Bound to Persist with Vaadin JPAContainer

Vaadin JPAContainer 2.0

Vaadin JPAContainer is a Vaadin Container implementation to bind Vaadin user interface components directly to Java objects persisted using the Java Persistence API.

Publication date of this manual version is 2011-11-18.

Published by:
Vaadin 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 that follow a multilayered architecture have an intermediate business logic or *domain layer*, which is implemented as Java classes.

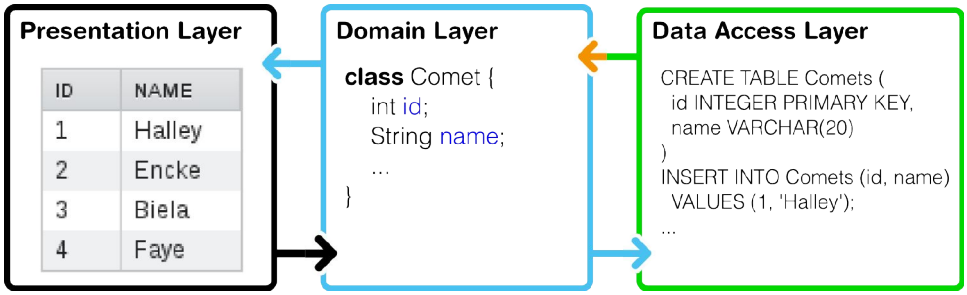

Figure 1: Three-Layer Architecture

The data binding model in Vaadin can be used to bind user interface components to a domain layer. The purpose of *Java Persistence API* is to define how the Java classes of the domain layer 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 connecting Vaadin user interface components directly to the persistent model 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.

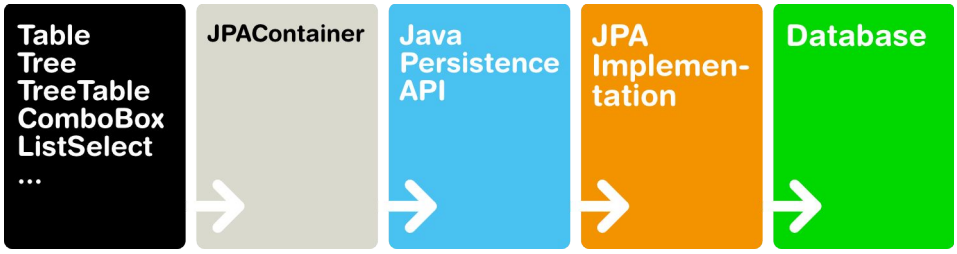

Figure 2: Role of JPAContainer in binding Vaadin components to a database

In this tutorial, we revisit the Vaadin Tutorial presented in vaadin.com/tutorial and add persistence to the application using JPAContainer.

But first, we start by going 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 or importing the JPAContainer AddressBook Demo, defining a domain model, and using the JPAContainer in an application.

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 object-relational mapping (ORM) of Java objects to a relational database. In JPA and entity-relationship modeling in general, a Java class 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 4.3: *Annotating the Classes for Persistence*.

1.5. JPA Implementations

TBD

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 chapter gives basic instructions for installing Vaadin JPAContainer in your own application. If you just wish to go on with the tutorial and install JPAContainer later, you can skip to Chapter 3: *JPAContainer AddressBook Demo*.

2.1. Dual Licensing Under AGPL and CVAL

Vaadin JPAContainer is available under two licenses: *Affero General Public License* (AGPL) and *Commercial Vaadin Add-on License* (CVAL). If your project is compatible with the open-source AGPL, you can use the add-on for free. Otherwise you must acquire a sufficient number of CVAL licenses before the 30-day trial period ends. Vaadin JPAContainer is distributed as a separate installation package for each license.

Use of Vaadin JPAContainer with the CVAL license is included in the Vaadin PRO subscription.

Free for Pro

2.2. Downloading JPAContainer from Vaadin Directory

JPAContainer is available for immediate download from the Vaadin Directory at http://vaadin.com/directory#addon/vaadin-jpacontainer.

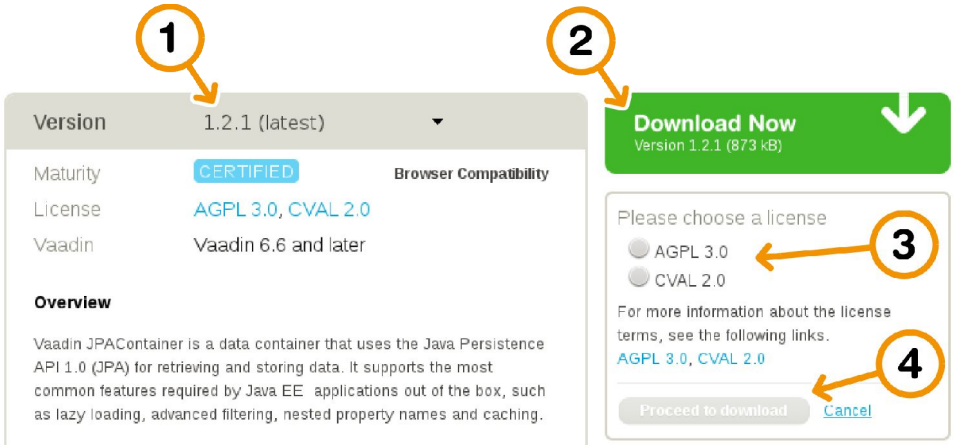

Figure 3: Downloading JPAContainer from Vaadin Directory

1. Select **Version** (the latest version is selected by default)
2. Click **Download Now**
3. Select either **AGPL** or **CVAL** license
4. Click **Proceed to download**
5. After downloading the installation package, extract its contents to a local folder using an appropriate ZIP extraction software for your operating system

2.3. Installation Package Content

Once extracted to a local folder, the contents of the installation directory are illustrated in Figure 4.

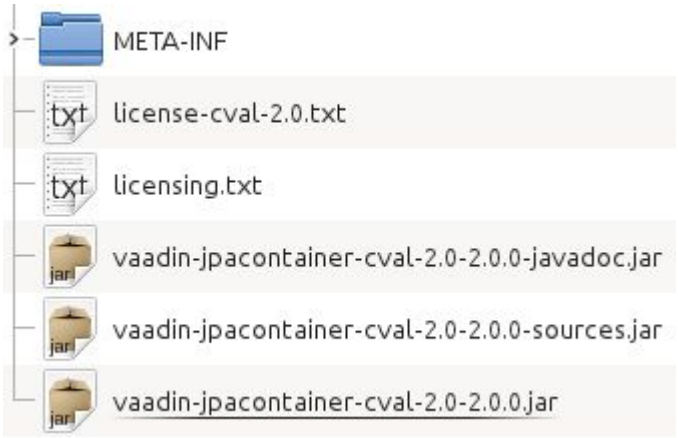

Figure 4: Installation Package Contents

The files in the installation package are as follows:

licensing.txt

General information about licensing.

license-xxxx-y.y.txt

The full license text for the library.

vaadin-jpacontainer-xxxx-y.y-z.z.z.jar

The actual Vaadin JPAContainer library. The `xxxx` is the license name and `y.y` its version number. The final `z.z.z` is the version number of the Vaadin JPAContainer.

vaadin-jpacontainer-xxxx-y.y-z.z.z-javadoc.jar

JavaDoc documentation JAR for the library. You can use it for example in Eclipse by associating the JavaDoc JAR with the JPAContainer JAR in the build path settings of your project.

vaadin-jpacontainer-xxxx-y.y-z.z.z-sources.jar

Full source code for the library. It may prove useful during development, for example when debugging, if you need to look inside the code. You can use it in Eclipse by associating the JavaDoc JAR with the JPAContainer JAR in the build path settings of your project.

2.4. 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. JPAContainer AddressBook Demo

In this JPAContainer tutorial, we look how to add persistence to the Vaadin Tutorial application presented in vaadin.com/tutorial.

In this tutorial, we use Maven, but you can set up the project just as well as a regular Eclipse project.

3.1. Importing the Tutorial Project

Before proceeding further, you should import the demo project to your favorite IDE...well, let's say that is Eclipse. As the project is Maven-based, Eclipse users need to install the m2e (or m2eclipse for older versions) plugin to be able to import Maven projects.

3.1.1. Checking Out from Repository (optional)

If you wish to use the latest version of the demo or do not have the Vaadin JPAContainer installation package at hand, you can check out the demo from the repository. If you do not wish to install Subclipse and the Subclipse SCM connector in Eclipse, which can cause some trouble sometimes, you can do the check-out from the command-line as follows:

1. Install Subversion command-line tools or use some alternative Subversion client
2. Run the following command in a terminal window or use some other Subversion client to do the checkout:

```
$ svn co http://dev.vaadin.com/svn/addons/JPAContain
er/trunk/jpacontainer-addressbook-demo
```

3.1.2. Importing from Existing Installation Folder

The demo project is included in the Vaadin JPAContainer installation package. Once extracted to a local folder or checked out from the repository as described above, you can import it in Eclipse as follows:

1. Install m2e in Eclipse if you have not done so already
2. Select File → Import in Eclipse
3. Select **Maven → Existing Maven Projects**, and click **Next**.
4. Click **Browse** to select the `jpacontainer-addressbook-demo` folder where you checked out the project.
5. Click **Finish**.

3.1.3. Import from Repository Using Subclipse

If you have installed Subclipse in Eclipse, you should be able to import the demo project a bit easier as follows:

1. Install m2e in Eclipse if you have not done so already
2. Select File → Import
3. Select **Maven → Check out Maven Project from SCM**, and click **Next**.
4. In **SCM URL**, select **svn** and enter URL http://dev.vaadin.com/svn/addons/JPAContainer/trunk/jpacontainer-addressbook-demo
5. Click **Finish**.

3.2. Browsing the Application

Importing the project takes a while. After it is finished, please take some time to inspect the project structure and files in Eclipse. The **Project Explorer** in Eclipse organizes Java source files in source folders, as illustrated in Figure 5.

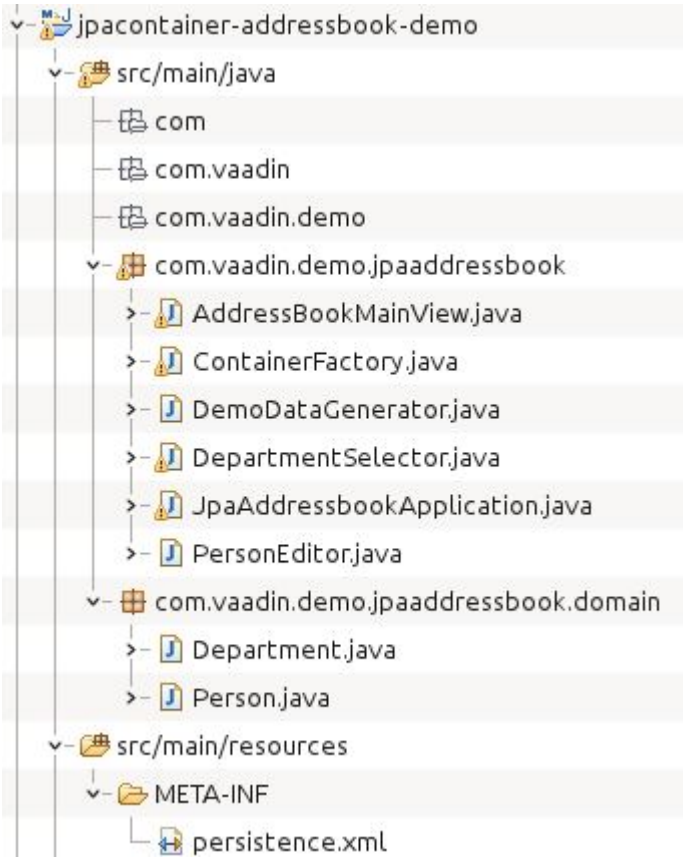

Figure 5: Browsing the project contents with Project Explorer in Eclipse.

3.3. Running the Application

If you imported the project as a Maven project, you can launch it in a server as follows:

1. Stop any web server you may have running in port 8080.
2. Right-click the project folder in Project Explorer and select **Run As → Maven Build...**.
3. In the **Edit Configuration** window that opens, enter "`package jetty:run`" in the **Goals** field and click **Run**. Wait for a while as the light-weight Jetty web server is launched.
4. Open a web browser with URL http://localhost:8080/.

The AddressBook Demo application should work as shown in Figure 6.

Figure 6: JPAContainer AddressBook Demo.

3.4. Browsing the Sources Online

You can also browse the source code of the demo project on-line at http://dev.vaadin.com/browser/svn/addons/JPAContainer/trunk/jpacontainer-addressbook-demo.

4. Defining a Domain Model

In this chapter, we look into defining a domain model, implementing it as Java classes, and then enabling persistence by annotating the classes with JPA annotations.

4.1. A Domain Model

The AddressBook has a simple domain model with two entities: a **Person** and a **Department**.

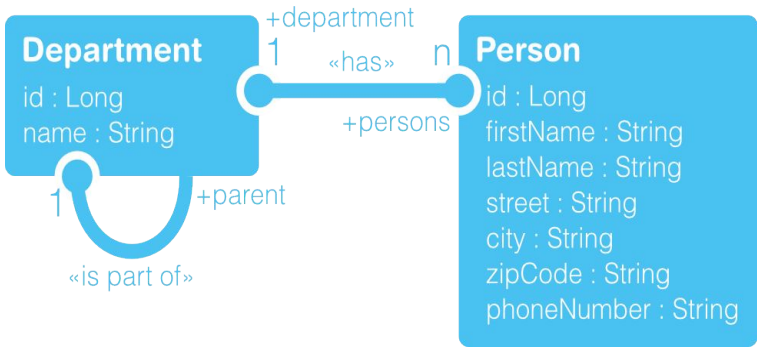

Figure 7: Domain Model

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.

4.2. 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 …
}
```

4.3. 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.

4.3.1. @Entity

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

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

4.3.2. @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.

4.3.3. @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;
```

4.3.4. @ManyToOne

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

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

4.3.5. @Transient

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

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

```
…
```

4.3.6. @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;
```

5. Application Architecture

This chapter describes the top-level user interface architecture of the JPAContainer AddressBook application.

5.1. Overview

TBD

5.2. Application Class

The **JpaAddressbookApplication** is the main class of the application.

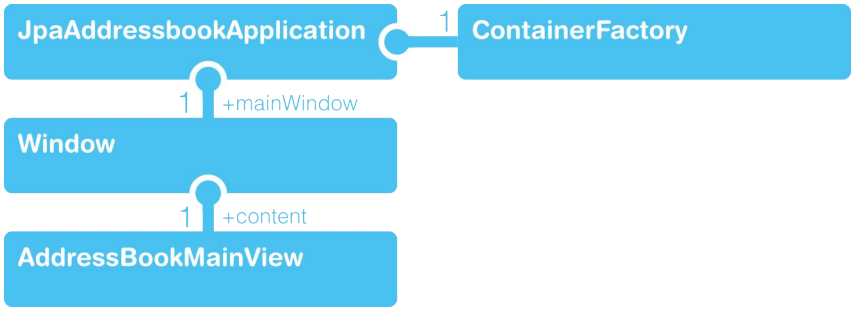

Figure 8: Address Book Application Class

The main task of the application class is to create the user interface in the *init()* method, but it also handles some other tasks:

- Generates example data in the database
- Maintains a **ContainerFactory**, which provides access to persistent containers and can be easily accessed globally
- Uses ThreadLocal pattern to allow global access to application data, most importantly the container factory

The application class begins as follows:

```
public class JpaAddressbookApplication extends Application
implements HttpServletRequestListener {

    static {
        DemoDataGenerator.create();
    }

    private static ThreadLocal<JpaAddressbookApplication>
        threadLocalApplication =
        new ThreadLocal<JpaAddressbookApplication>();

    private ContainerFactory containerFactory;

    public JpaAddressbookApplication() {
        containerFactory = new ContainerFactory();
```

```
        threadLocalApplication.set(this);
    }
```

The *init()* method of the application is trivial as the application only has one view which is defined in the AddressBookMainView class:

```
    @Override
    public void init() {
        Window window = new Window();
        setMainWindow(window);
        window.setContent(new AddressBookMainView());
    }
```

The rest of the application class is a getter for the container factory and handling the ThreadLocal Pattern as described next.

Global Access to Application Data Using the ThreadLocal Pattern

The JPAContainer AddressBook demo includes an implementation of the *ThreadLocal Pattern* to make it easier to access data global in a user session, such as [TBD]. The *getApplication()*, *getWindow()*, and *getLocale()* methods available in all Vaadin components can be used to access such data, but they do not work before the components are attached, most notably in the constructors. Using static variables for session-global data does not work, because they are shared by all users of the servlet. You can get a thread-local instance of the application object with the static *get()* method defined in AddressBookApplication.

The ThreadLocal Pattern is explained in detail in the Section: *ThreadLocal Pattern* in the Book of Vaadin.

5.3. AddressBookMainView

The main view of the application is realized in the **AddressBookMainView** class. It extends from **HorizontalSplitPanel** to allow user to resize the content.

On the left side of the split panel, we have a **Tree** which lets the user to browse and select a department that is used to filter the table on the right.

On the right side, we have a **Table** which shows all the persons in the selected department, and if no department is selected, all persons.

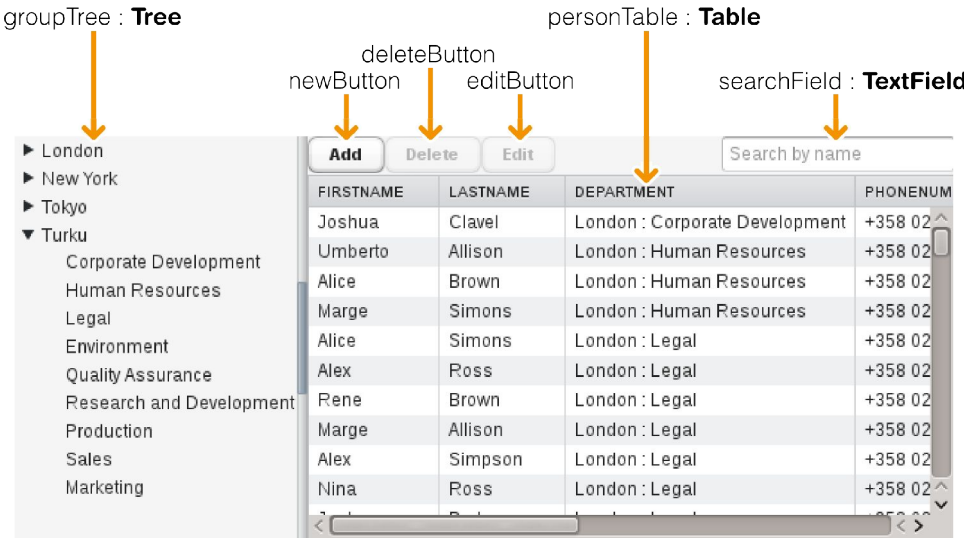

Figure 9: AddreddBookMainView

The underlying class relationships are shown in Figure 10.

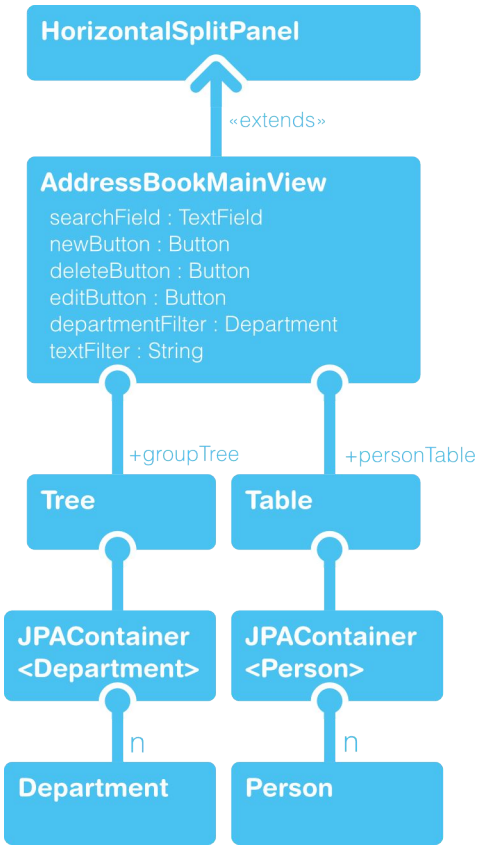

Figure 10: AddressBookMainView class diagram

Both the **Tree** and the **Table** are bound to a **JPAContainer** holding **Department** and **Person** objects, respectively.

5.4. Managing JPAContainers with ContainerFactory

TBD

6. Creating, Reading, Updating, and Deleting

This chapter focuses on how the **AddressBookMainView** implements CRUD – creation, reading, updating, and deletion of data objects in the domain model. The basics of these tasks were covered in the Vaadin Tutorial (notice that the code is not exactly the same), so we concentrate on the parts relevant to using JPAContainer.

6.1. Building the CRUD Area

The right side of the main view contains a CRUD area where you can add items in the **Person** table by clicking the **Add** button, delete them with the **Delete** button, and edit the selected person with the **Edit** button.

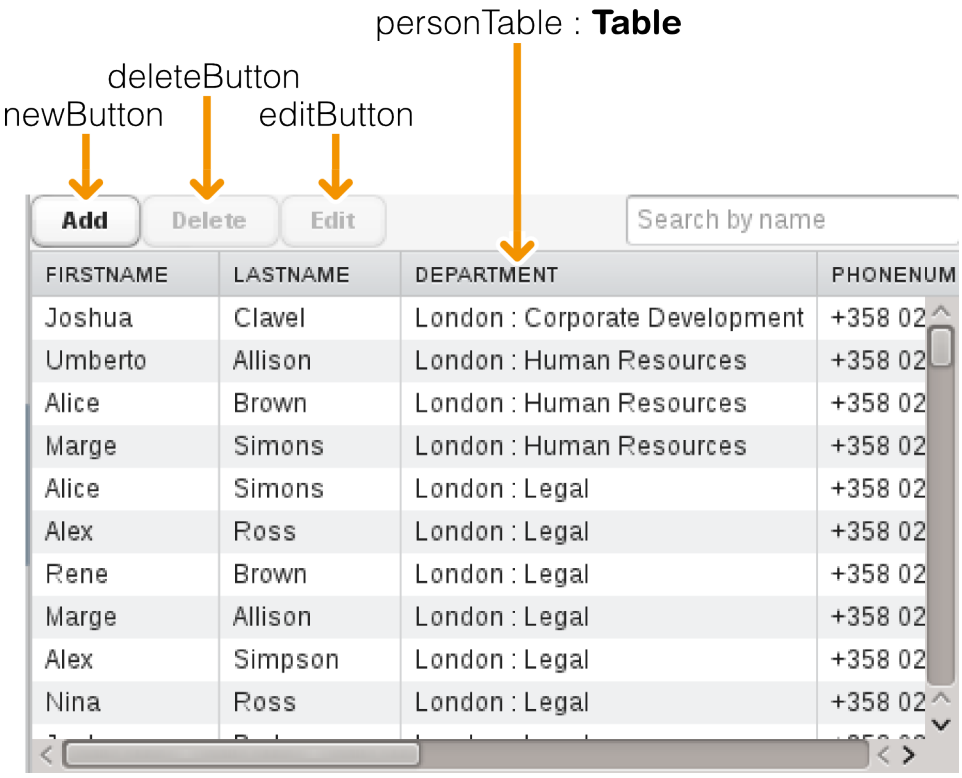

Figure 11: The CRUD area

The user interface controls for the CRUD as well as the **JPAContainer** references are defined as member variables:

```
public class AddressBookMainView extends
HorizontalSplitPanel implements ComponentContainer {
    private Tree groupTree;
    private Table personTable;

    private TextField searchField;
```

```
    private Button newButton;
    private Button deleteButton;
    private Button editButton;

    private JPAContainer<Department> departments;
    private JPAContainer<Person> persons;

    private Department departmentFilter;
    private String textFilter;
```

The `departments` and `persons` members are merely shorthand references to the containers managed by the **ContainerFactory**, as explained next.

6.1.1. Getting and Binding the JPAContainers

The constructor accesses the **ContainerFactory** held in the application object to request the containers for the **Department** and **Person** objects. References to the containers are kept for shorthand in the member variables.

```
public AddressBookMainView() {
    ContainerFactory cf =
        JpaAddressbookApplication.getInstance()
            .getContainerFactory();
    departments = cf.getDepartmentReadOnlyContainer();
    persons = cf.getPersonContainer();

    buildTree();
    buildMainArea();

    setSplitPosition(30);
}
```

Binding the **Person** table to the **JPAContainer** is done simply in the constructor of the **Table** just as with any Vaadin container:

```
    private void buildMainArea() {
        …
        personTable = new Table(null, persons);
        …
```

6.2. Creating an Item

New **Person** items are created by clicking the **Add** button. The button is created in the *buildMainArea()* method as follows:

```
newButton = new Button("Add");
newButton.addListener(new Button.ClickListener() {
```

When the button is clicked, a new **Person** object is created and wrapped in a **BeanItem** to allow binding it to a **Form**.

```
    @Override
    public void buttonClick(ClickEvent event) {
```

```
        final BeanItem<Person> newPersonItem =
                new BeanItem<Person>(new Person());
```

Editing is done in a **PersonEditor** window documented in Section 6.3: *Editing an Item in a PersonEditor*. The editor class defines an **EditorSavedListener** interface for handling **EditorSavedEvent**s fired when the user clicks "**Save**" in the editor.

```
        PersonEditor personEditor =
                new PersonEditor(newPersonItem);

        personEditor.addListener(new EditorSavedListener() {
```

The PersonEditor is described in more detail in Section 6.3: *Editing an Item in a PersonEditor*.

Adding an Entity to the JPAContainer

When saved, the edited new item is taken out of the **BeanItem** wrapper and added to the **JPAContainer<Person>** with the *addEntity()* method. The transaction needs to be committed to the database by calling *commit()* on the container.

```
            @Override
            public void editorSaved(EditorSavedEvent event) {
                persons.addEntity(newPersonItem.getBean());
                persons.commit();
            }
        });
        …
```

Once written in the container, the **Table** bound to the container is automatically updated to display the new item (notice that it is added to the end).

6.3. Editing an Item in a PersonEditor

The **Edit** button is disabled by default and enabled when an item is selected in the table. When clicked, the **PersonEditor** is opened for editing the currently selected item:

```
editButton = new Button("Edit");
editButton.addListener(new Button.ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        getApplication().getMainWindow()
          .addWindow(new PersonEditor(
             personTable.getItem(personTable.getValue())));
    }
});
```

PersonEditor

The **PersonEditor** is a sub-window that contains a **Form** for editing a **Person** item.

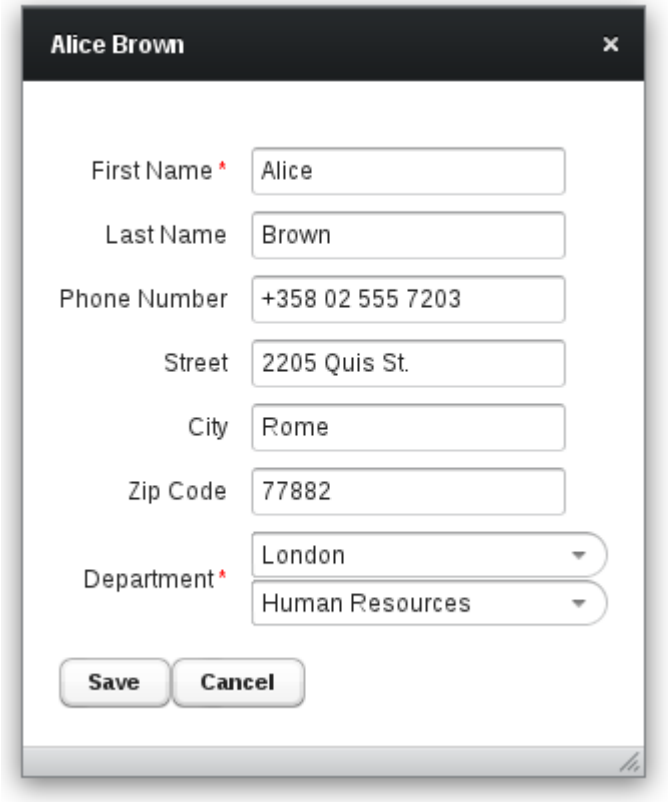

Figure 12: The PersonEditor

The **PersonEditor** is a sub-window that extends the **Window**. It also handles button clicks and creates the field components for the form.

```
public class PersonEditor extends Window
        implements Button.ClickListener, FormFieldFactory {
    private final Item personItem;
    private Form editorForm;
    private Button saveButton;
    private Button cancelButton;
```

The item to edit is given to the constructor as a parameter. When editing an existing item with the **Edit** button, the item is an **EntityItem<Person>**. When editing a new item added with the **Add** button, the item is a temporary **BeanItem<Person>** wrapper, as the person is not yet added to the **JPAContainer**.

```
    public PersonEditor(Item personItem) {
        this.personItem = personItem;
```

We us a **BeanValidationForm** from the Vaadin BeanValidation add-on to edit the item. We enable buffering by calling *setWriteThrough(false)*.

```
        editorForm =
```

```
            new BeanValidationForm<Person>(Person.class);
        editorForm.setFormFieldFactory(this);
        editorForm.setWriteThrough(false);
        editorForm.setImmediate(true);
```

And finally, we bind the person item to the form:

```
        editorForm.setItemDataSource(personItem,
            Arrays.asList("firstName", "lastName",
            "phoneNumber", "street", "city", "zipCode",
            "department"));
        …
```

Saving the Form

When the **Save** button is clicked, the buffered form is written to the underlying data model by calling *commit()*. In case of editing an existing item, the data is written directly to the **JPAContainer<Person>** and to the database. If editing a new item added with the **Add** button, the data is written to the **BeanItem<Person>**.

```
@Override
public void buttonClick(ClickEvent event) {
    if (event.getButton() == saveButton) {
        editorForm.commit();
```

Then an **EditorSavedEvent** is fired, to be received in an **EditorSavedListener**, as described in Section 6.2: *Creating an Item*. This allows the **Add** button mechanism to move the new **Person** object from the **BeanItem<Person>** to the **JPAContainer<Person>**.

```
        fireEvent(new EditorSavedEvent(this, personItem));
    } else if (event.getButton() == cancelButton) {
```

Otherwise, just discard the form. Then, close the sub-window.

```
        editorForm.discard();
    }
    close();
}
```

6.4. Deleting an Item

Deleting an item is done simply with the *removeItem()* method.

```
deleteButton = new Button("Delete");
deleteButton.addListener(new Button.ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        persons.removeItem(personTable.getValue());
    }
});
deleteButton.setEnabled(false);
```

7. Presenting Hierarchical Data

This chapter describes how to bind **JPAContainer** to a hierarchical component, such as a **Tree** or **TreeTable**.

7.1. JPAContainer is Hierarchical

JPAContainer implements the **HierarchicalEntityContainer** interface, which extends the **Container.Hierarchical** interface that defines hierarchical representation of data in a **Container**. You can bind any such container to a **Tree** or a **TreeTable**.

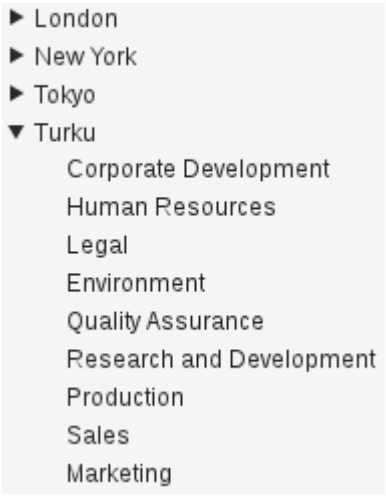

Figure 13: Tree Bound to a Hierarchical JPAContainer<Department>

The **HierarchicalEntityContainer** requires that the hierarchy is represented by parent references in the entities. Recall from the Chapter 4: *Defining a Domain Model* that the **Department** entity contains an *«is part of»* relationship with itself, represented with a *parent* property. The relationship is shown in Figure 14.

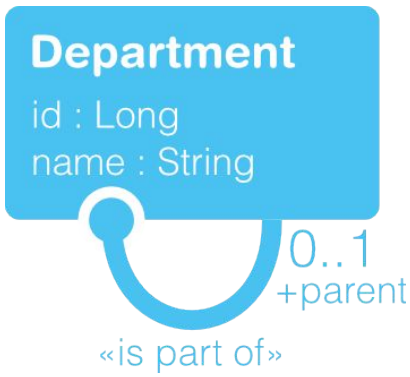

Figure 14: Hierarchy Representation in the Domain Model

7.1.1. Defining Hierarchy Relationship

The hierarchy relationships for the **JPAContainer<Department>** are defined in the **ContainerFactory** class, in the *getDepartmentReadOnlyContainer()* method.

Defining the areChildrenAllowed() Property

The factory defines a couple of things. First of all, it extends the **JPAContainer** class to customize the logic for when children are allowed (and the expansion marker is shown). We want to have only the non-leaf items expandable. Children are therefore allowed only for the root items, that is, the "super departments" having the `superDepartment` property as `true`.

```
public JPAContainer<Department>
        getDepartmentReadOnlyContainer() {
    JPAContainer<Department> departmentContainer =
        new JPAContainer<Department>(Department.class) {
        @Override
        public boolean areChildrenAllowed(Object itemId) {
            return super.areChildrenAllowed(itemId) &&
                   getItem(itemId).getEntity()
                       .isSuperDepartment();
    }
};
```

The *areChildrenAllowed()* in the **JPAContainer** superclass should be called to make basic checks.

The entity provider must be set for the container as follows:

```
    departmentContainer.setEntityProvider(
        new CachingLocalEntityProvider<Department>(
            Department.class, em));
```

Setting the Parent Property

The property representing the hierarchy by parenthood is defined with the *setParentProperty()* method. In our domain model, the property name is accidentally "`parent`".

```
    departmentContainer.setParentProperty("parent");
    return departmentContainer;
}
```

7.1.2. Creating Hierarchical Entities

The **Department** entities are created in the **DemoDataGenerator** by writing directly to the JPA **EntityManager**. We simply use the *setParent()* for the children to set the hierarchy relationship.

```
em.getTransaction().begin();
Random r = new Random(0);
for (String o : officeNames) {
    Department geoGroup = new Department();
    geoGroup.setName(o);
      for (String g : groupsNames) {
        Department group = new Department();
        group.setName(g);

        … add persons to the group …
```

```
            group.setParent(geoGroup);
            group.setPersons(gPersons);
            em.persist(group);
        }
        em.persist(geoGroup);
    }

    em.getTransaction().commit();
```

8. Filtering Data

TBD