Generic micro application

v 1.0

Zdenek Vales

30th January 2016

1 About

Generic micro application is a multi-module maven project 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.

The application also have it's own simple UI, although this is meant only for testing the application.

2 Core module

2.1 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).

Generic classes and interfaces are stored in `org.microapp.microappName.generic`.

2.2 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.

2.3 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.

3 Membernet utils module

This simple module provides connection to membernet core (the big system). Currently it contains only one package `org.microapp.membernet` with `MembernetManager` inter-

face and `MembernetManagerImpl` class.

There are not many methods implemented yet, but this will change in the future.

3.1 Resources

To successfully wire managers from membernet core, you need to specify some properties such as mail properties and database connection details. There is a file `prop.properties` where you can specify those properties.

There is also the `applicationContext-membernet.xml` file in which the membernet context and mentioned properties file are placed. You can find beans specified for autowiring here too.

4 User interface module

Generic micro application uses Wicket for its user interface. Source files are located in package `org.microapp`. This package is more for testing purposes than for using in actual application so it's quite small.

4.1 Package structure

There are three packages in the main package. `ui`, `base` and `membernet`. In the `ui` package, there is a homepage and actual wicket application. In `base` package there is a `GenericPage` class which you can use when making new pages (or you can make your own, but this is easier). And in the `membernet` package you can see page based on `GenericPage` which lists all memberships in database (using membernet utils module).

5 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.

5.1 Getting archetype and generate new project

The easiest way to start developing is to download the archetype from

`https://dev.yoso.fi/git/jakub/membernet-micro-services-poc/tree/master/microapp-generic-archetype`

After you have downloaded the archetype, run `mvn install` command from the root directory of that archetype. This will install the archetype to your local maven repository and you will be able to generate new micro applications.

Now, generate the application with this command:

```
mvn archetype:generate \
-DarchetypeGroupId=org.microapp \
-DarchetypeArtifactId=microapp-generic-archetype \
-DarchetypeVersion=0.2 \
-DgroupId=org.microapp \
-DartifactId=microapp-cars \
-Dversion=1.5 \
-DmicroappName=Cars
```

Don't confuse the `microappName` property with `name` tag in `pom.xml`. It's not the same, the `microappName` serves mainly to create unique names for application context configuration files and beans.

You can now run `mvn install` command from the root directory of your newly generated project and if you have done everything correctly, you will see the `BUILD SUCCESS` message.

Run your application simply by `mvn jetty:run` command from the `ui` directory and you should be able to see it on `localhost:8080`.

Those were the first steps. Lets make some simple application now.

5.2 Model classes

We will use a very simple scheme with two entities. Almost everything will be done in the core module now.

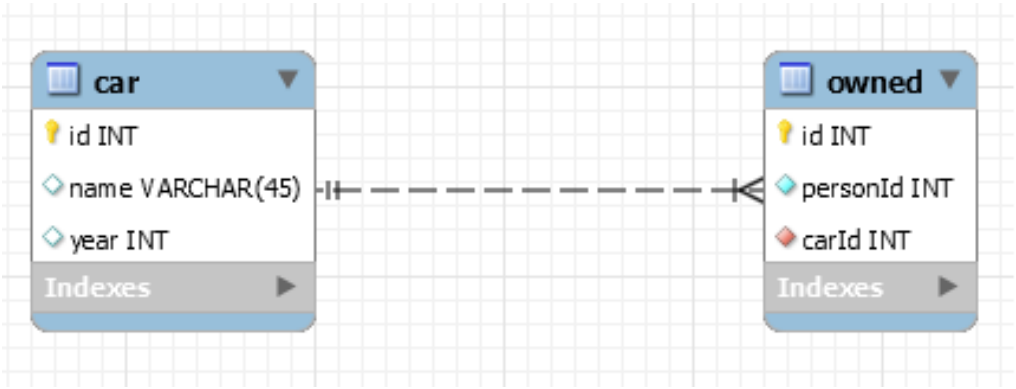

Figure 1: Database scheme

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

}
```

Since we're using the JPA instead of Hibernate, there is no need to map classes, use the `@Entity` annotation.

5.3 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 `CarDaoJpaTest` 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;

//in case you're not sure which
//import for Query to use, use this one
import javax.persistence.Query;

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();
        }

}
```

There are two ways of telling the spring about your new DAO. The annotation way and the XML way (which will be preferred in this tutorial). If you want to use the annotation way, just add this to you DAO class:

```
@Repository(carDaoJpa)
```

The XML way - add this to your `applicationContext-Cars-jpa.xml` file:

```
<bean id="carDaoJpa" class="org.microapp.Cars.dao.jpa.CarDaoJpa">
</bean>
```

We have the `carDaoJpa` bean defined and autowiring in our test class should work now.

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.

5.4 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);
}
```

Now, we will write a test class for our manager. Place the new class named `CarManagerImplTest` into the package `org.microapp.Cars.service.impl` in `src/test/java`. Right now, there aren't any generic classes for mock-testing, so you will need to create your DAOs first.

```
public class CarManagerImplTest extends BaseManagerTestCase {

        @Autowired
        private CarManager carManager;

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

}
```

And finally, we will write our implementation class - `CarManagerImpl`.

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

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 the `GenericManagerImpl` class.

Don't forget to create the `carManager` bean.

```
<bean id="carManager"
        class="org.microapp.Cars.service.impl.CarManagerImpl">
</bean>
```

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.GenericAccessManagerImpl">
        <constructor-arg value="ownedDaoJpa" />
</bean>
```

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

5.5 User interface

We will now build a simple webpage listing all cars in our database. For this purpose, we will use the Wicket-based UI module.

At first, add a dependency to the `pom.xml` in ui module.

```
<dependency>
        <groupId>org.microapp</groupId>
        <artifactId>microapp-cars-core</artifactId>
        <version>1.5</version>
</dependency>
```

And now, add a reference to the application context file, in the `webapp/WEB-INF/web.xml` file by adding this:

```
<!-- microapp core -->
classpath:applicationContext-Cars-jpa.xml
```

To the `<context-param>` tag.

CarsPage The `CarsPage` will be based on `GenericPage`. This class provides basic methods and design for displaying content. So, create two files - `CarsPage` class and `CarsPage.html` file in package `org.microapp.ui.cars`. Note that Apache Wicket needs both files to have the same names by default.

The basic page - without parameters will be enough for this example, so we need to autowire the `carManager` bean and override few methods.

```
@SpringBean
private CarManager carManager;
```

Note that when using Wicket-Spring combination you need to use `@SpringBean` annotation instead of `@Autowired`.

The first of methods we need to override is the `inic()` method:

```
@Override
public void inic() {
        super.inic();
        setTitle("Wicket - cars");
        setHeader("Cars");
}
```

Just like that. The super call is mandatory to do some basic initialization, the `setTitle()` method sets the html `<title>` tag and the `setHeader()` method sets the header of the page (`<h2>` tag).

The second method we will override is the `addComponents()` method:

```
@Override
public void addComponents() {
        super.addComponents();
        showCarsTable("carsTable");
}
```

Again, the super call is mandatory, because of some basic components being added into the page. The `showCarsTable()` method is responsible for creating and adding the table containing our cars into the page, it's described in the next step.

```
private void showCarsTable(String tableID) {

        //get the list of cars in database
        List<Car> cars = carManager.getAll();

        //create the dataview component
        final DataView dataView = new DataView(tableID,
                                new ListDataProvider(cars)) {

                //define how the item will be displayed
                @Override
                protected void populateItem(Item item) {
```

```
                        Car car = (Car)item.getModelObject();

                        //Label is the basic displaying
                        //component in Wicket
                        item.add(new Label("carId",car.getId()));
                        item.add(new Label("carName",car.getName()));
                        item.add(new Label("carYear",car.getYear()));
                }
        };

        //adds a dataview component to the page
        add(dataView);
}
```

As you can see from the comments, first, we create the `List` with all the cars in our database, then we crate a `DataView` component to represent our table and in the `populateItem()` method, we specify how one row in the table will look like.

The java-side of our page is almost done. We just need to add a link to the homepage pointing to our `CarsPage`.

Add this to the constructor in the `HomePage` class:

```
add(new Link("cars") {
        @Override
        public void onClick() {
                setResponsePage(CarsPage.class);
        }
});
```

This to the `HomePage.html`:

```
<a wicket:id="cars">Cars</a>
```

And now, we can move to the `CarsPage.html`. Since Wicket allows html pages to be extended, we will use this feature. There will be no `<body>`, `<head>` or `<html>` tags, those are already in `GenericPage.html`. The `CarsPage.html` will look just like this:

```
<wicket:extend>
        <table>
                <thead>
                        <tr>
                        <th>Car ID</th>
                        <th>Car's name</th>
                        <th>Year</th>
                        </tr>
                </thead>
                <tbody>
                        <tr wicket:id="carsTable">
```

```
                                    <td wicket:id="carId"></td>
                                    <td wicket:id="carName"></td>
                                    <td wicket:id="carYear"></td>
                                </tr>
                            </tbody>
                        </table>
</wicket:extend>
```

Run your application by `mvn jetty:run` from `ui/` directory and try it out on `localhost:8080`!

5.6 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-core</artifactId>
        <version>1.5</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>
```