questions/15568112/using-proxy-target-class-true-with-spring-beans> for the implications)

### Test context

The test configuration is located under src/test/resources/test-context.xml. It only contains the definition of the dataSource. For the tests to work correctly, the normal application context needs to be loaded before the test-context, which only overwrites the dataSource then:

@ContextConfiguration(locations={"classpath:/jpaContext.xml", "classpath:/test-context.xml"})

## Testing

Unittests are implemented for Service and data access components and are rather Integrationtests than Unittests. The database is not isolated from the tests, but the tests use their own database instance called “CityOptEmptyTestDb”. As the name suggests, this database should always be empty, to provide all tests the same initial state.

A good introduction for writing tests for data access code:

<http://www.petrikainulainen.net/writing-tests-for-data-access-code/>

### DBunit

Some issues when working with DBUnit:

-Note that DBunit uses “clean insert”: Before the database is filled with the testdata, DBunit will try to delete existing data from affected tables. As there are no cascade constraints in the database (except on delete cascade for TimeSeries->TimeSeriesVal) this is likely to fail on references.

-When using the @ExpectedDatabase Annotation there were some issues with tests not rolling back. The Annotation @DirtiesContext can be used as a workaround. It tells Spring to recreate the application context after the test.

-Sequences are not updated:   
Note that the identity columns used in testdata do not update the sequences, even if no identity value is provided in the testdata. Therefore tests can cause constraint violations, when Hibernate uses the sequence values to insert data. Workarounds are incrementing the sequences manually of using the updateSequences method in CustomQueryRepository.

## Proxy

Spring uses aspect oriented programming (AOP) by enabling specific behaviour over annotations: like transactions, background processes or injections. The annotations are processed when Spring generates instances of Components and adds them to the Spring Inversion of Control (IoC) Container. Therefore it does not really generate instances of the actual classes, but proxies which are extended with behaviour as defined by the annotations.

Further reading:

<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop-api.html>

Important when working with proxies:

<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html#aop-proxying>