Chapter 4. Data Binding Basics With SQLContainer

We have now created the basic interface for our address book application and a table (`PersonList`) where all the contacts should be displayed. The next step is to bind data to the `PersonList` to make it show something more than the two "Mark Smith" test rows we added in the previous step.

4.1. Database connection and initialization

SQLContainer is an add-on for Vaadin that allows users to easily generate Vaadin containers from various SQL speaking database sources. SQLContainer is documented in more detail in its own user manual which can be downloaded from the Vaadin Directory.

In the scope of the address book application we will create a database connection helper class called `DatabaseHelper`. The purpose of this class is to handle database connections, dummy data generation, container creation and other datasource-related tasks.

We begin by creating the helper class and by adding a method that creates a connection pool, which will be used to connect to the database of your choice. Note that the driver name, connection url, username and password will vary depending on your choice of database and the database settings. In this example we will be connecting to an in-memory HSQLDB database using a `SimpleJDBCConnectionPool` with between two and five open connections.

```
public class DatabaseHelper {
      private JDBCConnectionPool connectionPool = null;

      public DatabaseHelper() {
            initConnectionPool();
      }

      private void initConnectionPool() {
            try {
                  connectionPool = new SimpleJDBCConnectionPool(
                        "org.hsqldb.jdbc.JDBCDriver",
                        "jdbc:hsqldb:mem:sqlcontainer", "SA", "", 2, 5);
            } catch (SQLException e) {
                  e.printStackTrace();
            }
      }
}
```

We want to base our table data on two interrelated tables in a database. We will call these tables `PersonAddress` and `City` (note that the case-sensitivity of the table identifiers depends on the database used).

The `PersonAddress` table will hold the most important data of the address book application. Each row corresponds to an entry in the address book and contains the following columns: `firstname`, `lastname`, `email`, `phonenumber`, `streetaddress` and `postalcode`. In addition to these, the table will also contain an `ID` column, a `version` column for optimistic locking and a `cityid` column which contains a foreign key reference to the `City` table.

The `City` table contains only three columns: an `ID` column, a `version` column and a `name` column that will hold the name of the city.

With HSQLDB we can use the following SQL queries to create these tables. The complete Java method for this can be found from `DatabaseHelper.initDatabase()` but the code is omitted here for clarity.

```
create table city (id integer generated always as identity, name varchar(64),
version integer default 0 not null, primary key (id));

create table personaddress (id integer generated always as identity,
firstname varchar(64), lastname varchar(64), email varchar(64), phonenumber
varchar(64), streetaddress varchar(128), postalcode integer, cityId integer
not null, version integer default 0 not null , FOREIGN KEY (cityId)
REFERENCES city(id), primary key(id));
```

4.2. Initializing the containers

When the database connection is functional and the two tables are created successfully, we can create two SQLContainers; one connected to each table. The container initialization is really simple, we only need to create a `QueryDelegate` that the container will use. In this application, one instance of `TableQuery` will be created for each container. The `TableQuery` constructor requires the name of the table to fetch the data from, and a connection pool that it will use to get the database connections it needs.

Next we'll set the version column for the `TableQuery`. The version column in this application is not actually functional, but the `TableQuery` is designed in a way that requires a version column to be defined in order to support writing (see SQLContainer manual for optimistic locking).

Finally the SQLContainer is instantiated, given the `TableQuery` as its only parameter. The following method of `DatabaseHelper` contains all the code required to create the two containers:

```
    private SQLContainer personContainer = null;
    private SQLContainer cityContainer = null;
    private void initContainers() {
            try {
                /* TableQuery and SQLContainer for personaddress -table */
                TableQuery q1 = new TableQuery("personaddress", connectionPool);
                q1.setVersionColumn("VERSION");
                personContainer = new SQLContainer(q1);

                /* TableQuery and SQLContainer for city -table */
                TableQuery q2 = new TableQuery("city", connectionPool);
                q2.setVersionColumn("VERSION");
                cityContainer = new SQLContainer(q2);
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
```

After the containers are created, the `DatabaseHelper` constructor will call the `fillContainers()` -method, which will add some generated data to the database for demonstration purposes.

Finally, we create a few simple helper methods in the `DatabaseHelper` class to assist us later when we are processing data from the two containers. These methods are listed below and their operation should be quite self-explanatory by their signatures. Full definitions for these methods can be found in the AddressBook - SQLContainer demo application.

```
public String getCityName(int cityId)

public boolean addCity(String cityName)

public SQLContainer getPersonContainer()

public SQLContainer getCityContainer()
```

4.3. Basics of the Data Model

When populating a table there are a few terms that must be understood. Each component that includes data (e.g. a `Table`) has a `Container`. The container includes all the `Item`s in the component. An `Item` corresponds to a row in the table. An `Item` has *properties* (`Property`) which in a table correspond to the columns in a row.

In simple applications we can use the `addItem()`-methods in the `Table` class to populate it with data. The `addItem()`-methods add items (rows), which then in turn can be populated by adding properties and setting values for them - much like you would populate a traditional HTML table by creating `<TR>` and `<TD>` tags.

The data model in Vaadin also contains much more powerful features than this. Each table is connected to a data source which actually contains all the data displayed in the table. A data source can actually supply data for any type of component like a `Table`, `Select`, `Tree`, etc. The data source does not necessarily contain all the items for the table but can dynamically supply the table with data from another source when the `Table` needs it (for instance from a database). The data source is what we in Vaadin call a `Container`.

In this case we are using a `SQLContainer` as the container for the data shown in the table. It can maintain a list of our database contents and supply the table with information fetched directly from the database table when needed.

The nice thing with this container is that we can simply

- Instantiate the SQLContainer with a proper `QueryDelegate`
- Assign the container to a table as a data source
- VOILA, the information is displayed in the table

4.4. Binding Table to a Container

To access the database helper class and our newly created containers in our application we add a new field to our `AddressBookApplication` together with a getter:

```
    private DatabaseHelper dbHelp = new DatabaseHelper();
    public DatabaseHelper getDbHelp() {
        return dbHelp;
    }
```

In addition to this we need to change the `PersonList` constructor (remember to modify the call from `AddressBookApplication` too) so we pass the application instance to it (so it can acquire its data source). We can also remove the dummy test data creation from the previous step, leaving the `PersonList` class as:

```
public class PersonList extends Table {
      public PersonList(final AddressBookApplication app) {
            setSizeFull();
            setContainerDataSource(app.getDbHelp().getPersonContainer());
      }
}
```

The `setContainerDataSource()` sets the SQLContainer which we created in chapter 4.2 to be the data source for the `PersonList` table.

We now have a data source connected to the table and running the application will show that the `PersonList` contains 100 rows of data, based on 100 rows of generated dummy data in the database.

[Figure 4.1. Connected Table]

The order of the columns is still wrong, we at least want the name to come first. Additionally the field names extracted from the `PersonAddress` -table aren't all that human-friendly. We can handle this by first adding two static arrays to the `DatabaseHelper` class. Note that the order of the column headers must correspond to the specified column order.

```
    public static final Object[] NATURAL_COL_ORDER = new Object[] {
            "FIRSTNAME", "LASTNAME", "EMAIL", "PHONENUMBER", "STREETADDRESS",
            "POSTALCODE", "CITYID" };
    public static final String[] COL_HEADERS_ENGLISH = new String[] {
            "First name", "Last name", "Email", "Phone number",
            "Street Address", "Postal Code", "City" };
```

Next we'll add the following two rows to the constructor of the `PersonList` class to sort the columns and make the headers nicer.

```
setVisibleColumns(DatabaseHelper.NATURAL_COL_ORDER);
setColumnHeaders(DatabaseHelper.COL_HEADERS_ENGLISH);
```

The table will now look nicer:

[Figure 4.2. Table Column Names]

4.5. Replacing the CityID with the City Name

As you have probably noticed, the City column of the table currently shows only the City ID (actually the foreign key reference) and not the name of the city. This happens because the city data is stored in another database table and so far we have not even touched it. Therefore we need to modify the City column of the PersonList table to fetch the human-readable name of the city from the City table and display it to the user.

The easiest way to do this is to add a generated column to the table. When the column is generated we will fetch the city name using the `DatabaseHelper` and create a `Label` that will be rendered to the column. The code to do this can be seen below; add the code to the constructor of the `PersonList` class.

```
addGeneratedColumn("CITYID", new ColumnGenerator() {
    public Component generateCell(Table source, Object itemId,
            Object columnId) {
        if (getItem(itemId).getItemProperty("CITYID").getValue() != null) {
            Label l = new Label();
            int cityId = (Integer) getItem(itemId).getItemProperty(
                    "CITYID").getValue();
            l.setValue(app.getDbHelp().getCityName(cityId));
            l.setSizeUndefined();
            return l;
        }
        return null;
    }
});
```

Note that we are using the same column name for this column that is actually a property id in the person container. This way the generated column will override the column that only contained the City IDs.

4.6. Summary

In this step we first created a connection to a database and created two tables for the items that will be shown in the address book. Next we discussed `Container`s, `Item`s and `Property`-values - the central parts of databinding in Vaadin applications. We also introduced the `SQLContainer` and used it to easily link the database tables with our application. We created some dummy data and added it to the database tables using the SQLContainer. Finally we made the table a bit more readable and better looking. Next we will look at how the application starts to respond to user interactions.

5.4. Improving the form

Next we'll concentrate on the `PersonForm`. We want to edit the existing details. The value change listener in the table already puts the selected `Person` into the form. The values also get updated on the server. What we want to do is some usability improvements:

- Use the form as a detailed viewer (read only) in the first place and have an Edit button to enable editing.
- Put the form into a "buffered" mode so the form data gets submitted to the data model only if the Save button is clicked. A Cancel button should return the form to the read-only state with the old values.
- Make the form fields appear in the same order (and using the same captions) as in the table.

Let's start from the `PersonForm` constructor and do the following modifications:

- remove dummy fields.
- add a main application reference via the constructor.
- call `setWriteThrough(false)` to enable buffering.
- make the form footer invisible by default (we don't want to show the footer when the form is missing its datasource i.e. not showing any `Person`).
- let the `Form` implement `ClickListener` and make it listen to the buttons in the footer.
- add one more button to the footer: "Edit". This will be visible when in read-only state.

After these modifications the `PersonForm` looks like the following. Note that you must also modify the main application class to pass itself in the `PersonForm` constructor call.

```
package com.vaadin.demo.tutorial.addressbook.ui;
import com.vaadin.demo.tutorial.addressbook.AddressBookApplication;
import com.vaadin.ui.Button;
import com.vaadin.ui.Form;
import com.vaadin.ui.HorizontalLayout;
import com.vaadin.ui.Button.ClickEvent;
import com.vaadin.ui.Button.ClickListener;

public class PersonForm extends Form implements ClickListener {
    private Button save = new Button("Save", (ClickListener) this);
    private Button cancel = new Button("Cancel", (ClickListener) this);
    private Button edit = new Button("Edit", (ClickListener) this);
    private final AddressBookApplication app;
    public PersonForm(final AddressBookApplication app) {
         this.app = app;
         // Enable buffering so that commit() must be called for the form
         // before input is written to the data. (Form input is not written
         // immediately through to the underlying object.)
         setWriteThrough(false);

         HorizontalLayout footer = new HorizontalLayout();
         footer.setSpacing(true);
         footer.addComponent(save);
         footer.addComponent(cancel);
         footer.addComponent(edit);
         footer.setVisible(false);

         setFooter(footer);
     }

     public void buttonClick(ClickEvent event) {

     }
```

```
}
```

Setting the form to buffering mode (`setWriteThrough(false)`) means that changes made by the user do not immediately update the datasource. Instead the datasource is updated when `commit()` is called for the form. We can then easily revert the data and get the old status back by calling `discard()`. This needs to be implemented in the button click handler:

```
public void buttonClick(ClickEvent event) {
    Button source = event.getButton();
    if (source == save) {
        /* If the given input is not valid there is no point continuing */
        if (!isValid()) {
            return;
        }
        commit();
    } else if (source == cancel) {
        discard();
    } else if (source == edit) {
        setReadOnly(false);
    }
}
```

We can still not test our form as we have made the footer hidden so the form will by default be empty. Next we'll override four methods: `setItemDataSource()`, `setReadOnly()`, `commit()` and `discard()` to make the form work the way we want it to. The default implementations work just fine but we want to add some additional tasks to them.

The `setItemDataSource()` method is called when the datasource is changed, e.g. when the user selects a person in the table. At this point we want to do three things:

1. Set the form to a read-only state by calling `setReadOnly(true)`. This is the initial state.
2. Ensure that the form footer is visible.
3. Make sure the fields are rendered in the order we want (consistent with the table)

In the `setReadOnly()` method we want to control which buttons are shown in the footer. Save and Cancel need to be visible when the form is in edit mode and the Edit button when the form is in read-only mode. Calling `setItemDataSource()` will regenerate all fields inside the form so this is a good place to tell the form in which order we want the fields to be rendered.

The `commit()` and `discard()` methods need to be overridden in order to reflect the changes to the transaction aware SQLContainers. At the very least the `commit()` or `rollback()` method of the SQLContainer needs to be called so that the changes will be either properly written to the database or discarded from the container alltogether.

Our overridden methods look like the following:

```
@Override
public void setItemDataSource(Item newDataSource) {
    if (newDataSource != null) {
        List<Object> orderedProperties = Arrays
                .asList(DatabaseHelper.NATURAL_COL_ORDER);
        super.setItemDataSource(newDataSource, orderedProperties);
        setReadOnly(true);
        getFooter().setVisible(true);
    } else {
        super.setItemDataSource(null);
        getFooter().setVisible(false);
    }
}
```

```
@Override
public void setReadOnly(boolean readOnly) {
    super.setReadOnly(readOnly);
    save.setVisible(!readOnly);
    cancel.setVisible(!readOnly);
    edit.setVisible(readOnly);
}

@Override
public void commit() throws Buffered.SourceException {
    /* Commit the data entered to the person form to the actual item. */
    super.commit();

    /* Commit changes to the database. */
    try {
        app.getDbHelp().getPersonContainer().commit();
    } catch (UnsupportedOperationException e) {
        e.printStackTrace();
    } catch (SQLException e) {
        e.printStackTrace();
    }
    setReadOnly(true);
}

@Override
public void discard() throws Buffered.SourceException {
    super.discard();
    /* On discard roll back the changes. */
    try {
        app.getDbHelp().getPersonContainer().rollback();
    } catch (UnsupportedOperationException e) {
        e.printStackTrace();
    } catch (SQLException e) {
        e.printStackTrace();
    }
    /* Clear the form */
    setItemDataSource(null);
    setReadOnly(true);
}
```

Try it out and ensure that changing mode (*read only* / *editable*) works correctly and that the values in the `PersonList` are not updated until the Save button is clicked (when not using the buffering mode they are immediately updated). Remember to run Organize Imports in Eclipse after copy/paste editing.

5.5. Implement logic to add new contacts

Next we want to create new contacts. Most of this logic will go in `PersonForm` but we must first create some logic in the main application class. Before passing control to `PersonForm` for the actual addition we want to make sure that `ListView` is being shown and we also want to clear any possible selections from the table.

Add more logic to the click handler in `AddressBookApplication` and make a click on the Add contact button pass control to the `addNewContact()` method. Implement it as follows:

```
private void addNewContact() {
    showListView();
    personForm.addContact();
}
```

Next we create the `addContact()` method in `PersonForm`. Now the problem is that we do not have an existing item in the container to edit but need to create a new one. So we add a new, temporary item to the person container and put it into the form without affecting the actual table in the database. To avoid the user accidentally adding dozens of empty items, we'll do a container rollback first. We do not want a read only view when adding a contact so our `addContact()` method will look like this:

```
public void addContact() {
    /* Roll back changes just in case */
    try {
        app.getDbHelp().getPersonContainer().rollback();
    } catch (SQLException ignored) {
    }
    Object tempItemId = app.getDbHelp().getPersonContainer().addItem();
    setItemDataSource(app.getDbHelp().getPersonContainer().getItem(
            tempItemId));
    setReadOnly(false);
}
```

5.6. Map the City table to the contacts

Adding and editing contacts should now work but you will notice that the city of a contact is shown only by its ID, not by the actual city name. Since this is not very usable, we need to create a mapping from the city ID to the name of the city and display that instead.

For this purpose we will constuct a ComboBox which will contain the existing cities as well as allow the user to enter new ones. This way selecting an existing city becomes much easier.

Replacing the normal text field with a combo box will introduce you to the system used in Vaadin for populating a form. The same mechanism can be used in tables if you put the table into editable mode. Fields are generated using the strategy pattern by a class implementing the `FieldFactory` interface.

By default the `BaseFieldFactory` class is used and is generally a good base for customizations. The `FieldFactory` interface has several methods which we do not want to implement and `BaseFieldFactory` has good implementations for these. Later we'll use the same field factory to tune the form fields even more.

We start by adding a `ComboBox` field to our `PersonForm`:

```
private final ComboBox cities = new ComboBox("City");
```

We make adding new items possible and populate the ComboBox with existing cities from our data source. We will also set the ComboBox into immediate mode. Add the following code to the `PersonForm` constructor:

```
/* Allow the user to enter new cities */
cities.setNewItemsAllowed(true);
/* We do not want to use null values */
cities.setNullSelectionAllowed(false);
/* Cities selection */
cities.setContainerDataSource(app.getDbHelp().getCityContainer());
cities.setItemCaptionPropertyId("NAME");
cities.setImmediate(true);
```

Finally we'll make a field factory extending the `BaseFieldFactory`. Using the field factory we can choose what kind of field to use for each property. For the city property we'll return the cities ComboBox, for all other properties we'll let the `BaseFieldFactory` take care of creating the field.

```
setFormFieldFactory(new DefaultFieldFactory() {
    @Override
    public Field createField(Item item, Object propertyId,
       Component uiContext) {
        Field field;
        if (propertyId.equals("CITYID")) {
            field = cities;
        } else {
            field = super.createField(item, propertyId, uiContext);
        }
        return field;
    }
});
```

Now that we have the cities nicely listed in a ComboBox, we need to allow the user to add cities that are not yet present in the City database table. This is easy to implement using a NewItemHandler from the AbstractSelect class. In this case, our NewItemHandler will call the addCity() method of the DatabaseHelper class. Due to the temporary row IDs used in the

SQLContainer, we will also need add a listener to the City container to receive notifications about the automatically generated IDs of added items. Add the following code to the end of the `PersonForm` constructor:

```
        /* NewItemHandler to add new cities */
        cities.setNewItemHandler(new AbstractSelect.NewItemHandler() {
            public void addNewItem(String newItemCaption) {
                app.getDbHelp().addCity(newItemCaption);
            }
        });
        /* Add PersonForm as RowIdChangeListener to the CityContainer */
        app.getDbHelp().getCityContainer().addListener(this);
```

To implement the `RowIdChangeListener`, we first need to declare that the `PersonForm` class implements `QueryDelegate.RowIdChangeListener`. After the declaration, we must provide the implementation. In our implementation, we will make the cities `ComboBox` select the city that was just inserted; this `setValue()` call is necessary since the `ItemID` of the city has just changed. Finally, we'll set the key of the added city to the person item that we are currently editing, so that its `CITYID` property will reference the correct city. The implementation can be done like this:

```
    public void rowIdChange(RowIdChangeEvent event) {
        cities.setValue(event.getNewRowId());
        getItemDataSource().getItemProperty("CITYID").setValue(
                event.getNewRowId().getId()[0]);
    }
```

The final touch will be done in the `setItemDataSource()` method of the `PersonForm` class. Here we will add code that will select the correct city from the ComboBox every time a new item is set to the `PersonForm`. Since the item from the person container will only contain the key of the city, but the ComboBox requires a complete `RowId`, we will need to generate one. Add the following code to the `PersonForm.setItemDataSource()` method, right after the call to `super.setItemDataSource(newDataSource, orderedProperties)`.

```
      /* Select correct city from the cities ComboBox */
      setReadOnly(false);
      if (newDataSource.getItemProperty("CITYID").getValue() != null) {
          cities.select(new RowId(new Object[] { newDataSource
                        .getItemProperty("CITYID").getValue() }));
      } else {
          cities.select(cities.getItemIds().iterator().next());
      }
```

If the value of the city reference is null (e.g. the item is new), the first item will be selected from the cities ComboBox.

Try it out and check that creation of new contacts, editing of old ones and adding new cities works properly.

5.7. Implementing the search functionality

We have previously added logic for the search button so a search view is displayed when the button in the toolbar is clicked but the view is still empty. We want to create a simple view where you can enter a search term, select which field to match the term to and optionally save the search for later use. The finished view should look like this:

[Figure 5.1. Search User Interface]

We implement this using a FormLayout which will cause the captions to be rendered to the left of the fields instead of above (as the case is with a VerticalLayout). In addition to the layout we need a couple of TextFields, a NativeSelect, a CheckBox and a Search button. The following code is used for creating the layout.

```
package com.vaadin.demo.tutorial.addressbook.ui;
import com.vaadin.demo.tutorial.addressbook.AddressBookApplication;
import com.vaadin.demo.tutorial.addressbook.data.PersonContainer;
import com.vaadin.ui.Button;
import com.vaadin.ui.CheckBox;
import com.vaadin.ui.FormLayout;
import com.vaadin.ui.NativeSelect;
import com.vaadin.ui.Panel;
import com.vaadin.ui.TextField;
import com.vaadin.ui.Button.ClickEvent;
import com.vaadin.ui.Button.ClickListener;
public class SearchView extends Panel {
    private TextField tf;
    private NativeSelect fieldToSearch;
    private CheckBox saveSearch;
    private TextField searchName;
    private AddressBookApplication app;
    public SearchView(final AddressBookApplication app) {
         this.app = app;

         setCaption("Search contacts");
         setSizeFull();
         /* Use a FormLayout as main layout for this Panel */
         FormLayout formLayout = new FormLayout();
         setContent(formLayout);
         /* Create UI components */
         tf = new TextField("Search term");
         fieldToSearch = new NativeSelect("Field to search");
         saveSearch = new CheckBox("Save search");
         searchName = new TextField("Search name");
         Button search = new Button("Search");
         /* Initialize fieldToSearch */
         for (int i = 0; i < DatabaseHelper.NATURAL_COL_ORDER.length; i++) {
             fieldToSearch.addItem(DatabaseHelper.NATURAL_COL_ORDER[i]);
             fieldToSearch.setItemCaption(DatabaseHelper.NATURAL_COL_ORDER[i],
                     DatabaseHelper.COL_HEADERS_ENGLISH[i]);
         }
         fieldToSearch.setValue("lastName");
         fieldToSearch.setNullSelectionAllowed(false);
         /* Pre-select first field */
         fieldToSearch.select(fieldToSearch.getItemIds().iterator().next());
         /* Initialize save checkbox */
         saveSearch.setValue(true);
         /* Add all the created components to the form */
         addComponent(tf);
         addComponent(fieldToSearch);
```

```
            addComponent(saveSearch);
            addComponent(searchName);
            addComponent(search);
        }
}
```

All property ids intended to be visible from the `SQLContainer` are added to the field select in the for-loop. We cannot use the `SQLContainer` as a data source for the field as this would add all items to the drop-down instead of the properties (headers in the table). The `setItemCaption()` sets what is being displayed to more human-readable form, like the headers in the table.

We want to force the user to select a field to search from so we set *nullSelectionAllowed* to false, otherwise an empty value would be added to the select (and this would also be selected by default). We also make the assumption that the user most of the time wants to save his search so the `saveSearch` checkbox is checked by default (value = true).

Now we can run the program and the layout should appear when clicking on the Search button. There is one small problem though, nothing happens when you click on Search. So we need to add some logic to the view.

We want to add two things: unchecking the `saveSearch` checkbox should hide the `searchName`-field and clicking on the Search button should perform a search.

Hiding the `searchName` field when un-checking the `saveSearch` checkbox requires only a few lines of code to the `SearchView` constructor:

```
saveSearch.setImmediate(true);
saveSearch.addListener(new ClickListener() {
    public void buttonClick(ClickEvent event) {
        searchName.setVisible(event.getButton().booleanValue());
    }
});
```

First we need to set the `saveSearch` checkbox to immediate mode so we get an event every time the user interacts with it. We then attach a click listener to it so we can react to the user checking or un-checking the checkbox. We react to the click event simply by setting the visibility of the `searchName` field to the checkbox status (true if checked, false if unchecked).

Performing the search requires a little more effort. We need to pass the input from the user to the SQLContainer so it can generate an appropriate query and fetch only the matching rows. Additionally we need to store the search if the user has checked the `saveSearch` checkbox.

First we create a simple data class, SearchFilter, where we can store the input from the user.

```
package com.vaadin.demo.tutorial.addressbook.data;
import java.io.Serializable;
public class SearchFilter implements Serializable {

      private final String term;
      private final Object propertyId;
      private String propertyIdDisplayName;
      private String termDisplayName;
      private String searchName;

      public SearchFilter(Object propertyId, String searchTerm, String name) {
            this.propertyId = propertyId;
            this.term = searchTerm;
            this.searchName = name;
      }

      public SearchFilter(Object propertyId, String searchTerm, String name,
```

```
            String propertyIdDisplayName, String termDisplayName) {
        this(propertyId, searchTerm, name);
        setPropertyIdDisplayName(propertyIdDisplayName);
        setTermDisplayName(termDisplayName);
    }

    // + getters and setters
}
```

We then add a click listener to the search button which will fetch the value of the search field and term, create a `SearchFilter` array and pass it on to the main application. If `saveSearch` has been checked we also tell the application to save the search. Add the following code to the SearchView constructor:

```
search.addListener(new Button.ClickListener() {
    public void buttonClick(ClickEvent event) {
        performSearch();
    }
});
```

As the code is more than a few lines, we'll add it as a separate method:

```
    private void performSearch() {
        String searchTerm = (String) tf.getValue();
        if (searchTerm == null || searchTerm.equals("")) {
            getWindow().showNotification("Search term cannot be empty!",
                    Notification.TYPE_WARNING_MESSAGE);
            return;
        }
        List<SearchFilter> searchFilters = new ArrayList<SearchFilter>();
        searchFilters.add(new SearchFilter(fieldToSearch.getValue(),
                        searchTerm, (String) searchName.getValue(),
                        fieldToSearch.getItemCaption(fieldToSearch.getValue()),
                        searchTerm));

        /* If Save is checked, save the search through the main app. */
        if (saveSearch.booleanValue()) {
            if (searchName.getValue() == null
                    || searchName.getValue().equals("")) {
                getWindow().showNotification(
                        "Please enter a name for your search!",
                        Notification.TYPE_WARNING_MESSAGE);
                return;
            }
            SearchFilter[] sf = {};
            app.saveSearch(searchFilters.toArray(sf));
        }
        SearchFilter[] sf = {};
        app.search(searchFilters.toArray(sf));
    }
```

Now we need to add the search and `saveSearch` logic to the application class. Note that since the `PersonAddress` container contains both string and numeric properties, we can not use the `addContainerFilter` method. Instead, we must fetch the type of the column from the container and create an accordingly typed `Filter` instance.

```
  public void search(SearchFilter... searchFilters) {
      SQLContainer c = getDbHelp().getPersonContainer();
      /* Clear previous filters */
      c.removeAllContainerFilters();
      /* Add the filter(s) to the person container. */
      for (SearchFilter searchFilter : searchFilters) {
          Filter f = new Filter((String) searchFilter.getPropertyId(),
```

```
                    ComparisonType.CONTAINS, searchFilter.getTerm());
            if (Integer.class.equals(c.getType(searchFilter.getPropertyId()))) {
                try {
                    f = new Filter((String) searchFilter.getPropertyId(),
                                   ComparisonType.EQUALS,
                                   Integer.parseInt(searchFilter.getTerm()));
                } catch (NumberFormatException nfe) {
                    return;
                }
            }
            getDbHelp().getPersonContainer().addFilter(f);
        }
        showListView();
    }
```

The search method simply removes any previous filters and then adds a container filter to the person data source using the data from the `searchFilter` array argument. The `true` and `false` parameters to the `addContainerFilter()` call turns on `ignoreCase` and makes it match the term as a sub string (not only as prefix, i.e. it will search for "`*term*`", not "`term*`").

The filter also needs to be cleared when the user selects "Show All" in the tree, or is adding a new contact. This can be done in `itemClick()` and `addNewContact()` methods of the AddressBookApplication:

```
[...]
if (NavigationTree.SHOW_ALL.equals(itemId)) {
      /* Clear all filters from person container */
      getDbHelp().getPersonContainer().removeAllContainerFilters();
      showListView();
}
[...]

private void addNewContact() {
    showListView();
    tree.select(NavigationTree.SHOW_ALL);
    /* Clear all filters from person container */
    getDbHelp().getPersonContainer().removeAllContainerFilters();
    personForm.addContact();
}
```

The `saveSearch()` method adds a new item to the tree, under the Search node, and then selects that node:

```
public void saveSearch(SearchFilter... searchFilter) {
    tree.addItem(searchFilter);
    tree.setItemCaption(searchFilter, searchFilter[0].getSearchName());
    tree.setParent(searchFilter, NavigationTree.SEARCH);
    // mark the saved search as a leaf (cannot have children)
    tree.setChildrenAllowed(searchFilter, false);
    // make sure "Search" is expanded
    tree.expandItem(NavigationTree.SEARCH);
    // select the saved search
    tree.setValue(searchFilter);
}
```

The `searchFilter` array is used as the `itemId` for the tree node. A caption for the new item will be set using the `searchName` of the first `searchFilter` in the array.

Using the `searchFilter` array as the item ID in the tree makes it easy to add a case to the `itemClick()` handler for the tree so the search is performed every time the saved search is clicked. Make the following addition to `itemClick()` handler in

`AddressBookApplication`:

```
public void itemClick(ItemClickEvent event) {
      if(itemId != null) {
            [...]
            } else if (itemId instanceof SearchFilter[]) {
                  search((SearchFilter[]) itemId);
            }
      }

}
```

5.8. Implementing Search Based on a Foreign Key

In the previous chapter we created a simple search view, implemented the search functionality and support for saving the searches. However you may have noted that currently only city Ids will be searched if the City field is used in the search, not the actual city names. Therefore we need to modify the search to first look up the correct IDs from the city container, and after that create the required amount of search filters for the person container. Now you will also see why earlier the search filters were passed around and saved in an array.

We start from the `SearchView.performSearch()` method and replace this:

```
      searchFilters.add(new SearchFilter(fieldToSearch.getValue(),
                  searchTerm, (String) searchName.getValue(),
                  fieldToSearch.getItemCaption(fieldToSearch.getValue()),
                  searchTerm));
```

with this:

```
       if (!"CITYID".equals(fieldToSearch.getValue())) {
            /* If this is NOT a City search, one filter is enough. */
            searchFilters.add(new SearchFilter(fieldToSearch.getValue(),
                    searchTerm, (String) searchName.getValue(), fieldToSearch
                            .getItemCaption(fieldToSearch.getValue()),
                    searchTerm));
        } else {
            SQLContainer cc = app.getDbHelp().getCityContainer();
            cc.addContainerFilter("NAME", searchTerm, true, false);
            for (Object cityItemId : cc.getItemIds()) {
                searchFilters.add(new SearchFilter("CITYID",
                        cc.getItem(cityItemId).getItemProperty("ID").getValue()
                                .toString(), (String) searchName.getValue(),
                        fieldToSearch.getItemCaption(fieldToSearch.getValue()),
                        searchTerm));
            }
            cc.removeAllContainerFilters();
        }
```

In the above code all the other fields are handled exactly as before, except for the City field. If a City search is made, we will fetch the city container and add a container filter to it. This filter will search for the term the user entered. Once the results are in we will generate `SearchFilter`s with each of the matching cities' IDs.

Finally the search method of the application class also requires a slight modification. The search mode of the person SQLContainer must be changed from inclusive (`AND`) to exclusive (`OR`) in order to get any results at all. This is done simply by adding the following code to the method, right before the for-loop which adds the containerfilters.

```
        if (searchFilters.length > 1) {
            getDbHelp().getPersonContainer().setFilteringMode(
                    FilteringMode.FILTERING_MODE_EXCLUSIVE);
        } else {
```

```
    getDbHelp().getPersonContainer().setFilteringMode(
            FilteringMode.FILTERING_MODE_INCLUSIVE);
}
```

Chapter 6. Tuning the user experience

The application we have built so far is a fully functional application, but lacking some of the nice stuff which would make it easier to use. For example email addresses in the table could be links that open the user's default email application. Other examples discussed here are validation of user input in forms and giving proper feedback to users.

6.1. Turning email addresses into links

The email addresses should be links for directly sending an email and we should validate the data entered to avoid unintentional mistakes.

[Figure 6.1. Email Adresses as Text]

The table we have created and populated so far only lists data directly from the datasource. We would, however, want to turn the email addresses into links so the user can directly click on them to send email. We can accomplish this by using the generated column feature of the Table as shown in the code below. By adding a generated column with the same id as an existing column ("`email`") we override the existing column and display the generated column instead. Using a different id would result in both the plain and the linked email column being shown. The following code should be added to the `PersonList` constructor:

```
// customize email column to have mailto: links using column generator
addGeneratedColumn("EMAIL", new ColumnGenerator() {
    public Component generateCell(Table source, Object itemId,
                                  Object columnId) {
        if (getItem(itemId).getItemProperty("EMAIL").getValue() != null) {
            Link l = new Link();
            l.setResource(new ExternalResource("mailto:"
                + getItem(itemId).getItemProperty("EMAIL").getValue()));
            l.setCaption(getItem(itemId).getItemProperty("EMAIL")
                .getValue().toString());
            return l;
        }
        return null;
    }
});
```

The `ColumnGenerator` uses a simple interface with only one method, `generateCell()` which is called every time a value for the column is needed. Passed to this method is the `itemId` for the row which in our example is the same as the `Person`. We acquire the email address from the `Person` object and generate a `Link` component with the standard HTML "`mailto:<address>`" target. Returning the `Link` component from the generator will make the `Table` automatically insert it in the correct position (in place of the original, plain text email address).

The table now contains clickable email links:

[Figure 6.2. Email Adresses as Links]

6.2. Notifications

Notifications are a nice way of informing the user what is happening without adding e.g. a label which requires some space in the view. Let's test the notifications by adding a notification when searching, so the user sees what he searched for. This will be displayed at the same time as the results are displayed. We will add the notification for both normal and saved searches so the correct place for the code is the search method in the application class.

```
getMainWindow().showNotification(
          "Searched for:<br/> "
                  + searchFilters[0].getPropertyIdDisplayName() + " = *"
                  + searchFilters[0].getTermDisplayName()
                  + "*<br/>Found " + c.size() + " item(s).",
          Notification.TYPE_TRAY_NOTIFICATION);
```

Now the user gets a nice little popup in the bottom right corner of the screen which tells what the search criteria was and how many rows matched.
`Notification.TYPE_TRAY_NOTIFICATION` places the notification in the lower right corner; feel free to experiment with other types to see what happens.

[Figure 6.3. Tray Notification]

In addition to the successful search notification, we can also add a notification for invalid search terms . To accomplish this we will add the following code to the search method of the main application. The line should be added inside the catch block, right before the return statement.

```
getMainWindow().showNotification("Invalid search term!");
```

6.3. Fixing captions of form fields

One small detail that catches the eye are the captions of the `PersonForm`. As can be seen, they are not of the nice human-friendly type we used in the table, but they are once again the column names directly from the database. Luckily we can fix this very easily by adding the following code to the field factory created at the end of the previous chapter. The code should be inserted right before the '`return field;`' statement.

```
    /* Set the correct caption to each field */
    for (int i = 0; i < DatabaseHelper.NATURAL_COL_ORDER.length; i++) {
        if (DatabaseHelper.NATURAL_COL_ORDER[i].equals(propertyId)) {
            field.setCaption(DatabaseHelper.COL_HEADERS_ENGLISH[i]);
        }
    }
```

6.4. Automatically validate user input

We want to validate two of the fields in the form: the postal code field should contain five numbers and not start with a zero, and the email address should be in a valid format.

For the postal code use a `RegexpValidator` which implements the `Validator` interface. The user input is checked against the regular expression

```
"[1-9][0-9]{4}"
```

which returns true only for a 5 digit `String` where the first character is `1-9`.

We assign the validator to the field in our `FieldFactory` and at the same time make the field required. We make another improvement at the same time by changing the null representation for the field. Instead of the default "`null`" we want to display an empty field before the user has entered any value. The code for the `postalCode` field in the `FieldFactory` is now:

```
/*
 * Field factory for overriding how the component for city selection is
 * created
 */
setFormFieldFactory(new DefaultFieldFactory() {
    @Override
    public Field createField(Item item, Object propertyId,
            Component uiContext) {
        if (propertyId.equals("city")) {
            return cities;
        }
        Field field = super.createField(item, propertyId, uiContext);
        if (propertyId.equals("postalCode")) {
            TextField tf = (TextField) field;
            /*
             * We do not want to display "null" to the user when the
             * field is empty
             */
            tf.setNullRepresentation("");
            /* Add a validator for postalCode and make it required */
            tf.addValidator(new RegexpValidator("[1-9][0-9]{4}",
                "Postal code must be a five digit number and cannot
                      start with a zero."));
            tf.setRequired(true);
        }
        return field;
    }
});
```

The field now contains a red star, telling the user that it must be filled out. It is also initially empty.

[Figure 6.4. Validated Field Initially Empty]

If the user enters an invalid value into the field, a red exclamation mark appears. When the user moves the mouse cursor over the field a tooltip, containing the error message from the validator, appears.

[Figure 6.5. Validated Field with Invalid Value]

We do the same thing for the email field but use a different validator implementation, which checks for legal e-mail address format. Using the `EmailValidator` provided by Vaadin is left here as an exercise for the reader. Additionally, you might want to change the `null` representation for all the

TextFields in the form by stating the following in the `createField` method:

```
/* Set null representation of all text fields to empty */
if (field instanceof TextField) {
    ((TextField) field).setNullRepresentation("");
}
```

6.5. Making Table row selection more intuitive

You may have noticed that the selected row of the main Table does not always reflect the previous action, or even seem logical at all. Fortunately, this usability-enhancing feature can be implemented rather simply. First, we'll need to create a method that will fix the selected row and scroll it into view if necessary. A reasonable place to implement this is the `PersonList` class. Add the following method:

```
public void fixVisibleAndSelectedItem() {
    if ((getValue() == null || !containsId(getValue())) && size() > 0) {
        Object itemId = getItemIds().iterator().next();
        select(itemId);
        setCurrentPageFirstItemId(itemId);
    } else {
        setCurrentPageFirstItemId(getValue());
    }
}
```

This method will scroll the current selection into view, or if no selection is made, select the first row from the table.

Now, where should we call this method? If we observe the selections of the `PersonList` table while using the application, we'll notice three actions which will make the selected row disappear:

- After moving from a search result view to the Show All view
- After moving from the Show All view to a search result view
- After a new contact has been added

For the first two cases the fix is easy. We'll only need to add a call to the method we just created into the `showListView()` and `showSearchView()` methods of the `AddressBookApplication`, for example like this:

```
private void showSearchView() {
    setMainComponent(getSearchView());
    personList.fixVisibleAndSelectedItem();
}
```

The third case is a bit more problematic, since we need to get the `RowId` of the added contact to be able to select it from the `PersonList`. The trick is to use the `RowIdChangeListener` like we did in chapter 5.6. with the City container. First, we'll make the `AddressBookApplication` implement the `QueryDelegate.RowIdChangeListener` interface, and implement the `rowIdChange()` method as follows. We'll also add a call to the `fixVisibleAndSelectedItem()` method

```
public void rowIdChange(RowIdChangeEvent event) {
    /* Select the added item and fix the table scroll position */
    personList.select(event.getNewRowId());
    personList.fixVisibleAndSelectedItem();
}
```

Finally we should register the main application class as a listener to the `PersonAddress` container, and we should be all done! The following statement at the end of the `AddressBookApplication init()` method will do the job:

```
dbHelp.getPersonContainer().addListener(this);
```

Go ahead and see if the table row selections make more sense after these enhancements.

6.6. Enabling advanced features in a Table

The `Table` component in Vaadin is quite powerful. There are several features available that enhance the user experience. Some of these (like sorting) require support from the data source. The best place to enable these features is naturally the constructor in our `PersonList` class.

Sorting is on by default provided the data source supports it. The `SQLContainer` has sorting support for every field whose type implements `Comparable`. Try clicking on column headers to sort the table according to any column. Sorting can be explicitly disabled if necessary.

Other features enhancing the usability of a Table are column reordering and column collapsing. These are disabled by default. Add the following lines to `PersonList` and then try dragging column headers. Also try the menu appearing in the top right corner of the table.

```
setColumnCollapsingAllowed(true);
setColumnReorderingAllowed(true);
```