Bound to Persist with Vaadin JPAContainer

Vaadin JPAContainer
2.0

Vaadin JPAContainer is...

Publication date of this manual version is 2011-11-16.

Published by:
Oy IT Mill Ltd
Ruukinkatu 2-4
20540 Turku
Finland

Table of Contents

1. Introduction

1.1. Overview

Most business web applications have a user interface that is used to either store data in or retrieve it from a database, or both. However, the data is rarely written directly from the user input to the database. Applications following the Model-View-Presenter (MVP) or -Controller (MVC) architecture pattern use an intermediate business or *domain model* implemented as Java classes. The purpose of *Java Persistence API* is to define how the Java classes are mapped to database tables. Using the mappings, the instances of such classes can be *persisted* – stored in and retrieved from the database – without writing complex queries and copying the values between the queries and the domain model.

Vaadin JPAContainer allows of connecting Vaadin user interface components directly to the persistent objects. It is an implementation of the container interface defined in the Vaadin core framework. You can use the container to display data in a table, tree, any other selection component, or edit the data in a form.

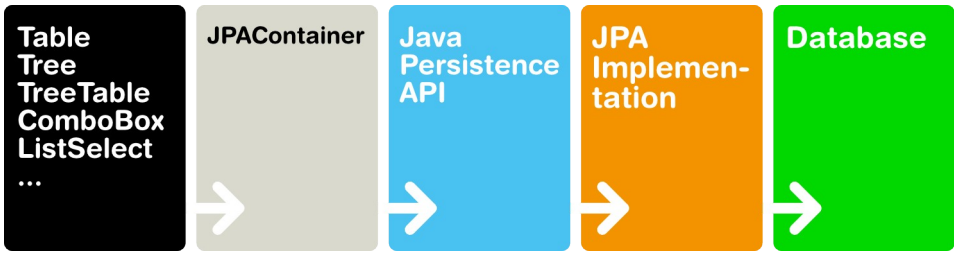

Figure 1: Role of JPAContainer in binding Vaadin components to a database

In this introduction, we go through the basic concepts of Vaadin data binding, Java Persistence API, JPA implementations, relational databases, and the role of JPAContainer to bind them. Then, we are ready to look into installing Vaadin JPAContainer, defining a domain model, and using JPAContainer.

1.2. Technologies Covered in the Tutorial

The basic purpose of this tutorial is to demonstrate the use of the following technologies:

- Vaadin Framework
- Vaadin JPAContainer
- Java Persistence API
- Java Bean Validation API

We use the following additional tools:

- EclipseLink
- Vaadin Bean Validation add-on

- Hibernate Validator
- H2 Database Engine
- CustomField add-on

Any implementation of the Java Persistence API could be used, but in this tutorial we use EclipseLink, which is the reference implementation of JPA 2.0. Its choice is only relevant in the project configuration.

The Vaadin Bean Validation add-on allows validation of Vaadin forms using the Java Bean Validation API 1.0 (JSR-303). It is based on annotations for the bean properties. We use the Hibernate Validator as the chosen implementation of the API.

For the back-end persistence database, we use an in-memory database using the H2 database engine.

The CustomField add-on allows creating composite field components much the same way as the standard CustomComponent in Vaadin core framework.

1.3. Vaadin Data Model

Let us first recapture the Vaadin data model introduced in the Chapter 9: *Binding Components to Data* in Book of Vaadin. It consists of three levels of containment: *properties*, *items*, and *containers*.

Properties

Atomic data is represented as **Property** objects that have a *value* and a *type*. Properties can be bound to **Field** components, such as a **TextField**, or a table cell.

```
// A property of some type
ObjectProperty<String> p =
    new ObjectProperty<String>(“the property value”);

// A field
TextField field = new TextField(“Name”);

// Bind the field to the property
field.setPropertyDataSource(p);
```

The **ObjectProperty** used above is just one implementation of the **Property** interface – in **JPAContainer** we use a different property type.

Items

An **Item** is a collection of properties that is typically bound to a **Form** or a row in a **Table** or some other selection component. Properties of an item are identified by a *property identifier* (PID).

```
// Create an item with two properties
PropertysetItem item = new PropertysetItem();
item.addItemProperty("name",
```

```
                        new ObjectProperty<String>("Zaphod"));
item.addItemProperty("age",
                        new ObjectProperty<Integer>(42));

// Bind it to a form
Form form = new Form();
form.setItemDataSource(item);
```

Containers

A **Container** is a collection of items, usually bound to a **Table**, **Tree**, or some other selection component. Items in a container are identified by an item identifier (IID). Normally, all the items in a container have the same type and the same properties.

For a more detailed description of the Vaadin data model, please refer to.

Normal (Non-Persistent) Binding to JavaBeans

Vaadin core library provides **BeanItem** implementation of the **Item** interface to bind bean objects. At the **Container** level, Vaadin provides the **BeanItemContainer** and **BeanContainer** implementations. They are very useful for handling transient (non-persistent) objects.

```
// Here is a bean
public class Bean implements Serializable {
    String name;
    double energy; // Energy content in kJ/100g

    public Bean(String name, double energy) {
        this.name   = name;
        this.energy = energy;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getEnergy() {
        return energy;
    }

    public void setEnergy(double energy) {
        this.energy = energy;
    }
}

...
// Create a container for such beans
BeanItemContainer<Bean> beans =
    new BeanItemContainer<Bean>(Bean.class);
```

```
// Add some beans to it
beans.addBean(new Bean("Mung bean",   1452.0));
beans.addBean(new Bean("Chickpea",    686.0));
beans.addBean(new Bean("Lentil",      1477.0));
beans.addBean(new Bean("Common bean", 129.0));
beans.addBean(new Bean("Soybean",     1866.0));

// Bind a table to it
Table table = new Table("Beans of All Sorts", beans);
layout.addComponent(table);
```

Actually, the classes added to a **BeanItemContainer** do not need to be beans; only the setters and getters are relevant, and it is not necessary to implement **Serializable** and have a default constructor.

So, that should be simple. Next, we see how to persist Java classes.

1.4. Java Persistence API

Java Persistence API (JPA) is an API for mapping and storing Java objects to a relational database. In JPA and entity-relationship modeling generally, a Java class or POJO is considered an *entity*. Class (or entity) instances correspond with a row in a database table and member variables of a class with columns. Entities can also have relationships with other entities.

JPA specifies Java annotations that provide metadata about the entities and their relationships. For example, if we look at the **Person** class in the *JPAContainer AddressBook Demo*, we define various database-related metadata for member variables of a class:

```
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @NotNull
    @Size(min = 2, max = 24)
    private String firstName;

    @Size(min = 2, max = 24)
    private String lastName;

    @NotNull
    private Department department;
    …
```

The JPA implementation uses reflection to read the annotations and defines a database model automatically from the class definitions.

JPA annotation is discussed in detail later in Section 3.3: *Annotating the Classes for Persistence*.

1.5. JPA Implementations

1.6. Rapid Start with Spring Roo

Perhaps the quickest way to create a domain model and persist it with JPAContainer is to use Spring Roo, a rapid development tool for Java applications. It generates code that uses the Spring Framework, Java Persistence API, and Apache Maven. It also allows extending its functionality using add-ons, such as the Vaadin Plugin for Spring Roo. The Vaadin add-on can generate a user interface views based on the data model definitions given to Roo.

For a Spring Roo tutorial with Vaadin, please refer to Chapter 12: *Rapid Development Using Vaadin and Roo* in Book of Vaadin.

2. Installing Vaadin JPAContainer

This step gives basic instructions for installing Vaadin JPAContainer.

2.1. Downloading from Vaadin Directory

2.2. Defining a Dependency in Maven

Detailed instructions for using Vaadin Add-ons in Maven projects are given in Book of Vaadin Chapter 13: *Using Vaadin Add-ons*, in the Section *Using Add-ons in a Maven Project*.

3. Defining a Domain Model

3.1. A Domain Model

In this tutorial, we use a simple domain model with two entities: a **Person** and a **Department**.

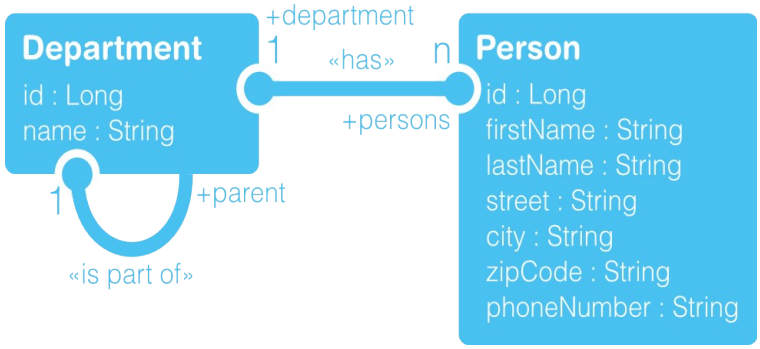

Figure 2: Domain Model

A **Department** has an integer (Long) identifier and a name. Departments have a hierarchy, which is represented by a `parent` reference. Each department has *n* persons working in it.

A **Person** has also an integer (Long) identifier and a first and last names, as well as basic contact information. A person works in a department.

3.2. Defining the Classes

The classes are defined in the `com.vaadin.demo.jpaaddressbook.domain` package of the demo project.

Department Class

The **Department** class is defined from the domain model in as follows:

```
public class Department {
    private Long id;
    private String name;
    private Set<Person> persons;
    private Boolean superDepartment;
    private Department parent;

    … setters and getters …
}
```

Notice that we added a *superDepartment* member that was not in the domain model. It is a shorthand variable that is only used internally for optimization to mark departments that only have sub-department and no personnel.

```
public boolean isSuperDepartment() {
    if (superDepartment == null) {
        superDepartment = getPersons().size() == 0;
    }
    return superDepartment;
}
```

In addition, we define a *hierarchicalName* property that only has a getter and is not associated with a member variable. Its value is the hierarchy path calculated dynamically using the *toString()* method.

```
public String getHierarchicalName() {
    if (parent != null) {
        return parent.toString() + " : " + name;
    }
    return name;
}

public String toString() {
    return getHierarchicalName();
}
```

Person Class

The **Person** class is defined from the domain model in as follows:

```
public class Person {
    private Long id;
    private String firstName;
    private String lastName;
    private String street;
    private String city;
    private String zipCode;
    private String phoneNumber;
    private Department department;

    … setters and getters …
}
```

3.3. Annotating the Classes for Persistence

Let us look at the basic JPA metadata annotations as they are used in the **Department** class of the tutorial project. Please refer to JPA reference documentation for a full list of possible annotations.

3.3.1. @Entity

Each class that is enabled as a persistent entity must have the `@Entity` annotation.

```
@Entity
public class Department {
```

3.3.2. @Id

Entities must have an identifier that is used as the primary key for the table. It is used for various purposes in database queries, most commonly for joining tables.

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

The identifier is usually generated automatically in the database. The strategy for generating the identifier is defined with the `@GeneratedValue` annotation.

3.3.3. @OneToMany

As noted earlier, entities can have relationships. The **Department** entity of the domain model has one-to-many relationship with the **Person** entity ("Department has persons"). This relationship is represented with the `@OneToMany` annotation.

```
@OneToMany(mappedBy = "department")
private Set<Person> persons;
```

3.3.4. @ManyToOne

Many departments can belong to the same higher-level department, represented with `@ManyToOne` annotation.

```
@ManyToOne
private Department parent;
```

3.3.5. @Transient

Properties that are not persisted are marked as transient with the `@Transient` annotation.

```
@Transient
private Boolean superDepartment;
…
@Transient
public String getHierarchicalName() {
…
```

3.3.6. @NotNull

In the **Person** class, we define that the *department* and *firstName* properties as not null. Trying to persist an entity with null value in such a property results in an exception.

```
@NotNull
@Size(min = 2, max = 24)
private String firstName;

@NotNull
private Department department;
```

4. Creating the Main View

TBD