Generic micro application

v 0.2

Zdenek Vales

19th January 2016

1 About

Generic micro application is a stand-alone maven module and it serves as a base for developing various micro applications

Application uses its own database, its own spring context and has its own test classes, so the application can be fully developed and tested before its deployed and used in real time application.

1.1 Package structure

Base package is `org.microapp.microappName`, where `microappName` is the name of developed micro application. All code should be placed in this package (and created sub-packages).

1.2 Generic package

Package containing generic classes is `org.microapp.microappName.generic`. The `generic` package can also be placed in `org.microapp` but it causes problems (or rather confusion) when importing various generic classes from different micro applications

The generic package itself is based on AppFuse one-module project. However, it only contains base class for entities, generic DAO (JPA instead of Hibernate) and generic manager.

Base object This abstract class represents a basic entity. There is an ID field with proper annotations, so there is no need to declare this field again in child classes. Every entity should extend this class.

Base access object This abstract class is based on `BaseObject` and it represents entity related to a member of society. The person ID field in this class is meant to represent the membership ID from table membership (membernet database).

Generic DAO As there are two types of base objects, there are also two types of DAOs. The first type —`GenericDao`— is used as normal DAO in micro application It's designed for `BaseObject`, but since the `BaseAccessObject` is a child of `BaseObject` it's also possible to access objects of this type. Same as in AppFuse, you don't have to write a new DAO if you need just CRUD functions. You can just create a new bean based on this class.

Generic access DAO The second type is `GenericAccessDao` and it's used for DAOs which are accessing objects related with a member of society. This DAO is designed for `BaseAccessObject`. Right now, only difference from Generic DAO is the `getAllForPerson()` method.

Generic manager This class represents the simplest manager for generic DAO. It has only basic CRUD functions and again, there's no need to write new manager if you need just the CRUD functions.

Generic access manager This manager is used to access `BaseAccessObject` and it uses `GenericAccessDAO`. This is the type of manager by which the micro application should be accessed from outside.

Tests Tests are located in src/test in package `org.microapp.microappName`. There is a package `generic` containing `BaseDaoTestCase` which you should use for testing your DAOs - both generic and generic access.

1.3 Resources

Application context and persistence unit There are two application-context files. The first file is `applicationContext-microappName-jpa.xml`. There are several beans defined and quite a lot of renaming needs to be done here (as described in the Example part). Also, this is the file you need to import in application context of your application with user interface.

The second file is `applicationContext-microappName-resources.xml`. There is a data source bean and a property configurer bean defined. However, there are some weird problems when using the property configurer and a property file, so I will probably stop using this.

The persistence unit file is named `persistence-microappName.xml` and it contains only the definition of persistence unit.

JDBC properties The `jdbc-microappName.properties`serves as a properties file for database connection (jdbc and hibernate). This file will probably be deleted in next versions.

Sample data The `sample-data.xml` file is located in src/test/resources and it's used for inserting testing data to database. Format of sample data is the same as in AppFuse projects.

2 Example

In this little example, I will show you how to use the generic micro application to build a simple micro application which will manage member's cars.

I will be using Eclipse as an IDE.

2.1 Getting the code

Since the whole module is available on GitHub, all you need to do to get the code is to execute

```
git clone https://github.com/Cajova-Houba/microapp-generic.git
```

Now, you need to do some renaming. The renaming itself isn't mandatory, but if you want to use more than one micro application (which you probably want to) you should do it to separate micro applications properly.

First, change out the `pom.xml` file a bit so the header of the maven module looks like this.

```
<groupId>org.microapp</groupId>
<artifactId>microapp-cars</artifactId>
<version>1.0-SNAPSHOT</version>
<name>Cars microapplication</name>
```

When we are in the `pom.xml` file, we can also change some properties such as database settings or the location of the jdbc properties file (or we can do this as a last step after editing of other files is finished). You will need to change those properties:

- `$db.name`
- `$jdbc.username`
- `$jdbc.password`
- `$jdbc.properties.file`

Other properties can remain the same.

And import the code to Eclipse. File > Import > Existing maven Projects

2.2 Customizing

There are several files, spring beans and names which need to be renamed. Most of them contain *microappName* or *Generic*. Basically, every occasion of this needs to be replaced with the name of our micro application In this case it's *cars*.

2.3 Customizing - properties file

The first file that should be customized is `jdbc-microappName.properties`. Change the file name, so it looks like `jdbc-cars.properties` and fill in your database settings.

2.4 Customizing - persistence unit

The file is located in src/main/resources/META-INF and there's only one bean inside the file. The file is named `persistence-microappName.xml` by default. So, change the file name to `persistence-cars.xml` and the persistence unit name, so it's like

```
<persistence-unit name="ApplicationEntityManagerCars" ... >
</persistence-unit>
```

2.5 Customizing - application context

2.5.1 Resources xml file

Lets start with `applicationContext-microappName-resources.xml`. Again, change the filename so it's like `applicationContext-cars-resources.xml`. Then change the name of the property file in property configurer.

```
<value>classpath:jdbc-cars.properties</value>
```

And change the name of data source. As you can see, the data source loads it's properties from `jdbc-cars.properties` file. But you can also skip the whole property file thing and just write your database settings directly to the data source bean definition.

```
<bean id="dataSourceCars" ... >
...
</bean>
```

2.5.2 JPA xml file

Now, the `applicationContext-microappName-jpa.xml` file. Change the filename to `applicationContext-cars-jpa.xml`. You will need to import this file to the application context of your application.

There is a list of beans which names need to be changed. Those beans are entity manager factory, persistence unit manager and transaction manager.

Renamed beans should look like this

```
<!-- EntityManagerFactory -->
<bean id="entityManagerFactoryCars" ... >
<property name="persistenceUnitManager" ref="persistenceUnitManagerCars"/>
...
</bean>

<!-- Persistence unit manager -->
<bean id="persistenceUnitManagerCars" ... >
<property name="persistenceXmlLocations">
<list>
<value>classpath:META-INF/persistence-cars.xml</value>
</list>
</property>
<property name="defaultDataSource" ref="dataSourceCars"/>
</bean>

<!-- for @Transactional annotations -->
<tx:annotation-driven transaction-manager="transactionManagerCars"/>
```

```
<!-- Transaction manager -->
<bean id="transactionManagerCars" ... >
<property name="entityManagerFactory" ref="entityManagerFactoryCars" />
</bean>
```

Also, don't forget to update the import of the resources file at the beginning of this file.

```
<import resource="classpath:applicationContext-cars-resources.xml"/>
```

There are component scan tags at the end of the file. You can rename them now, or later, when you add your own DAOs and managers.

2.6 Customizing - generic classes

We have made some changes in our application context files and now we need to make sure those changes are update in the generic code.

2.6.1 Generic DAOs

There are two types of generic DAOs and we need to change the value of the *PERSISTENCE_UNIT_NAME* constant in `GenericDaoJpa`. The class is located in the package `org.microapp.microappName.generic.dao.jpa`. Just change the value of the mentioned constant from *ApplicationEntityManagerGeneric* to *ApplicationEntityManagerCars*

2.6.2 Generic managers

There is a `save()` method annotated with `@Transactional` annotation in `GenericManagerImpl`. You will need to change the annotation properties so it looks like this

```
@Transactional (
value = "transactionManagerCars",
propagation = Propagation.REQUIRES_NEW
)
```

The class is located in the package `org.microapp.microappName.generic.service.impl`.

2.6.3 Tests

Now we need to update the base test cases. Currently, there is only one test case class - `BaseDaoTestCase` in `org.microapp.microappName.generic.dao.jpa`. We need to change the application context file names in the `@ContextConfiguration` annotation so it looks like this

```
@ContextConfiguration(
locations={"classpath:/applicationContext-cars-resources.xml",
"classpath:/applicationContext-cars-jpa.xml"})
```

and we also need to change the value of the *ENTITY_MANAGER_FACTORY* constant and set it to `entityManagerFactoryCars`.

2.7 Customizing - package names

The package structure is still `org.microapp.microappName.*` and it's time to change that. The best way to replace the `microappName` with the real micro application name (`cars`) is to use the refactor function in Eclipse. You should end up with every class having package declaration similar to this

```
package org.microapp.cars.*;
```

You can now build your project with the `mvn install` command from the root directory. The build should be successful and if not, go through the error trace and make sure you did everything correctly.

If everything went fine, we can move on and create a simple micro application, test it and use it in our application.

2.8 Model classes

We will use a very simple scheme with two entities.

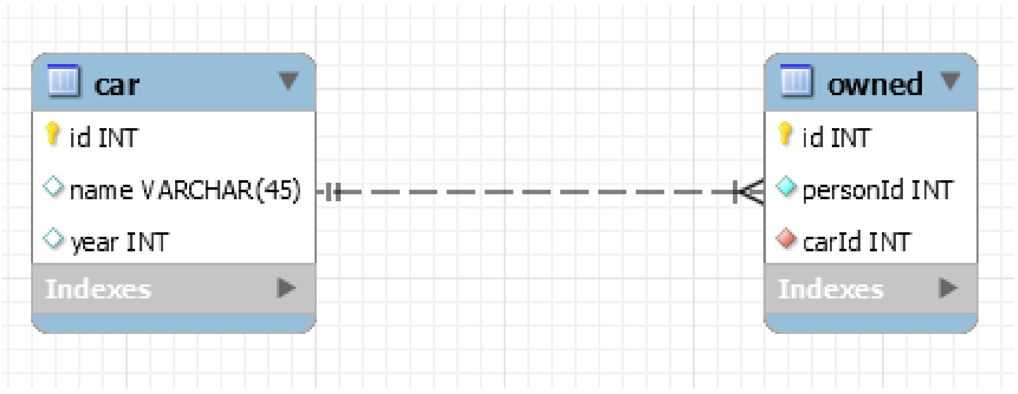

Figure 1: Database scheme

The *car* table will hold all available cars and the *owned* table will hold the information about ownership (which member owns which car(s)). The `personId` attribute makes a relationship to the id column of the membership table in the membernet core.

Create the *Car* class in `org.microapp.cars.model`. Don't forget that this class must extend the `BaseObject` class.

```
package org.microapp.cars.model;

@Entity
@Table(name="car")
public class Car extends BaseObject {

        @Column(length=50)
        private String name;

        private int year;

        // defining your own constructor isn't mandatory
        public Car (long id, String name, int year) {
        this.id = id;
        this.name = name;
        this.year = year;
        }

        //getters, setters

}
```

Create the *Owned* class in `org.microapp.cars.model`. The class will extend the `BaseAccessObject` class.

```
package org.microapp.cars.model;

@Entity
@Table(name="owned")
public class Owned extends BaseAccessObject {

        @ManyToOne(fetch=FetchType.EAGER)
        @JoinColumn(name="car_id")
        private Car car;

        // id and personId fields are inherited
        // from parent class

        // getters, setters

}
```

2.9 DAOs

We don't need to create new DAOs if all we need is just the CRUD functionality. However, just for example, I will show one DAO created as a new class and one DAO created as a bean in the spring context file.

CarDao Let's say we want to have a function which returns a list of cars made after a certain year. This isn't a standard CRUD function, so we will have to implement our own DAO.

At first, we will need to create a new interface - `CarDao`. This interface will extend the `GenericDao` interface and will contain definition of our specific function.

```
package org.microapp.cars.dao;

public interface CarDao extends GenericDao<Car, Long> {

        public List<Car> getCarsAfterYear(int year);
}
```

Even before we start implementing the `CarDao`, we will write a simple test case. We're also going to need some test data for our testing. Place this into your `sample-data.xml` file.

```
<table name="car">
        <column>id</column>
        <column>name</column>
        <column>year</column>
        <row>
                <value description="id">1</value>
                <value description="name">
                        Pontiac GTO
                </value>
                <value description="year">1967</value>
        </row>
        <row>
                <value description="id">2</value>
                <value description="name">
                        Plymouth Road Runner Hemi
                </value>
                <value description="year">1968</value>
        </row>
        <row>
                <value description="id">3</value>
                <value description="name">
                        Ford Mustang Boss 429
                </value>
                <value description="year">1969</value>
        </row>
        <row>
                <value description="id">4</value>
                <value description="name">
                    Chevrolet Camaro ZL1
                </value>
                <value description="year">1969</value>
        </row>
</table>
```

Now, we can create the test class and write our test case. Create a class `CarDaoTest` which will extend the `BaseDaoTestCase` class.

```
package org.microapp.cars.dao.jpa;

public class CarDaoJpaTest extends BaseDaoTestCase {

        @Autowired
        @Qualifier("carDaoJpa")
        private CarDao carDao;

        @Test
        public void testGetCarsAfterYear() {
        //there are two cars made after year 1968 in our
        //test data
        int year = 1968;
        List<Car> cars = carDao.getCarsAfterYear(year);
        assertFalse(cars.isEmpty());
        }
}
```

You can run the tests by `mvn test` command from the root directory. If you try to run your test now, you will get errors and your test won't pass, because we haven't implemented the `CarDaoJpa` class yet. We will do it right now.

```
package org.microapp.cars.dao.jpa;

@Repository("carDaoJpa")
public class CarDaoJpa extends GenericDaoJpa<Car, Long> implements
                                        CarDao {

        public CarDaoJpa() {
                super(Car.class);
        }
        public List<Car> getCarsAfterYear(int year) {
                // It depends on your preferences. I like using
                // the HQL/SQL queries

                String query = "SELECT c FROM Car c WHERE year > ?";
                Query q = getEntityManager().createQuery(query);
                q.setParameter(1, year);
                return q.getResultList();
        }

}
```

We have defined the `carDaoJpa` object and autowiring in our test class should work now. Make sure the scanning for DAOs is activated by this line in your application context file (`applicationContext-cars-jpa.xml`).

```
<context:component-scan base-package="org.microapp.cars.dao"/>
```

Try running the `mvn test` command now, everything should work and you should see `BUILD SUCCESS` message.

OwnedDao We will need just the CRUD functionality, so it's enough if we just create a bean named `ownedDaoJpa` in the `applicationContext-cars-jpa.xml` file.

```
<bean id="ownedDaoJpa"
      class="org.microapp.cars.generic.dao.jpa.GenericAccessDaoJpa">
        <constructor-arg value="org.microapp.cars.model.Owned" />
</bean>
```

And that's all. We can continue with writing managers now.

2.10 Managers

We created one DAO class and one DAO bean. So we also need to create one manager class and one manager bean (or class, but for CRUD functionality the bean is enough).

CarManager First, we need co create an interface which will extend the `GenericManager`. In this interface we will define our function for getting cars.

```
package org.microapp.cars.service;

public interface CarManager extends GenericManager<Car, Long> {

        public List<Car> getCarsAfterYear(int year);
}
```

I haven't created any generic testing classes for managers, so it will be without any testing this time and we continue to implementing this interface with `CarManagerImpl` class.

```
package org.microapp.cars.service.impl;

@Service("carManager")
public class CarManagerImpl extends GenericManagerImpl<Car, Long>
implements CarManager {

        private CarDao carDao;

        @Autowired
        // The Qualifier annotation isn't necessary,
        // but it makes things a little bit cleaner
        public CarManagerImpl(@Qualifier("carDaoJpa") CarDao carDao) {
                super(carDao);
                this.carDao = carDao;
        }

        public List<Car> getCarsAfterYear(int year) {
                return this.carDao.getCarsAfterYear(year);
        }
}
```

It's a very simple class. No annotations are needed on this method, but don't forget to add `@Transactional` annotation every time you write a method which inserts data into the database. See the `save()` method in one of generic managers.

OwnedManager Now, we can declare the `OwnedManagerImpl` bean. Add this to your `applicationContext-cars-jpa.xml` file.

```
<bean id="ownedManagerImpl"
      class="org.microapp.cars.generic.service.impl.GenericAccessManager">
        <constructor-arg value="ownedDaoJpa" />
</bean>
```

The last thing you need to do is activate scanning for your managers (for classes). You can do this with this line in your `applicationContext-cars-jpa.xml` file.

```
<context:component-scan base-package="org.microapp.cars.service" />
```

And that's it. You have created your first micro application You should be able to successfully build it with `mvn install` command.

2.11 Using the microapplication

To use the micro application you need to add dependency to the `pom.xml` of your maven project.

```
        <groupId>org.microapp</groupId>
        <artifactId>microapp-cars</artifactId>
        <version>1.0-SNAPSHOT</version>
```

Now, we need to import the spring context - for autowiring. Add this line to the application context file of your project.

```
<import resource="classpath:applicationContext-cars-jpa.xml"/>
```

Alternatively, you can also use the `web.xml` file.

```
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
                classpath:applicationContext-cars-jpa.xml
        </param-value>
</context-param>
```

If you use the `web.xml` file, don't forget to add a context loader listener.

```
<listener>
        <listener-class>
                org.springframework.web.context.ContextLoaderListener
        </listener-class>
</listener>
```