

Eclipse Scout One Day Tutorial

Version 22.0

Table of Contents

The “Contacts” Application	2
Tutorial Overview	4
Setting up the Initial Project	6
Creating the initial Codebase	6
Removing unnecessary Components	8
Changes to Class WorkOutline	9
Changes to Class Desktop	13
What have we achieved?	15
Adding the Person and Organization Page	17
Creating additional Packages	17
Creating the Country Lookup Call	18
Creating the Person Page	19
Adding Table Columns to the Page	21
Link the Person Page to the Contacts Outline	26
Adding the Company Page	27
What have we achieved?	28
Creating and Accessing the Database	30
Adding the Infrastructure	30
Scout Config Properties	33
The SQL Service and SQL Statements	36
The Database Setup Service	38
What is missing?	40
Fetching Organization and Person Data	41
What have we achieved?	43
Adding a Form to Create/Edit Persons	44
Designing the Person Form	44
Implementing the Form	47
Adding a Gender Code Type	55
Adding Form Fields	58
Person Form Handler and Person Service	66
What have we achieved?	70
Form Field Validation and Template Fields	71
Simple Form Field Validation	71
Complex Form Field Validation	72
Creating Template Fields	76
Adding a simple URL Input Form to the Picture Field	79
More Template Fields	84
What have we achieved?	84

Adding the Company Form	86
What have we achieved?	92
Linking Organizations and Persons	93
Creating an Organization Lookup Call	93
Using the Lookup Call in the Person Form and the Person Table	95
Link the Person Page to Organizations	97
What have we achieved?	103
Additional Concepts and Features	104
Theming and Styling	104
Modular Scout Applications	104
Infrastructure	105
Git configuration	106



Looking for something else? Visit <https://eclipsescout.github.io> for all Scout related documentation.

In this tutorial we will create the “Contacts” Scout application. The goal of this application is to learn about the most prominent features of the Eclipse Scout framework using a fully functional application.

The application is kept small enough to complete this tutorial within less than a day. An extended version of “Contacts” is available as a Scout sample application on [Github](#).

This tutorial uses the Eclipse IDE and the Scout SDK. If you have successfully completed the [Hello Scout Tutorial](#) you should have a running Eclipse Scout installation. If not, then you should do the Hello Scout Tutorial now.

The “One Day Tutorial” is organized as follows. In the first section, the finished “Contacts” application is explained from the user perspective. The remaining sections focus on the individual steps to implement the “Contacts” tutorial application.

The “Contacts” Application

The “Contacts” demo application is a client server application to manage personal contacts, organizations and events. The persistence of entered data is achieved via simple JDBC access to a Derby database.

It is recommended that you first import the full “Contacts” demo application into a separate workspace. This gives you the possibility to check your source code against the full implementation during the various steps of the tutorial. To do so, just clone the Scout demo repository and import it into a new Eclipse workspace.

```
git clone https://github.com/bsi-software/org.eclipse.scout.docs.git
```

Alternatively, you can also view the source code of the “Contacts” demo application on [Github](https://github.com/bsi-software/org.eclipse.scout.docs).



Figure 1. The “Contacts” application with the person page.

The “Contacts” application shows the basic user interface layout of a typical Scout application. The main areas of this layout are briefly introduced below.

Outline Button

In [Figure 1](#) the top left area shows a folder icon that represents the "Contacts" outline. The small down arrow at the folder icon indicates that additional outlines are available when clicking on this view button. On the right of the button with the folder icon is a second outline button that activates a search outline (not implemented yet).

Navigation Tree

The navigation tree on the left side of the layout shows the pages that are available for the selected outline. For the "Contacts" outline, the navigation tree provides access to the pages "Persons", "Organizations" and "Events". Selecting a page then shows associated information on the right side in the bench area. In the case of the selected "Persons" page the bench area shows a list of persons in the form of a table.

Header

The header area is located at the top and holds the available top level menus. In this example these are the "Quick access", "Options" menu points as well as a user menu that shows the username of the currently logged in user "mzi".

Bench

The bench represents the main display area of a Scout application. When selecting the "Persons" page, a table provides access to all available persons as shown in [Figure 1](#). Selecting a specific person provides access to all actions that are available for the selected person. The selected person can then be opened with the *Edit* menu which opens the person in a view that is displayed in the bench area again as shown in [Figure 2](#).

For entering and editing of data in Scout applications views are used in most cases. Views are displayed in the bench area of a Scout application. Several views can also be opened simultaneously. To show a specific view the user has to click on the view button associated with the desired view. An example of an opened view is shown for person "Alice" in [Figure 2](#).



The screenshot shows a web browser window titled "Contacts Application" with the URL `localhost:8082/?dl=outline-117708&i=contacts`. The interface has a dark blue sidebar on the left with a search icon and a menu containing "Contacts", "Persons", "Organizations", and "Events". The main content area is titled "Person Alice" and includes "Quick access" and "Options" links, along with a user profile for "sre". Below the title are "OK" and "Cancel" buttons. The form contains a profile picture of Alice, a vertical ellipsis menu, and fields for "First name" (Alice), "Last name", "Date of birth" (11/26/1865), and "Gender" (Female selected). At the bottom, there are tabs for "Contact Info", "Work", "Events", and "Notes". The "Contact Info" tab is active, showing fields for "Street", "Location" (Daresbury, Ches United Kingd), "Phone", "Mobile", and "E-Mail". A "Show on map" link is also present.

Figure 2. The "Contacts" application with a person opened in a form.

Tutorial Overview

This tutorial walks you through the implementation of a Scout application consisting of a frontend and a backend application. The frontend application contains outlines with navigation trees, pages to present information in tabular form, and forms to view and edit data. In the backend application the tutorial shows how to implement services, logging, database access, and several other aspects of Scout applications.

The tutorial is organized as a sequence of consecutive steps as listed below. Each step is described in a individual section that results in a executable application that can be tested and compared against the full "Contacts" demo application.

Step 1: Setting up the Initial Project (*Setting up the Initial Project*)

We will create a new project and prepare the generated code base by adapting existing components and deleting unnecessary components. At the end of step one we have a project setup that allows us to start adding new components to the application.

Step 2: Adding the Person and Organization Page (*Adding the Person and Organization Page*)

The second step adds the user interface components to display persons and organizations. For this a "Persons" page and an "Organizations" page are created and added to the "Contacts" outline as shown in [Figure 1](#).

Step 3: Creating and Accessing the Database (*Creating and Accessing the Database*)

This step concentrates on the backend of the "Contacts" application. The covered topics include dealing with application properties, setup and access of a database and using the database to provide data for the person and organization page created in the previous step.

Step 4: Adding a Form to Create/Edit Persons (*Adding a Form to Create/Edit Persons*)

After having access to the database the components that allow a user to create and edit persons and organizations in the user interface of the "Contacts" application can be added. In addition, this tutorial step also demonstrates how to design and implement complex form layouts with the Scout framework.

Step 5: Form Field Validation and Template Fields (*Form Field Validation and Template Fields*)

This step provides an introduction into form field validation and the creation of template fields. Validation of user input is important for many business applications and template fields help to improve code quality with a mechanism to reuse application specific user interface components.

Step 6: Adding the Company Form (*Adding the Company Form*)

We create the organization form to create and enter organizations in the "Contacts" application. As we can reuse many of the components developed so far this is the shortest tutorial step.

Step 7: Linking Organizations and Persons (*Linking Organizations and Persons*)

In this step we modify the user interface to implement a 1:n relationship between organizations and persons. This includes the creation of a hierarchical page structure for organization, adding an organization column to the person page and adding an organization field to the person form to manage the association of a person to an organization.

Step 8: Additional Concepts and Features (*Additional Concepts and Features*)

The last tutorial part discusses the gap between the tutorial application and the complete "Contacts" demo application.

Setting up the Initial Project

This section deals with setting up the initial workspace and code base for the "Contacts" application. The creation up of the initial project setup consists of the tasks listed below.

- Creating the initial Codebase ([Creating the initial Codebase](#))
- Removing unnecessary Components ([Removing unnecessary Components](#))
- Changes to Class WorkOutline ([Changes to Class WorkOutline](#))
- Changes to Class Desktop ([Changes to Class Desktop](#))

This first step of the "Contacts" tutorial ends with a review of the results of this first tutorial step in [What have we achieved?](#).

Creating the initial Codebase

In case your workspace contains modules from the “Hello World” tutorial, you may want to multi-select them and to either close them by invoking the context menu “Close Projects” or to delete them by invoking the context menu “Delete”. The initial code for the “Contacts” application is then generated using the *New Scout Project* wizard. For the wizard fields you may use the values below and as shown in [Figure 3](#)

- *Group Id*: `org.eclipse.scout`
- *Artifact Id*: `contacts`
- *Display Name*: "Contacts Application"

New Scout Project

Create a Scout Project
Create a new Scout Project

Project Name

Group Id: org.eclipse.scout

Artifact Id: contacts

Display Name: Contacts Application

Programming language of the user interface

☐ JavaScript & JSON

☒ Java

Project Location

☒ Use default Workspace location

Target Directory: C:\eclipse\workspace\helloworld Browse...

? < Back Next > **Finish** Cancel

Figure 3. The creation of the initial "Contacts" application.

To create this initial application click on **[Finish]**. The project wizard then creates a number of Maven modules as shown in [Figure 4](#).

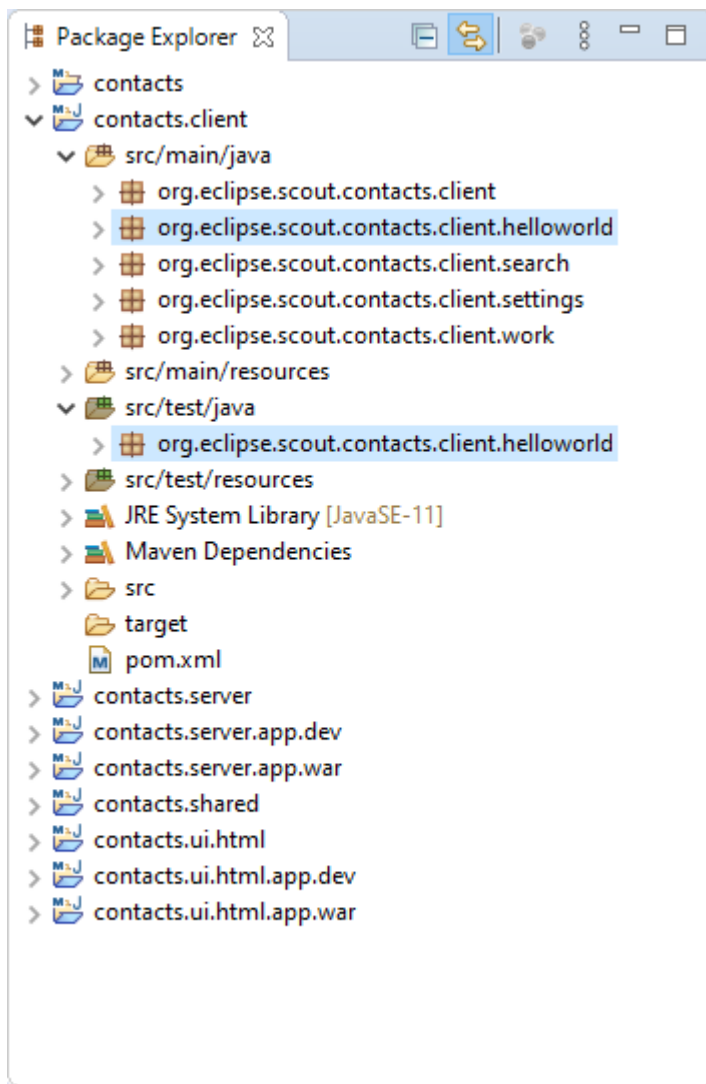


Figure 4. The package explorer with the initial Maven modules created for the "Contacts" application.

Removing unnecessary Components

We start with removing the `*.helloworld` and `*.settings` packages in all Maven modules of the "Contacts" application. To delete packages, first select an individual package or packages in the Eclipse package explorer as shown in Figure 4 and then hit the `Delete` key.

The packages to delete is provided per Maven module in the list below.

Client Module `contacts.client`

- In folder `src/main/java`
 - Delete package `org.eclipse.scout.contacts.client.helloworld`
 - Delete package `org.eclipse.scout.contacts.client.settings`
- In folder `src/test/java`
 - Delete package `org.eclipse.scout.contacts.client.helloworld`

Server Module `contacts.server`

- In folder `src/main/java`
 - Delete package `org.eclipse.scout.contacts.server.helloworld`

- In folder `src/test/java`
 - Delete package `org.eclipse.scout.contacts.server.helloworld`

Shared Module `contacts.shared`

- In folder `src/main/java`
 - Delete package `org.eclipse.scout.contacts.shared.helloworld`
- In folder `src/generated/java`
 - Delete package `org.eclipse.scout.contacts.shared.helloworld`

The deletion of these outlines results in a number of compile errors in classes `WorkOutline` and `Desktop`. All these errors will be resolved in the following two sections where we modify the two classes to our needs.

Changes to Class `WorkOutline`

Instead of adding a new "Contacts" outline to the application we reuse the generated code and rename the "Work" outline into "Contacts" outline. For this, we perform the following modifications to class `WorkOutline`.

- Rename the class package to `org.eclipse.scout.contacts.client.contact`
- Rename the class to `ContactOutline`
- Change the outline title to "Contacts"
- Change the outline icon to `Icons.CategoryBold`

To quickly find class `WorkOutline` we first open the *Open Type* dialog from the Eclipse IDE by hitting `Ctrl` + `Shift` + `T` and enter "workoutline" into the search field as shown in [Figure 5](#). In the result list, we select the desired class and click the **[OK]** button to open the file `WorkOutline.java` in the Java editor of the Eclipse IDE.



Figure 5. Use the Open Type dialog to quickly find java types in the Eclipse IDE.

We start with the package rename. To rename the package `org.eclipse.scout.contacts.client.work` to `org.eclipse.scout.contacts.client.contact` click into the word "work" of the package name and hit `Alt + Shift + R`. This opens the package rename dialog as shown in Figure 6 where we replace "work" by "contact" in the *New name* field. After pressing the **[Ok]** button Eclipse informs the programmer that the code modification may not be accurate as the resource has compile errors. This warning can be acknowledged by clicking **[Continue]**.



Figure 6. Use the Eclipse Rename Package dialog to rename a Java package.

In next step we rename class `WorkOutline` to `ContactOutline`. In the Java editor we can rename the class by clicking into the class identifier `WorkOutline` and hitting `Alt + Shift + R`. Inside the edit box we can then change the class name to `ContactOutline` and hit the `[Enter]` key to execute the change. If Eclipse shows a Rename Type dialog just hit button `[Continue]` to let Eclipse complete the rename operation. To get rid of the compile error in method `execCreateChildPages` we simply delete the content in the method body.

Next, we change the outline title in method `getConfiguredTitle` by replacing the string "Work" with "Contacts", setting the cursor at the end of the word "Contacts" and hitting `Ctrl + Space` to open the Scout content assist as shown in Figure 7.



Figure 7. Use the Scout content assist to add new translations.

To enter a new translated text we double click on the proposal *New text...* to open the Scout new entry wizard as shown in [Figure 8](#).



Figure 8. Use the Scout new entry wizard to add translated texts to the application.

As the last modification we change the return value of method `getConfiguredIconId` to value `Icons.CategoryBold` and end with the code shown in [Listing 1](#).

To conclude the modifications we must get rid of the faulty import statement. To do this we could in principle press `Ctrl + Shift + O` to organize the imports automatically. However, we want to register the organization of imports as a so-called “Save Action”. This can be done in the Eclipse preferences.

Invoke the top-level menu “Window/Preferences” and enter “Save Action” as a filter text to narrow down the tree on the left side of the preferences window. Then, chose the “Java/Editor/Save Actions” node and check the boxes “[x] Format Source Code” and “[x] Organize imports”. As you may have guessed, this formats the source code and organizes the import statements every time the user saves a Java class.

After having done so, perform a dummy modification in the class `ContactOutline` (e.g. add a space somewhere in the class body) and save the file. As a consequence, the faulty import statement at the beginning of the class file should have vanished and the class should show no compile errors anymore.

Listing 1. Initial implementation of class `ContactOutline`.

```
@ClassId("303c0267-3c99-4736-a7f5-3097c5e011b6")
public class ContactOutline extends AbstractOutline {

    @Override
    protected void execCreateChildPages(List<IPage<?>> pageList) {
    }

    @Override
    protected String getConfiguredTitle() {
        return TEXTS.get("Contacts");
    }

    @Override
    protected String getConfiguredIconId() {
        return Icons.CategoryBold;
    }
}
```

Changes to Class Desktop

The second class to adapt for the "Contacts" application is the `Desktop` class. This class is implemented exactly once in each Scout application and holds the available outlines and top level menus of the application in the form of inner classes.

For the "Contacts" application we adapt the initial implementation to have outline view buttons for the "Contacts" and "Search" outlines. The top level menus are then adapted to hold the menus "Quick Access", "Options" and a menu for the logged in user.

Start with opening the class in the Java editor using `Ctrl` + `Shift` + `T` to quickly access the class. In the source code of method `getConfiguredOutlines` remove `SettingsOutline.class` from the list of return values as shown in [Listing 2](#).

Listing 2. Method `getConfiguredOutlines` defines the outlines associated with the desktop of the application.

```
@Override
protected List<Class<? extends IOutline>> getConfiguredOutlines() {
    return CollectionUtility.<Class<? extends IOutline>> arrayList(ContactOutline
.class, SearchOutline.class);
}
```

Then, perform the following changes in class `Desktop`

- Delete the inner class `SettingOutlineViewButton`
- Delete the inner class `UserProfileMenu`.
- Rename inner class `WorkOutlineViewButton` to `ContactOutlineViewButton`
- Create a new inner class called `QuickAccessMenu` after the `SearchOutlineViewButton`. For this

navigate the cursor after the `SearchOutlineViewButton` class, press `Ctrl + Space` and select the `Menu` entry. Adapt the created code until it matches the template as shown in Listing 3. Note that in your implementation, the menu should extend `AbstractMenu` instead of `AbstractFormMenu` in contrast to what is shown in Listing 3. As a consequence, the method `getConfiguredForm` from Listing 3 is obsolete.

- Create another menu called `OptionsMenu` right after the newly created `QuickAccessMenu` according to Listing 3.
- Create a last menu called `UserMenu` after the `OptionsMenu` according to Listing 3.
- Delete the method `onThemeChanged`.
- Remove the statement in the body of the constructor.
- Note that it is not necessary to organize the imports since you have registered the organization of imports as a save action earlier in this tutorial.

At the end of these changes the inner class structure of class `Desktop` will look similar to the sample shown in Listing 3.

Listing 3. Structure of class `Desktop` with outline buttons and top level menus.

```
@ClassId("70eda4c8-5aed-4e61-85b4-6098edad8416")
public class Desktop extends AbstractDesktop {

    // outline buttons of the application
    @Order(1)
    @ClassId("9405937b-66e8-491a-831d-69adca724b90")
    public class ContactOutlineViewButton extends AbstractOutlineViewButton {
    }

    @Order(2)
    @ClassId("55febc84-ad6d-4ee8-9963-d1d40169a63a")
    public class SearchOutlineViewButton extends AbstractOutlineViewButton {
    }

    // top level menus for the header area of the application
    @Order(10)
    @ClassId("50df7a9d-dd3c-40a3-abc4-4619eff8d841")
    public class QuickAccessMenu extends AbstractMenu {

        @Override
        protected String getConfiguredText() {
            return TEXTS.get("QuickAccess");
        }
    }

    @Order(20)
    @ClassId("4fce42bf-85f9-4892-96a2-2e89e18eeae")
    public class OptionsMenu extends AbstractFormMenu<OptionsForm> { ①
```

```

@Override
protected String getConfiguredText() {
    return TEXTS.get("Options");
}

@Override
protected String getConfiguredIconId() {
    return Icons.Gear;
}

@Override
protected Class<OptionsForm> getConfiguredForm() {
    return OptionsForm.class;
}

}

@Order(30)
@ClassId("8dbfbe9d-0382-471a-ae43-3178f7a9e720")
public class UserMenu extends AbstractFormMenu<UserForm> { ②

    @Override
    protected String getConfiguredIconId() {
        return Icons.PersonSolid;
    }

    @Override
    protected Class<UserForm> getConfiguredForm() {
        return UserForm.class;
    }

}
}

```

- ① In your implementation `OptionsMenu` should extend `AbstractMenu` and the method `getConfiguredForm` should be deleted.
- ② In your implementation `UserMenu` should extend `AbstractMenu` and the method `getConfiguredForm` should be deleted.

What have we achieved?

In the first step of the "Contacts" tutorial we have created the initial project setup that will serve as the basis for all the following tutorial steps.

As the "Contacts" application is in a clean state you can now test the application using the following steps. The user interface of the application will now look as shown in [Figure 9](#).

- Activate the launch group `[webapp]` all to start the JS build, the frontend and the backend
- Open address <http://localhost:8082/> in your browser



Figure 9. The "Contacts" application at the end of tutorial step 1.

From the coding perspective we now have all necessary maven Modules for the "Contacts" application including Java package and class names to match with the complete Scout "Contacts" demo application. This point is important as it simplifies the comparison of intermediate stages of the tutorial application with the Scout demo application. The same is true for the user perspective: The layout of the current state of the tutorial matches with the complete "Contacts" sample application.

Adding the Person and Organization Page

In the second step of the Scout tutorial the components to display persons and organizations are added to the "Contacts" outline of the user interface of the Scout application. Specifically, a "Persons" page and an "Organizations" page are created and added to the navigation tree of the "Contacts" outline.

Database access and populating the pages with actual data from the database is not part of this section but will be covered in the next tutorial step ([Creating and Accessing the Database](#)).

The addition of the "Persons" page is described in detail in the sections listed below.

- Creating additional Packages ([Creating additional Packages](#))
- Creating the Country Lookup Call ([Creating the Country Lookup Call](#))
- Creating the Person Page ([Creating the Person Page](#))
- Adding Table Columns to the Page ([Adding Table Columns to the Page](#))
- Link the Person Page to the Contacts Outline ([Link the Person Page to the Contacts Outline](#))

The addition of the company page is described in [Adding the Company Page](#). Finally, the state of the "Contacts" application is summarized in [What have we achieved?](#).

Creating additional Packages

A substantial part of the "Contacts" application deals with persons. In addition to the "Persons" page we will also add a Scout form to enter/edit persons in a later tutorial step. For the "Contacts" application we use this fact to justify the addition of a specific Java package that will hold all classes related to persons. This person package can be created with the following steps.

- Open the "Contacts" Maven module `contacts.client` in the Eclipse Package Explorer
- Select the Java package `org.eclipse.scout.contacts.client` in folder `src/main/java`
- Press `Ctrl+N`, enter "package" into the search field
- Select the *Package* wizard in the proposal box and click *Next*
- Enter `org.eclipse.scout.contacts.client.person` and click *Finish* as shown in [Figure 10](#)
- Make sure the newly created person package is selected in the Eclipse Package Explorer

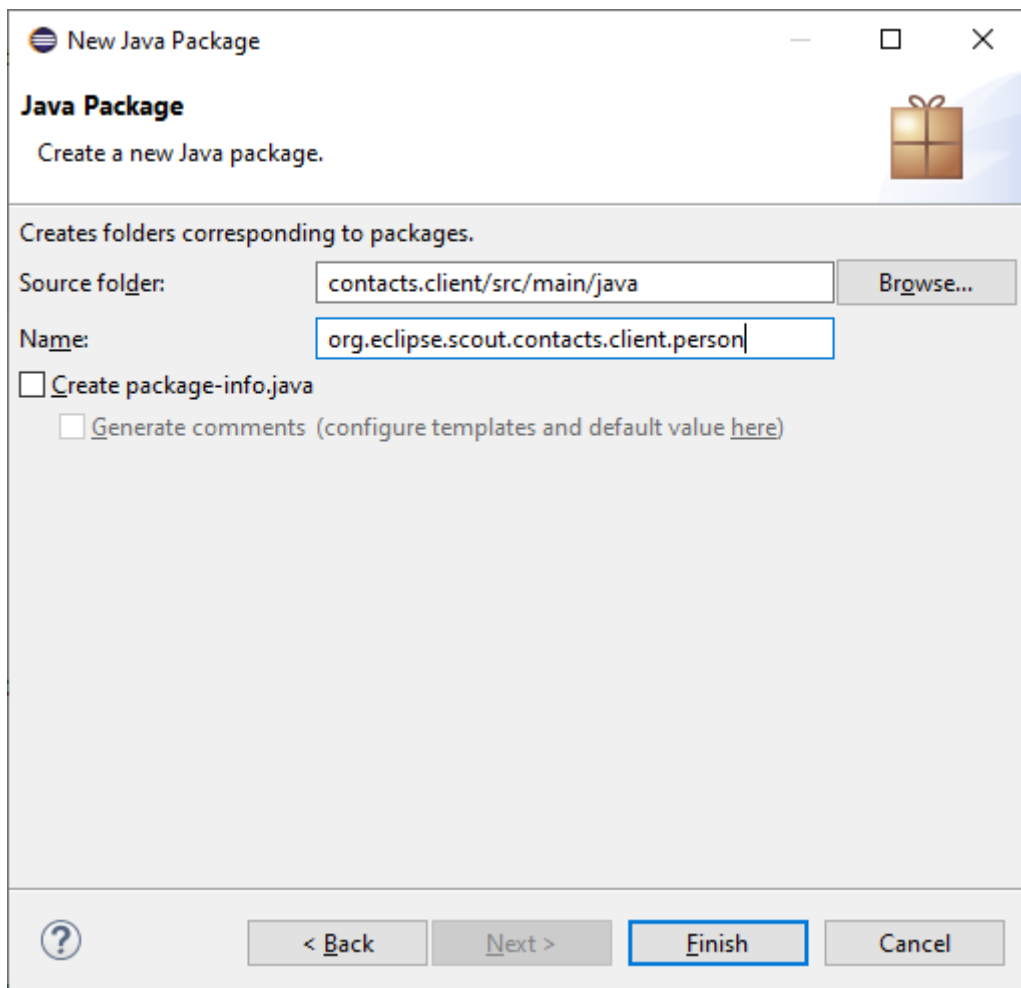


Figure 10. Add the person package to the "Contacts" application.

We will also need a separate package for organizations and some common elements.

- Add package `org.eclipse.scout.contacts.client.organization`
- Add package `org.eclipse.scout.contacts.client.common`

Creating the Country Lookup Call

The pages for the persons and the organizations will also display country information. To display country names we will be using a special column, that maps the country codes received from the backend application to translated country names. As the Java class `Locale` already contains both country codes and country names we can take advantage of this class and use it in a Scout local lookup call.

In package `org.eclipse.scout.contacts.client.common` create a new class `CountryLookupCall` according to the implementation provided in [Listing 4](#).

Listing 4. The Scout lookup call for countries. This lookup call will be used for the address field.

```
@ClassId("37736ea5-e861-43d8-a6bc-144dad3c208f")
public class CountryLookupCall extends LocalLookupCall<String> { ❶

    private static final long serialVersionUID = 1L;

    @Override
    protected List<LookupRow<String>> execCreateLookupRows() { ❷
        List<LookupRow<String>> rows = new ArrayList<>();

        for (String countryCode : Locale.getISOCountries()) {
            Locale country = new Locale("", countryCode);
            rows.add(new LookupRow<>(countryCode, country.getDisplayCountry())); ❸
        }

        return rows;
    }
}
```

- ❶ Makes the `CountryLookupCall` to work with key type `String`
- ❷ Defines the set of lookup rows to be used
- ❸ Add a row with the country code as key and the country name as display value

Creating the Person Page

In this section we create the Scout page that will be used to list all entered persons to the user of the "Contacts" application. Out-of-the box this page will support the sorting and filtering of all the persons. This "Persons" page is then added to the navigation tree below the "Contacts" outline.

We can now add the Scout person page as described below.

- Select the newly created package `org.eclipse.scout.contacts.client.person` in the Package Explorer
- Press `Ctrl+N`, enter "scout page" into the search field
- Select the *Scout Page* wizard in the proposal box and click *Next*
- Un-check the *Create an Abstract Super Page* option, as we don't need an additional abstract super class for our new page
- Enter `PersonTablePage` as the class name and click *Finish* as shown in [Figure 11](#)

Create a new Page

Create a new Page

New Page Details

Source Folder: contacts.client/src/main/java

Package: org.eclipse.scout.contacts.client.person

Name: PersonTablePage

Super Class: AbstractPageWithTable

PageData and Service Source Folders

Shared Source Folder: contacts.shared/src/main/java

Server Source Folder: contacts.server/src/main/java

Options

☐ Create an Abstract Super Page

< Back Next > Finish Cancel

Figure 11. Add the person page to the "Contacts" application.

The Scout *New Page Wizard* then creates an initial implementation for the `PersonTablePage` class very similar to the listing provided in [Listing 5](#) below.

Listing 5. Initial implementation of class *PersonTablePage*.

```
@PageData(PersonTablePageData.class)
@ClassId("23c10251-66b1-4bd6-a9d7-93c7d1aedede")
public class PersonTablePage extends AbstractPageWithTable<Table> {

    @Override
    protected String getConfiguredTitle() {
        return TEXTS.get("Persons"); ①
    }

    @Override
    protected void execLoadData(SearchFilter filter) {
        importPageData(BEANS.get(IPersonService.class)
            .getPersonTableData(filter, getOrganizationId())); ②
    }

    @Override ③
    protected boolean getConfiguredLeaf() {
        return true;
    }

    @ClassId("3fa1374b-9635-441b-b2f8-feb24b50740a")
    public class Table extends AbstractTable {
        // container class to hold columns and other elements for this table page ④
    }
}
```

Before we start to add the columns to the table of the page we need to do some minor adaptations to Listing 5.

- ① Specify the title "Persons" for the page using the Scout NLS tooling.
- ② You don't need to update method `execLoadData` to match this listing for now.
- ③ Add method `getConfiguredLeaf` to specify that the person page will not have any child pages.
- ④ We will add the columns in the next section of this tutorial.

We are now ready to populate the inner class `Table` of the person page with the columns to display various person attributes.

Adding Table Columns to the Page

Table pages are an important UI element of Scout applications as they frequently play a central role in the interactions of a user with the application. Out of the box table pages offer powerful options to sort, filter and re-arrange the data contained in the table. This functionality offers a good starting point to decide which columns to add to a table page.

To decide about the columns to add the following criteria have been useful in practice.

- Unique identifier of an element
- Attributes that are most frequently used in searches
- Category attributes that are useful for filtering
- Fewer columns are better



As the visible data of all users is held in the memory of the frontend server it is good practice to keep the number of columns as low as possible. Not taking this advice into account can substantially increase the memory footprint of the frontend server in production.

For the person page of the "Contacts" application we will add the following columns.

- **PersonId**: Hidden attribute of type string to hold the person key. Class name: `PersonIdColumn`
- **First Name**: String column. Class name: `FirstNameColumn`
- **Last Name**: String column. Class name: `LastNameColumn`
- **City**: String column. Class name: `CityColumn`
- **Country**: Smart column. Class name: `CountryColumn`
- **Phone**: String column, not visible per default. Class name: `PhoneColumn`
- **Mobile Phone**: String column, not visible per default. Class name: `MobileColumn`
- **Email**: String column, not visible per default. Class name: `EmailColumn`
- **Organization**: String column, not visible per default. Class name: `OrganizationColumn`



Make sure to use column class names exactly as indicated above. Working with different names is possible but requires additional work later in the tutorial when the data retrieved from the database is mapped to these column class names.

To add the first column `PersonIdColumn` we open class `PersonTablePage` in the Java editor and place the cursor inside of the body of the inner `Table` class. We then open the Scout content assist with `Ctrl` + `Space` and select the *Column* proposal as shown in [Figure 12](#).



Figure 12. Adding a column to the person page table.

In the first edit box we type "PersonId" as shown in Figure 13 and press `Enter`.



Figure 13. Adding a column to the person page table.

To configure this column as an invisible primary key we modify the newly created column class according to Listing 6.

Listing 6. Implementation of the person primary key column `PersonIdColumn`.

```
@Order(1)
@ClassId("1cbc9059-caef-4684-b013-bfa3bc0d0642")
public class PersonIdColumn extends AbstractStringColumn {

    @Override ①
    protected boolean getConfiguredDisplayable() {
        return false;
    }

    @Override ②
    protected boolean getConfiguredPrimaryKey() {
        return true;
    }
}
```

① Returning `false` here makes this column invisible. As this column will be excluded from the table control the user is not aware of the existence of this column.

② Returning `true` marks this attribute as a primary key (or part of a primary key)

We can now add the additional columns `FirstNameColumn`, `LastNameColumn`, `CityColumn` below. After entering the class name press `Tab` twice to move the cursor to the label text of the field. In the case of the first name enter "FirstName" and hit `Ctrl` + `Space` to open the wizard to add the translated text "First Name".

For these three columns the default implementation is fine and does not need any adaptations. Listing 7 below provides an example for this type of columns.

Listing 7. Implementation of the first name column.

```
@Order(2)
@ClassId("99df594a-6731-4757-a799-aacdbb4788d3")
public class FirstNameColumn extends AbstractStringColumn {

    @Override
    protected String getConfiguredHeaderText() {
        return TEXTS.get("FirstName");
    }

    @Override
    protected int getConfiguredWidth() {
        return 120;
    }
}
```

For column `CountryColumn` we will use a smart column. We again use `Ctrl` + `Space` to open the wizard and enter "Country" for the class name box and press `Tab` once and select `AbstractSmartColumn` as column type. Next we press `Tab` again to enter "Country" as the translated text.

In the created class `CountryColumn` we need to add the generic type parameter to the super class declaration (`AbstractSmartColumn<String>`) and add the method `getConfiguredLookupCall` according to [Listing 8](#).

Listing 8. Implementation of the country smart column.

```
@Order(5)
@ClassId("a39ad408-b5e5-4794-b86a-ddc13025862e")
public class CountryColumn extends AbstractSmartColumn<String> {

    @Override
    protected String getConfiguredHeaderText() {
        return TEXTS.get("Country");
    }

    @Override
    protected int getConfiguredWidth() {
        return 120;
    }

    @Override ①
    protected Class<? extends ILookupCall<String>> getConfiguredLookupCall() {
        return CountryLookupCall.class;
    }
}
```

- ① The configured lookup call is used to map country codes to the country names used in the user interface.

After the country column we add the four columns `PhoneColumn`, `MobileColumn`, `EmailColumn` and `OrganizationColumn` that are initially not visible in the user interface. As an example for such a column [Listing 9](#) is provided below.

Listing 9. Implementation of the (initially invisible) phone column.

```
@Order(6)
@ClassId("fa879506-d38c-46a6-990c-1f1ae4b74d4e")
public class PhoneColumn extends AbstractStringColumn {

    @Override
    protected String getConfiguredHeaderText() {
        return TEXTS.get("Phone");
    }

    @Override ①
    protected boolean getConfiguredVisible() {
        return false;
    }

    @Override
    protected int getConfiguredWidth() {
        return 120;
    }
}
```

- ① Returning `false` hides the column initially. Using the table control the user can then make this column visible in the user interface.



Use the Eclipse content assist to efficiently add method `getConfiguredVisible`. Place the cursor after method `getConfiguredHeaderText`, type "getConVis" and hit `Ctrl` + `Space`. Then select the proposal `getConfiguredVisible` with `Enter` and the method is inserted for you.

We have now created a person page with corresponding table columns. However, this new UI component is not yet visible in the user interface. What is missing is the link from the application's contacts outline class to the newly created `PersonTablePage` class. This is what we will do in the following section.

Link the Person Page to the Contacts Outline

In this section we add the person page to the contacts outline created during the initial project setup of the first step of this tutorial. This will make the person page visible in the navigation area below the "Contacts" outline.

For this we have to add a single line of code to method `execCreateChildPages` of class `ContactOutline` according to [Listing 10](#)

Listing 10. Adding the `PersonTable` to the `ContactOutline`.

```
@Override
protected void execCreateChildPages(List<IPage<?>> pagelist) {
    // pages to be shown in the navigation area of this outline
    pagelist.add(new PersonTablePage()); ①
}
```

- ① A new instance of the `PersonTable` is added to this outline. This makes the person page visible in the navigation area below the contacts outline.

The application is now in a state where we can restart the backend and the frontend server to verify our changes in the user interface.

Adding the Company Page

This section creates and adds a table page for organization to the "Contacts" outline. To create an organizations page the same steps are required as for the creation of the person page. The description is therefore kept on a higher level and in the text below only the main steps are described. Where appropriate, pointers are provided to the detailed descriptions for the creation of the person page.

- Add page `OrganizationTablePage` with title "Organizations" using the Scout new page wizard

Listing 11. Initial implementation of class `OrganizationTablePage`.

```
@PageData(OrganizationTablePageData.class)
@ClassId("18f7a78e-0dd0-4e4e-9234-99892bb4459f")
public class OrganizationTablePage extends AbstractPageWithTable<Table> {

    @Override
    protected String getConfiguredTitle() {
        return TEXTS.get("Organizations"); ①
    }

    @Override
    protected void execLoadData(SearchFilter filter) {
        importPageData(BEANS.get(IOrganizationService.class).getOrganizationTableData(
            filter));
    }

    @ClassId("54f3d730-7a62-462b-99ec-78fd1e6bb69d")
    public class Table extends AbstractTable {
        // container class to hold columns and other elements for this table page
    }
}
```

- ① Make sure to add a translated text entry for "Organizations" using the Scout NLS tooling

The implementation of class `OrganizationTablePage` using the Scout new page wizard then looks as

shown in [Listing 11](#).

As in the case of the person page you can now add the columns for the inner **Table** class. For the organization page add the columns according to the specification provided below.

- **OrganizationId**: Hidden attribute of type string to hold the organization key. Class name: **OrganizationIdColumn**
- **Name**: String column. Class name: **NameColumn**
- **City**: String column. Class name: **CityColumn**
- **Country**: Smart column. Class name: **CountryColumn**
- **Homepage**: String column, not visible per default. Class name: **HomepageColumn**

As in the case of the person page we have to add the newly created class **OrganizationTablePage** in method **execCreateChildPages** of the outline class **ContactOutline** as shown in [Listing 12](#).

Listing 12. Adding the OrganizationTablePage to the ContactOutline.

```
@Override
protected void execCreateChildPages(List<IPage<?>> pagelist) {
    // pages to be shown in the navigation area of this outline
    pagelist.add(new PersonTablePage()); ①
    pagelist.add(new OrganizationTablePage());
}
```

- ① The pages will appear in the user interface according to the order in which they are added to the outline.

What have we achieved?

In the second step of the "Contacts" tutorial we have created a person page and an organization page to display data of persons and organizations.

The "Contacts" application is in a clean state again and you can (re)start the backend and the frontend of the application and verify the user interface in your browser. The user interface should look like the screenshot provided in [Figure 14](#).



Figure 14. The "Contacts" application with the person and organization pages at the end of tutorial step 2.

When comparing the state of the "Contacts" tutorial application with the Scout demo application in [Figure 1](#) the main difference is the missing person data. Adding access to a database is the focus of the next tutorial step.

Creating and Accessing the Database

This tutorial step shows how Scout applications can interact with databases via JDBC. Due to the clean layering implemented in the "Contacts" application only the Scout backend server connects to the database. We therefore focus on the Scout backend in this part of the tutorial.

For the "Contacts" application we will work with a [Derby database](#). The choice of Derby is based on the fact that no additional installation is required and it is possible to work with in-memory databases.

We start this tutorial step with copying the classes that handle the database creation/access from the full "Contacts" demo application as described in [Adding the Infrastructure](#). The setup is then explained in the following sections.

- Scout Config Properties ([Scout Config Properties](#))
- The SQL Service and SQL Statements ([The SQL Service and SQL Statements](#))
- The Database Setup Service ([The Database Setup Service](#))

With the basic infrastructure in place we review the existing "Contacts" backend to answer the question [What is missing?](#). In [Fetching Organization and Person Data](#) we then add the missing pieces.

At the end of this tutorial step the "Contacts" backend server provides person and organization data to the frontend server as summarized in [What have we achieved?](#).

Adding the Infrastructure

This section describes the installation of the necessary components and classes that handle the database creation/access of the "Contacts" application.

To add the support for the Scout JDBC components and the Derby database we first need to declare the corresponding dependencies in the pom.xml file of the Maven server module. This can be done using the following steps.

- Expanding the Maven module `contacts.server` in the Eclipse Package Explorer
- Open the `pom.xml` file (use a double click on the file in the package explorer) and switch to the "pom.xml" tab in the Maven POM Editor.
- Add the database related dependencies according to [Listing 13](#)

Listing 13. The additional dependencies needed in the server pom.xml to use the derby database

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>org.eclipse.scout.contacts</groupId>
    <artifactId>org.eclipse.scout.contacts</artifactId>
    <version>22.0-SNAPSHOT</version>
    <relativePath>../org.eclipse.scout.contacts</relativePath>
  </parent>

  <artifactId>org.eclipse.scout.contacts.server</artifactId>

  <dependencies>
    <!-- database related dependencies --> ①
    <dependency>
      <groupId>org.eclipse.scout.rt</groupId>
      <artifactId>org.eclipse.scout.rt.server.jdbc</artifactId>
    </dependency>
    <!-- Derby 10.15.2.0 requires Java 9+ and org.apache.derby:derbytools module for
the JDBC driver -->
    <dependency>
      <groupId>org.apache.derby</groupId>
      <artifactId>derby</artifactId>
      <version>10.14.2.0</version>
    </dependency>
  </dependencies>
```

- ① Add the **derby** and the **org.eclipse.scout.rt.server.jdbc** dependencies to the pom.xml of your "Contacts" server module.

After adding the database dependencies to the server's pom.xml file we need to update all Maven server modules for the "Contacts" app. To do this, select the three modules **org.eclipse.scout.contacts.server.*** and hit **Alt + F5** as shown in [Figure 15](#). Start the update with **[OK]**.



Figure 15. Update the Maven server modules for the "Contacts" application.

The next step is to create the `org.eclipse.scout.contacts.server.sql` package.

- Expand folder `src/main/java` of Maven module `contacts.server`
- Select the existing package `org.eclipse.scout.contacts.server` and hit `Ctrl + N`
- This opens the dialog to select a wizard. Enter "package" into the search field
- Select the *New Java Package* wizard with a double click on the *Java Package* proposal
- Enter `org.eclipse.scout.contacts.server.sql` into the *Name* field of the wizard and click **[Finish]**

We are now ready to copy the classes related to the database infrastructure from the "Contacts" demo application to our tutorial workspace.

The simplest way to do this is to open a second Eclipse IDE with the workspace where you have

imported the Scout demo applications. If you have not done this yet go to the beginning of this tutorial [\[cha-large_example\]](#) and catch up now.

In the demo application workspace navigate to the same package `org.eclipse.scout.contacts.server.sql` and copy over all its classes. After copying these classes make sure that the structure of your server Maven module looks as shown in [Figure 16](#).



Figure 16. The copied database classes in the tutorial workspace.

The imported classes are described in the following sections. Additional information is provided where these classes are relying on Scout concepts that have not previously been introduced.

Scout Config Properties

Scout Config properties can greatly improve the flexibility of Scout applications. For the "Contacts" application this feature is used to keep its database setup configurable. Moving from a in-memory setup to a disk based database is then possible without any reprogramming.

The Scout backend (and frontend) applications initialize config properties from matching values found in file `config.properties`. For missing property values the default values defined in the config property classes are used.

In the case of the "Contacts" application the config property files are located in the subfolder `src/main/resources` of the Maven modules that specify the frontend and the backend application.

- Expand Maven module `contacts.server.app.dev`
- Expand subfolder `src/main/resources`
- Open file `config.properties` in the text editor
- Append all properties defined in [Listing 14](#) to the file

Listing 14. Properties relevant for creating and accessing the database.

```
### Database
contacts.database.jdbc.mappingName=jdbc:derby:memory:contacts-database
contacts.database.autocreate=true
contacts.database.autopopulate=true

### Application specific
contacts.superuser=system
```

These added property values then match the config properties defined in the class `DatabaseProperties` provided in [Listing 15](#). Remember that this is one of the database infrastructure classes we have copied before.

Listing 15. Typed properties for the "Contacts" application

```
public class DatabaseProperties {

    public static class DatabaseAutoCreateProperty extends AbstractBooleanConfigProperty
    {
        // defines default value and key

        @Override
        public Boolean getDefaultValue() {
            return Boolean.TRUE; ①
        }

        @Override
        public String getKey() {
            return "contacts.database.autocreate"; ②
        }

        @Override
        public String description() {
            return "Specifies if the contacts database should automatically be created if it
does not exist yet. The default value is true.";
        }
    }

    public static class DatabaseAutoPopulateProperty extends
AbstractBooleanConfigProperty {
        // defines default value and key
    }

    public static class JdbcMappingNameProperty extends AbstractStringConfigProperty {
        // defines default value and key
    }

    public static class SuperUserSubjectProperty extends AbstractSubjectConfigProperty {
        // defines default value and key
    }
}
```

① Defines the default value of the property that is used if the property is not defined in file `config.properties`

② Defines the key to be used in file `config.properties`

In the Scout framework config properties are always typed and need to implement interface `IConfigProperty`. For commonly used types Scout already provides classes. A boolean property may be created by extending Scout class `AbstractBooleanConfigProperty`.

Accessing the actual property values in the code is demonstrated in the next section.

The SQL Service and SQL Statements

Accessing databases with the Scout framework is implemented with SQL services that extend base class `AbstractSqlService`. As the "Contacts" application will be working with a Derby database we also need a Derby specific SQL service.

This is why we have copied over class `DerbySqlService`. The only project specific method is `getConfiguredJdbcMappingName` as implemented in [Listing 16](#).

Listing 16. The Derby SQL service to connect to the database

```
public class DerbySqlService extends AbstractDerbySqlService {
    @Override
    protected String getConfiguredJdbcMappingName() {
        String mappingName = CONFIG.getPropertyValue(JdbcMappingNameProperty.class);

        // add create attribute if we need to auto create the db
        if (CONFIG.getPropertyValue(DatabaseAutoCreateProperty.class)) {
            return mappingName + ";create=true"; ①
        }
        return mappingName;
    }
}
```

① Check the [Derby documentation](#) for additional attributes.

This listing also demonstrates how to use the config properties in the code. With the property values defined in the previous section the "Contacts" application is working with an in-memory database.

To change the setup to a disk based version, we would have to change the value for the property `contacts.database.jdbc.mappingName` from `jdbc:derby:memory:contacts-database` to `jdbc:derby:<path-to-dir>`. For a Windows box a concrete example could look like this: `jdbc:derby:c:\\derby\\contacts-database`.

Now we look at how the actual SQL statements of the "Contacts" application work. For our application all statements are collected into a single class. While there are many more options how to organize SQL and Java code this setup has its own advantages.

- Efficient maintenance as all SQL statements are located in a single place
- Code completion support in the Eclipse IDE when using the statements
- The setup is easy to explain

The SQL statements related to the database structure are provided in [Listing 17](#). The statements (or building blocks of statements) in interface `SQLs` are plain SQL in many cases. In the other cases the statement texts include Scout specific syntax extensions with `:` as a prefix character. Examples are `<identifier>` and `{<identifier>.<attribute>}`.

Listing 17. Interface SQLs with the SQL commands for the creation of the database tables.

```
public interface SQLs {

    String SELECT_TABLE_NAMES = ""
        + "SELECT    UPPER(tablename) "
        + "FROM      sys.systables "
        + "INTO      :result"; ①

    String ORGANIZATION_CREATE_TABLE = ""
        + "CREATE    TABLE ORGANIZATION "
        + "          (organization_id VARCHAR(64) NOT NULL CONSTRAINT ORGANIZATION_PK "
PRIMARY KEY, "
        + "          name VARCHAR(64), "
        + "          logo_url VARCHAR(512), "
        + "          url VARCHAR(64), "
        + "          street VARCHAR(64), "
        + "          city VARCHAR(64), "
        + "          country VARCHAR(2), "
        + "          phone VARCHAR(20), "
        + "          email VARCHAR(64), "
        + "          notes VARCHAR(1024)"
        + "          );

    String PERSON_CREATE_TABLE = ""
        + "CREATE    TABLE PERSON "
        + "          (person_id VARCHAR(64) NOT NULL CONSTRAINT PERSON_PK PRIMARY KEY, "
        + "          first_name VARCHAR(64), "
        + "          last_name VARCHAR(64), "
        + "          picture_url VARCHAR(512), "
        + "          date_of_birth DATE, "
        + "          gender VARCHAR(1), "
        + "          street VARCHAR(64), "
        + "          city VARCHAR(64), "
        + "          country VARCHAR(2), "
        + "          phone VARCHAR(20), "
        + "          mobile VARCHAR(20), "
        + "          email VARCHAR(64), "
        + "          organization_id VARCHAR(64), "
        + "          position VARCHAR(512), "
        + "          phone_work VARCHAR(20), "
        + "          email_work VARCHAR(64), "
        + "          notes VARCHAR(1024), "
        + "          CONSTRAINT ORGANIZATION_FK FOREIGN KEY (organization_id) REFERENCES "
ORGANIZATION (organization_id)"
        + "          );

}
```

① The syntax ':identifier' adds convenience and is supported by the Scout framework

The next section discusses how the components introduced above are used by the "Contacts"

application to create an initial "Contacts" database during the startup phase of the application.

The Database Setup Service

The database setup service is responsible to create the "Contacts" database during the startup of the application. In order to implement such a service, a number of Scout concepts are combined into class `DatabaseSetupService`.

- Access config properties using class `CONFIG`
- Executing SQL statements via class `SQL`
- Logging via class `LOG`
- Scout platform with the annotations `@ApplicationScoped`, `@CreateImmediately` and `@PostConstruct`

How these elements are used in class `DatabaseSetupService` is shown in [Listing 18](#). The actual creation of the "Contacts" database is performed by the method `autoCreateDatabase`.

At the time of the database creation no user is yet logged into the application. This is why we use a run context associated with the super user. The context is then used to execute the runnable that creates the organization and person tables.

Listing 18. Class `DatabaseSetupService` to create the database tables for the "Contacts" application.

```
@ApplicationScoped
@CreateImmediately
public class DatabaseSetupService implements IDataStoreService {
    private static final Logger LOG = LoggerFactory.getLogger(DatabaseSetupService.
class);

    @PostConstruct
    public void autoCreateDatabase() {
        if (CONFIG.getPropertyValue(DatabaseAutoCreateProperty.class)) {
            try {
                RunContext context = BEANS.get(SuperUserRunContextProducer.class).produce();
                IRunnable runnable = () -> {
                    createOrganizationTable();
                    createPersonTable();
                };

                context.run(runnable);
            }
            catch (RuntimeException e) {
                BEANS.get(ExceptionHandler.class).handle(e);
            }
        }
    }

    public void createOrganizationTable() {
        if (!getExistingTables().contains("ORGANIZATION")) {
            SQL.insert(SQLs.ORGANIZATION_CREATE_TABLE);
        }
    }
}
```

```

LOG.info("Database table 'ORGANIZATION' created");

    if (CONFIG.getPropertyValue(DatabaseAutoPopulateProperty.class)) {
        SQL.insert(SQLs.ORGANIZATION_INSERT_SAMPLE + SQLs.ORGANIZATION_VALUES_01);
        SQL.insert(SQLs.ORGANIZATION_INSERT_SAMPLE + SQLs.ORGANIZATION_VALUES_02);
        LOG.info("Database table 'ORGANIZATION' populated with sample data");
    }
}

public void createPersonTable() {
    if (!getExistingTables().contains("PERSON")) {
        SQL.insert(SQLs.PERSON_CREATE_TABLE);
        LOG.info("Database table 'PERSON' created");

        if (CONFIG.getPropertyValue(DatabaseAutoPopulateProperty.class)) {
            SQL.insert(SQLs.PERSON_INSERT_SAMPLE + SQLs.PERSON_VALUES_01);
            SQL.insert(SQLs.PERSON_INSERT_SAMPLE + SQLs.PERSON_VALUES_02);
            LOG.info("Database table 'PERSON' populated with sample data");
        }
    }
}

private Set<String> getExistingTables() {
    StringArrayHolder tables = new StringArrayHolder();
    SQL.selectInto(SQLs.SELECT_TABLE_NAMES, new NVPair("result", tables)); ①
    return CollectionUtility.hashSet(tables.getValue());
}
}

```

① The existing tables are stored in the `StringArrayHolder` object named "result".

The usage of `CONFIG` is already covered by the previous section. Introductions for `SQL`, `LOG` and the Scout platform annotations are provided below.

Logging

Scout uses the `SLF4J` framework for logging. For the actual implementation of the loggers Scout uses `Logback` per default. To use logging a local logger is first created using the `SLF4J LoggerFactory` class. Additional information regarding the logging configuration is provided below.

Executing SQL Statements

For the execution of SQL statements Scout provides the convenience class `SQL`. The various methods can be used with a simple SQL command as in `SQL.insert(mySqlCommand)` or using additional named objects as in `SQL.insertInto(mySqlCommand, myHolder)`. The Scout class `NVPair` is frequently used to create such named objects. Make sure that the identifiers (using the Scout `:` syntax) provided in the SQL commands always match with the names associated with the named objects.

Scout Platform

The Scout platform provides the basic infrastructure and a number of services to a Scout application. Services are represented by Scout beans that are registered at startup with the platform and created once they are needed. For class `DatabaseSetupService` we can use the Scout annotation `@ApplicationScoped` to register the service and to make sure that there will only be a single instance of this class. To force the creation of a bean `DatabaseSetupService` at startup time we also add Scout annotation `@CreateImmediately`. Finally, the annotation `@PostConstruct` executes our method `autoCreateDatabase` as soon as the `DatabaseSetupService` bean is created.

Changing the basic log level of an application is a frequently used scenario. As Scout is using Logback per default we can adapt the log level in the `logback.xml` configuration files as shown in [Listing 19](#). For the "Contacts" application these configuration files are located in folder `src/main/resources` of the Maven modules that define the frontend and the backend applications. More information regarding these configuration files is provided in the [Logback manual](#).

Listing 19. Setting the log level in the logback.xml configuration file.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<configuration>
  <root level="INFO"> ①
    <appender-ref ref="STDOUT" />
    <appender-ref ref="STDERR" />
  </root>
</configuration>
```

① The `level` attribute of the `<root>` element is used as the basic log level. Try "DEBUG" or "WARN" as alternative values.

What is missing?

This section reviews the backend infrastructure that has been created so far and identifies the pieces that are missing to fetch person and organization data to send it to the frontend server of the "Contacts" application.

During the creation of the person page and the organization page the Scout wizards created more than just Scout pages that are visible in the user interface. It also added corresponding classes in the shared module and the server module of the "Contacts" application.

The new page wizard basically added the complete round trip from the client (frontend server) to the server (backend server) and back. Using the organization page as an example, the setup created by the page wizard involves the following classes.

- Class `OrganizationTablePage` with method `execLoadData` in the client module
- The service interface `IOrganizationService` and class `OrganizationTablePageData` in the shared module
- Class `OrganizationService` with the method stub `getOrganizationTableData` in the server module

On the client side the server roundtrip is implemented in method `execLoadData` as shown in [Listing](#)

Listing 20. Accessing the "Contacts" backend server to fetch organization data.

```
@Override
protected void execLoadData(SearchFilter filter) {
    importPageData(BEANS.get(IOrganizationService.class).getOrganizationTableData
(filter));
}
```

This roundtrip between class `OrganizationTablePage` and `OrganizationService` works through the following steps.

1. `BEANS.get(IOrganizationService.class)` returns a reference to a client proxy service
2. Method `getOrganizationTableData(filter)` is executed on the corresponding server service
3. This method returns the organization data in the form of an `OrganizationTablePageData` object
4. Method `importPageData` transfers the data from the page data into the table of the user interface

On the server side fetching the data from the database will be implemented in class `OrganizationService` according to [Listing 21](#).

Listing 21. Method `getTableData` to access the database and map the data into a `pageData` object.

```
public class OrganizationService implements IOrganizationService {

    @Override
    public OrganizationTablePageData getOrganizationTableData(SearchFilter filter) {
        OrganizationTablePageData pageData = new OrganizationTablePageData();
        return pageData;
    }
}
```

In the next section we will implement the database access logic in the `getOrganizationTableData` methods of the server classes `OrganizationService` and `PersonService`.

Fetching Organization and Person Data

We are now ready to fetch data from the Derby database using the available infrastructure and the SQL statements prepared in class `SQLs`. For the implementation of method `getOrganizationTableData` in class `OrganizationService` we will use the two SQL snippets provided in [Listing 22](#).

Listing 22. Interface SQLs with the SQL to fetch the list of organizations with their attributes.

```
public interface SQLs {
    String ORGANIZATION_PAGE_SELECT = ""
        + "SELECT    organization_id, "
        + "           name, "
        + "           city, "
        + "           country, "
        + "           url "
        + "FROM      ORGANIZATION ";

    String ORGANIZATION_PAGE_DATA_SELECT_INT0 = ""
        + "INTO      :{page.organizationId}, " ①
        + "           :{page.name}, "
        + "           :{page.city}, "
        + "           :{page.country}, "
        + "           :{page.homepage}";
}
```

① The syntax '{identifier.attribute}' adds convenience to map SQL result sets to Scout page data objects.

Taking advantage of the SQL convenience offered by the Scout framework, we can add the missing functionality with two lines of code. See [Listing 23](#) for the full listing of method `getOrganizationTableData`. After adding the two additional lines, we update the imports of the classes with pressing `Ctrl` + `Shift` + `0`.

Listing 23. Method `getTableData` to access the database and map the data into a `pageData` object.

```
public class OrganizationService implements IOrganizationService {

    @Override
    public OrganizationTablePageData getOrganizationTableData(SearchFilter filter) {
        OrganizationTablePageData pageData = new OrganizationTablePageData();

        String sql = SQLs.ORGANIZATION_PAGE_SELECT + SQLs
            .ORGANIZATION_PAGE_DATA_SELECT_INT0; ①
        SQL.selectInto(sql, new NVPair("page", pageData)); ②

        return pageData;
    }
}
```

① Added line 1: Assembling of the SQL statement

② Added line 2: Fetching the data from the database and storing the result in `pageData`

Note that the identifier "page" in the `NVPair` object will be mapped to the same identifier used in the `ORGANIZATION_PAGE_DATA_SELECT_INT0` statement.

Finally, we have to also implement the loading of the person data in class `PersonService`. The

implementation of method `getPersonTableData` is provided in [Listing 24](#).

Listing 24. Method `getPersonTableData` to access the database and map the data into a page data object.

```
public class PersonService implements IPersonService {

    @Override
    public PersonTablePageData getPersonTableData(SearchFilter filter) {
        PersonTablePageData pageData = new PersonTablePageData();

        String sql = SQLs.PERSON_PAGE_SELECT + SQLs.PERSON_PAGE_DATA_SELECT_INTRO;
        SQL.selectInto(sql, new NVPair("page", pageData));

        return pageData;
    }
}
```

What have we achieved?

In the third step of the "Contacts" tutorial we have added the infrastructure to work with a Derby database. The infrastructure is used to create and populate the initial database. In addition person and organization data is now fetched from the database on the "Contacts" backend server and handed to the "Contacts" frontend server via a page data object.

The "Contacts" application is in a clean state again and you can (re)start the backend and the frontend of the application and verify the result in your browser. Person and company data is now visible in the user interface as shown in [Figure 17](#).



The screenshot shows a web browser window titled "Contacts Application" with the URL `localhost:8082/?dl=outline-11770&i=contacts`. The interface has a dark blue sidebar on the left with a search icon and a list of "Persons" and "Organizations". The main content area displays a table of person data. The table has columns for "First name", "Last name", "City", "Country", and "Organization". The data is as follows:

First name	Last name	City	Country	Organization
Alice		Daresbury, ...	United Kingdom	org01
Rabbit	White	Daresbury, ...	United Kingdom	org01
Gegor	Bauer	Aarau	Switzerland	org02
Alexandre	Schroder	Zürich	Switzerland	org02
André	Wegmüller	Walkringen	Switzerland	org02
Catherine	Crowden	Schoenried	Switzerland	org02
Cédric	Amstalden	Stands	Switzerland	org02

At the bottom of the table, there is a status bar showing "29 rows loaded", "Reload data", and "No row selected Select all".

Figure 17. The "Contacts" application displaying person data at the end of tutorial step 3.

Adding a Form to Create/Edit Persons

In this tutorial step we add the Scout forms that are used to create and edit persons and organizations in the user interface. This tutorial step also provides an introduction into the design and implementation of complex form layouts with the Scout framework.

Before we start with the actual implementation of the form [Designing the Person Form](#) provides an introduction to the layouting concepts of the Scout framework. Based on this information we design a hierarchical form layout for the person form and can then dive into the creation of the person form.

- Implementing the Form ([Implementing the Form](#))
- Adding a Gender Code Type ([Adding a Gender Code Type](#))
- Adding Form Fields ([Adding Form Fields](#))
- Person Form Handler and Person Service ([Person Form Handler and Person Service](#))

The tutorial step concludes with a summary in [What have we achieved?](#).

Designing the Person Form

We start with the sketch of the form layout as shown in [Figure 18](#).

Figure 18. A sketch of the target layout for the person form.

The upper half of the form shows a picture of the person and contains some primary attributes such as first name and the gender of the person.

The lower half of the form contains tab boxes. A "Contact Info" tab provides contact details of the person and adding notes for the person in the form of free text is possible in the "Notes" tab.

Figure 19 below shows how the sketched form can fit with the logical grid layout of the Scout framework. Scout containers have two columns (indicated in red) per default and as many rows (indicated in yellow) as needed.

	First Name	<input type="text"/>	
	Last Name	<input type="text"/>	
	Date of Birth	<input type="text" value="/ /"/>	
	Gender	<input type="radio"/> Male <input type="radio"/> Female	
<div> <div>Contact Info</div> <div>Work</div> <div>Notes</div> </div>			
Street	<input type="text"/>	Phone	<input type="text"/>
Location	<input type="text" value="City"/> <input type="text" value="Country"/>	Mobile	<input type="text"/>
		Email	<input type="text"/>

Figure 19. Logical columns and rows of the Scout form layout. Scout containers have two columns per default.

Individual form fields consist of a label part and a field part and occupy a single cell in the logical grid. Examples for fields using the default configuration are the first name field or the email field. When needed, fields can be configured to occupy several columns or rows. An example for this case is the image field that will hold the picture of the person. This field is configured to occupy 5 logical rows.

With Scout's container widgets such as group boxes, tab boxes and sequence boxes complex layouts can be achieved. Containers provide a lot of flexibility as these widgets can be nested hierarchically as shown in [Figure 20](#)

Figure 20. The hierarchical organization of the form including Scout group boxes, tab boxes and a sequence box.

The sketch above details the organization of the container components to match the desired layout for the person form. The different container widgets can all be used with their default settings except for the address box.

For the address box we will have to modify its size and its inner organization. As group boxes occupy two columns per default we will need to reduce the width of the address box to a single column. The second change is to the inner layout of the address box. To force the location box to come below the street field we have to change the inner layout of the group box to a single column as well. Otherwise, the location box would be shown next to the street field.

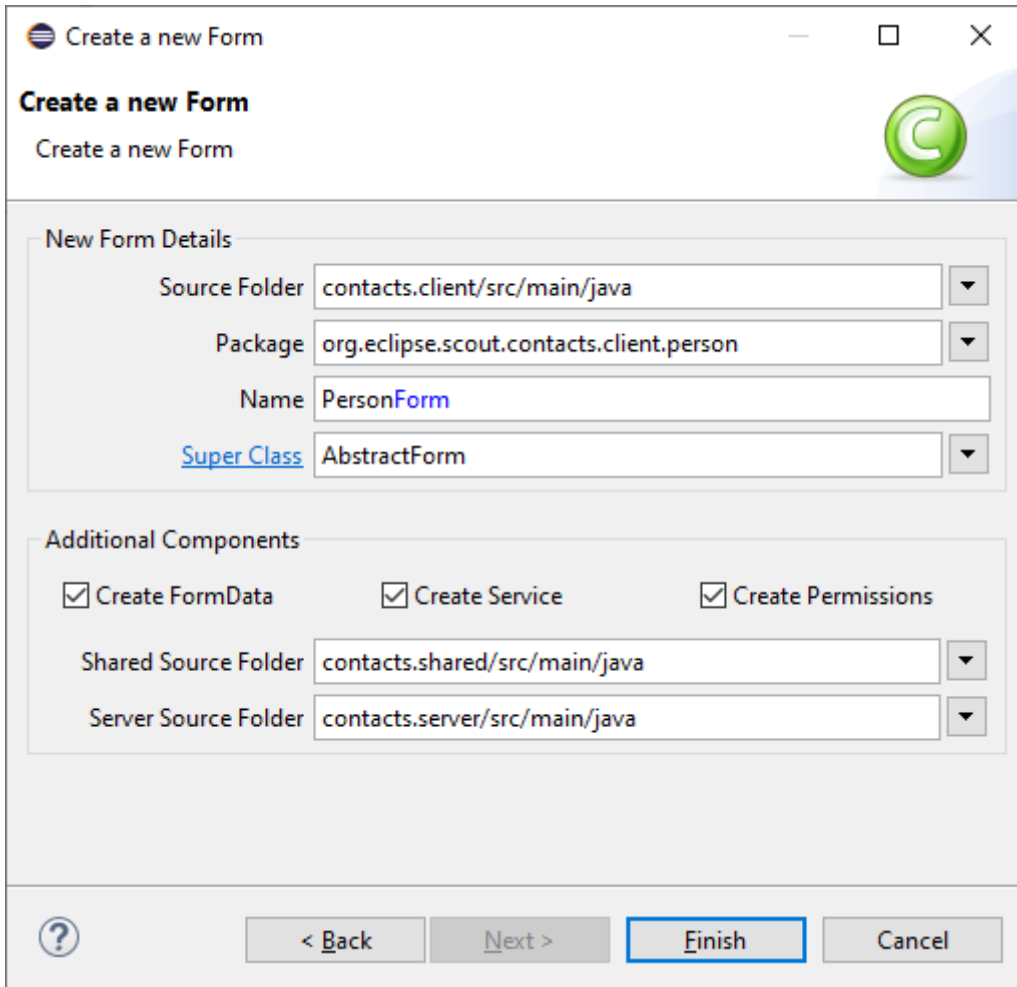
In the next section we will start to implement the person form with the layout described above.

Implementing the Form

In this section we implement the person form with its container widgets as described in the previous section. To be able to use the form to create and edit persons we will add "New" and "Edit" context menus to the table in the person page. Finally we will also add a "Create Person" entry to the the "Quick Access" top level menu of the application.

Start the form creation with the Scout new form wizard following the steps listed below.

1. Expand the Maven module `contacts.client` in the Eclipse package explorer
2. Select package `org.eclipse.scout.contacts.client.person` in folder `src/main/java`
3. Press `Ctrl + N` and enter "form" into the search field of the wizard selection dialog
4. Select the *Scout Form* proposal and click the **[Next]** button
5. Enter "Person" into the *Name* and verify that the field contents match [Figure 21](#)
6. Click **[Finish]** to start the creation of the form and its related components



The screenshot shows the 'Create a new Form' dialog box. The title bar says 'Create a new Form'. Below the title bar, there's a sub-header 'Create a new Form' and a green circular icon with a white 'C'. The main area is divided into two sections: 'New Form Details' and 'Additional Components'. In 'New Form Details', there are four fields: 'Source Folder' (contacts.client/src/main/java), 'Package' (org.eclipse.scout.contacts.client.person), 'Name' (PersonForm), and 'Super Class' (AbstractForm). In 'Additional Components', there are three checked checkboxes: 'Create FormData', 'Create Service', and 'Create Permissions'. Below these are two more fields: 'Shared Source Folder' (contacts.shared/src/main/java) and 'Server Source Folder' (contacts.server/src/main/java). At the bottom, there are four buttons: '?', '< Back', 'Next >', and 'Finish' (which is highlighted with a blue border), and 'Cancel'.

Figure 21. Use the New Scout Form to create the person form.

Now open the newly created class `PersonForm` in the Java editor and perform the changes listed below as shown in [Listing 25](#).

- Add property `personId` with the corresponding getter and setter methods
- Add method `computeExclusiveKey`
- Add method `getConfiguredDisplayHint`
- Verify the translated text entry in method `getConfiguredTitle`

Listing 25. Add `getConfiguredDisplayHint` and the methods related to the person's primary key.

```
@ClassId("1cde38c1-da32-4fdd-92e7-28d82a5d7bf9")
@FormData(value = PersonFormData.class, sdkCommand = SdkCommand.CREATE) ①
public class PersonForm extends AbstractForm {

    // represents the person's primary key
    private String personId;

    @FormData ②
    public String getPersonId() {
        return personId;
    }

    @FormData ②
    public void setPersonId(String personId) {
        this.personId = personId;
    }

    @Override
    public Object computeExclusiveKey() { ③
        return getPersonId();
    }

    @Override
    protected int getConfiguredDisplayHint() { ④
        return IForm.DISPLAY_HINT_VIEW;
    }

    @Override
    protected String getConfiguredTitle() {
        return TEXTS.get("Person");
    }
}
```

- ① Links the form with its form data class `PersonFormData`.
- ② The annotation `@FormData` on the getter and setter method define the `personId` as a property that will be included in the form data.
- ③ The object returned by this method is used by the framework to verify if a specific entity is already opened in some other form.
- ④ Configure this form to be opened in the view mode. Views are opened in the bench area of the user interface.

We are now going to add the layout containers according to Listing 26. First add class `GeneralBox` using the Scout content assist selecting the *Group Box* proposal. Delete method `getConfiguredLabel`, as we are only using this group box to organize fields.

After the general box add a tab box container class by choosing the *Tab Box* proposal in the Scout content assist. Inside of class `DetailsBox` create the individual tab containers "Contact Info", "Work"

and "Notes" as inner classes of the details box according to [Listing 26](#).

Listing 26. The layouting structure of the person form using Scout container widgets.

```
public class PersonForm extends AbstractForm {

    @Order(10)
    @ClassId("27a040ac-eac5-47c6-a826-572633b9d4ef")
    public class MainBox extends AbstractGroupBox { ❶

        @Order(10)
        @ClassId("08832a97-8845-4ff4-8dfd-c29366c22742")
        public class GeneralBox extends AbstractGroupBox { ❷
        }

        @Order(20)
        @ClassId("3469046e-ee95-4e86-b0c9-a8ed01fbf664")
        public class DetailsBox extends AbstractTabBox { ❸

            @Order(10)
            @ClassId("2081b483-3d6e-4239-b7da-b6e2d2aa3b7a")
            public class ContactInfoBox extends AbstractGroupBox { ❹

                @Order(10)
                @ClassId("736450dd-ba89-43cd-ba52-bcd31196b462")
                public class AddressBox extends AbstractGroupBox {
                }
            }

            @Order(20)
            @ClassId("8e18a673-aca5-44a2-898f-60a744e4467a")
            public class WorkBox extends AbstractGroupBox {
            }

            @Order(30)
            @ClassId("fcb5b155-2c89-4ef8-9a96-ac41e9032107")
            public class NotesBox extends AbstractGroupBox {
            }
        }

        @Order(30)
        @ClassId("e54548b8-601e-41a4-842c-db25b5f1cad1")
        public class OkButton extends AbstractOkButton {
        }

        @Order(40)
        @ClassId("26612eb9-1832-4284-ac5a-9f450dc7ff9b")
        public class CancelButton extends AbstractCancelButton {
        }
    }
}
```

❶ Every Scout form has a class `MainBox`. It contains all visible UI components.

- ② The `GeneralBox` will hold the picture field, first name and last names, the date of birth and the gender.
- ③ The `DetailsBox` tab box will contain the various tabs implemented in inner group boxes.
- ④ The containers `ContactInfoBox`, `WorkBox` and `Notes` represent the three tabs of the tab box.

To actually open the person form the form needs to be integrated in the user interface. In Scout application forms are typically opened by first selecting a specific row in a page and then using a context menu. For the "Contacts" application we will follow this pattern too.

Open class `PersonTablePage` in the Java editor and create the context menus "New" and "Edit" in the inner class `Table` according to [Listing 27](#).

Listing 27. The page context menus to open the person form.

```
@PageData(PersonTablePageData.class)
@ClassId("23c10251-66b1-4bd6-a9d7-93c7d1aedede")
public class PersonTablePage extends AbstractPageWithTable<Table> {

    @ClassId("3fa1374b-9635-441b-b2f8-feb24b50740a")
    public class Table extends AbstractTable {

        @Override
        protected Class<? extends IMenu> getConfiguredDefaultMenu() { ①
            return EditMenu.class;
        }

        @Order(10)
        @ClassId("4a8f5e0e-6eb8-4296-8ad7-012151f572f2")
        public class EditMenu extends AbstractMenu {
            @Override
            protected String getConfiguredText() {
                return TEXTS.get("Edit");
            }

            @Override
            protected void execAction() {
                PersonForm form = new PersonForm();
                form.setPersonId(getPersonIdColumn().getSelectedValue()); ②
                form.addFormListener(new PersonFormListener());
                // start the form using its modify handler
                form.startModify();
            }
        }

        @Order(20)
        @ClassId("8ac358f2-de17-4b2b-93f3-73e21a7415d8")
        public class NewMenu extends AbstractMenu {

            @Override
            protected String getConfiguredText() {
```

```

        return TEXTS.get("New");
    }

    @Override
    protected Set<? extends IMenuType> getConfiguredMenuTypes() { ③
        return CollectionUtility.<IMenuType> hashSet(
            TableMenuType.EmptySpace, TableMenuType.SingleSelection);
    }

    @Override
    protected void execAction() {
        PersonForm form = new PersonForm();
        form.addFormListener(new PersonFormListener());
        // start the form using its new handler
        form.startNew();
    }
}

private class PersonFormListener implements FormListener {

    @Override
    public void formChanged(FormEvent e) {
        // reload page to reflect new/changed data after saving any changes
        if (FormEvent.TYPE_CLOSED == e.getType() && e.getForm().isFormStored()) {
            reloadPage();
        }
    }
}
}
}
}
}

```

- ① This action gets executed when the user presses on a table row or double clicks on a table row.
- ② Transfer the primary key of the selected person row to the person form.
- ③ Including `TableMenuType.EmptySpace` in the return value activates the "New" menu even when no row is selected.

In addition to the context menus defined for the person page we also add a "Create new person" menu on the desktop under the "Quick Access" top level menu. To do this open class `Desktop` in the Java editor and navigate to the inner class `QuickAccessMenu`. We can then add a `NewPersonMenu` using the Scout content assist and selecting the `Menu` proposal entry. The final implementation for the "Create new person" menu is provided in [Listing 28](#).

Listing 28. The "Create new person" menu on the desktop.

```
@ClassId("70eda4c8-5aed-4e61-85b4-6098edad8416")
public class Desktop extends AbstractDesktop {

    @Order(10)
    @ClassId("50df7a9d-dd3c-40a3-abc4-4619eff8d841")
    public class QuickAccessMenu extends AbstractMenu {

        @Override
        protected String getConfiguredText() {
            return TEXTS.get("QuickAccess");
        }

        @Order(10)
        @ClassId("effb3b69-f488-4aed-8923-d430a5f1fd97")
        public class NewPersonMenu extends AbstractMenu {

            @Override
            protected String getConfiguredText() {
                return TEXTS.get("NewPersonMenu");
            }

            @Override
            protected void execAction() {
                new PersonForm().startNew();
            }
        }
    }
}
```

We have now created the initial implementation of the person form including context menus to open the form from the person page and the "Quick Access" top level menu of the "Contacts" application. At this point it is already possible to verify that the person form can be opened on the user interface via the context menus. A screenshot of the current state is shown in [Figure 22](#).



Figure 22. The initial person form and the top level menu "Create new person".

This initial implementation of the person form is also ready to add the individual form fields into the container boxes. For the fields of the person form we can directly extend the abstract form field classes offered by the Scout framework. Only for the implementation of the gender field we need a Scout code type that represents the possible values for the radio buttons.

Adding a Gender Code Type

In this section we will add a gender code type for the "Contacts" application. As code types can be used for the specification of the options of a radio button group, we will be able to implement the gender field by providing a reference to the code type. To keep things simple, the gender code type will contain a "Male" code and a "Female" code.

Code types are frequently used in both the frontend and the backend of an application. This implies that code type classes need to be implemented in the application's shared module. As the gender code type is related to persons we will implement this class in the person package.

Follow the steps described below to create the gender code type.

1. Expand the Maven module `contacts.shared` in the Eclipse package explorer
2. Select package `org.eclipse.scout.contacts.shared.person` in folder `src/main/java`
3. Press `Ctrl + N` and enter "code" into the search field of the wizard selection dialog
4. Select the *Scout CodeType* proposal and click the **[Next]** button
5. Enter "Gender" into the *Name* field and use the type `String` for the first and second type argument according to [Figure 23](#)

Create a new CodeType

Create a new CodeType

Create a new CodeType

New CodeType Details

Source Folder: contacts.shared/src/main/java

Package: org.eclipse.scout.contacts.shared.person

Name: GenderCodeType

Super Class: AbstractCodeType

Type Arguments

First Argument: String

Second Argument: String

Third Argument:

? < Back Next > Finish Cancel

Figure 23. Create the gender code using the Scout new code wizard.

Now click button **[Finish]** to start the wizard. Then, open the newly created class `GenderCodeType` in the Java editor and set the `ID` constant to "Gender". The created class will then look like Listing 29 except for the missing inner code classes. We will add these inner codes as the next step.

Listing 29. The Scout code type to represent the gender of a person. This code type will be used for the gender field.

```
@ClassId("bbe8fae2-4923-42bc-9745-3bb3ef592b12")
public class GenderCodeType extends AbstractCodeType<String, String> {

    private static final long serialVersionUID = 1L;
    public static final String ID = "Gender";

    @Override
    public String getId() {
        return ID;
    }

    @Order(1000)
    @ClassId("8893e1e4-7b6c-46c2-8c84-42c914ec29d5")
    public static class MaleCode extends AbstractCode<String> {

        private static final long serialVersionUID = 1L;
        public static final String ID = "M";

        @Override
        protected String getConfiguredText() {
            return TEXTS.get("Male");
        }

        @Override
        public String getId() {
            return ID;
        }
    }

    @Order(2000)
    @ClassId("23e1540e-2914-401f-9f42-e409ac2fb605")
    public static class FemaleCode extends AbstractCode<String> {

        private static final long serialVersionUID = 1L;
        public static final String ID = "F";

        @Override
        protected String getConfiguredText() {
            return TEXTS.get("Female");
        }

        @Override
        public String getId() {
            return ID;
        }
    }
}
```

To add an inner class `MaleCode` code to the gender code type perform the steps below.

1. Press `Ctrl` + `Space` and select the *Code* proposal with a double click
2. Enter "Male" into the first box to be used in the `MaleCode` class name
3. Tab to the value for the `ID` constant and set it to "M"
4. Tab to the value in `TEXTS.get` and add "Male" and its translated text
5. Hit `Enter` to finish

Then repeat the steps above for the female code.

Adding Form Fields

In this section we will add the form fields to the layout containers of the person form. We will start with filling the general box with the picture field, followed by the other fields in the upper part of the person form. Finally, we fill the individual tab boxes into the details box in the lower part of the person form.

As the first field we add the field that will show the picture of the person to the `GeneralBox` container.

1. Open class `PersonForm` in the Java editor
2. Place the cursor in the body of the inner class `GeneralBox`.
3. Copy the code provided in [Listing 30](#) into the general box.
4. Add for each field a getter method above the `MainBox` (where all other getters are). Alternatively you could use the SDK to create both fields (including getters) and add the code from [Listing 30](#).

Listing 30. The picture field for the person form.

```
@Order(10)
@ClassId("617ffd40-0d69-4d02-b4f8-90c28c68c6ce")
public class PictureUrlField extends AbstractStringField {

    @Override ①
    protected boolean getConfiguredVisible() {
        return false;
    }
}

@Order(20)
@ClassId("6366a23e-f8ba-4b50-b814-202e63daffc8")
public class PictureField extends AbstractImageField {

    @Override ②
    protected Class<PictureUrlField> getConfiguredMasterField() {
        return PictureUrlField.class;
    }

    @Override ③
    protected void execChangedMasterValue(Object newMasterValue) {
        updateImage((String) newMasterValue);
    }

    @Override
    protected boolean getConfiguredLabelVisible() {
        return false;
    }

    @Override
    protected int getConfiguredGridH() {
        return 5;
    }

    @Override
    protected boolean getConfiguredAutoFit() {
        return true;
    }

    @Override
    protected String getConfiguredImageId() {
        return Icons.User;
    }

    protected void updateImage(String url) {
        setImageUrl(url);
    }
}
```

- ① Sets the field invisible. An invisible field does not occupy space in the user interface.
- ② Declares `PictureUrlField` as the master field of the picture field.
- ③ This method will be called when the value of the master field has changed.

Using the combination of the `PictureField` and `PictureUrlField` as its master field has two benefits. First, having a field that contains the the URL makes sure that this information is also stored in the form data and second, the method `execChangedMasterValue` can then be used to trigger the refresh of the actual picture when the picture URL is changed.

For security reasons, the browser is not allowed to load content from other servers by default. For our demo images, we add a well-considered exception. Open the `config.properties` file of your UI server project and make sure it contains the following line:

Listing 31. Content Security Policy Configuration (config.properties)

```
scout.cspDirective[img-src]='self' www.gravatar.com wiki.eclipse.org  
upload.wikimedia.org
```

The remaining fields for the general box can then be added using the Scout content assist or by copying [Listing 32](#) into the code below the picture field, again not forgetting the getters above the `MainBox`.

Listing 32. The other fields in the general box.

```
@Order(30)
@ClassId("359be835-439f-456e-9b0d-c832b034a298")
public class FirstNameField extends AbstractStringField {

    @Override
    protected String getConfiguredLabel() {
        return TEXTS.get("FirstName");
    }
}

@Order(40)
@ClassId("8679ade5-21fb-470e-8f00-13bd15199101")
public class LastNameField extends AbstractStringField {

    @Override
    protected String getConfiguredLabel() {
        return TEXTS.get("LastName");
    }
}

@Order(50)
@ClassId("7c602360-9daa-44b8-abb6-94ccf9b9db59")
public class DateOfBirthField extends AbstractDateField {

    @Override
    protected String getConfiguredLabel() {
        return TEXTS.get("DateOfBirth");
    }
}

@Order(60)
@ClassId("b9d0593e-3938-4f97-bdca-fdb6a1ce1d77")
public class GenderGroup extends AbstractRadioButtonsGroup<String> {

    @Override
    protected String getConfiguredLabel() {
        return TEXTS.get("Gender");
    }

    @Override ①
    protected Class<? extends ICodeType<?, String>> getConfiguredCodeType() {
        return GenderCodeType.class;
    }
}
```

- ① The codes defined in `GenderCodeType` will be used to determine the actual radio buttons to add to the gender field.

Whenever we add several fields to a Scout container field the individual fields will be displayed

according to their order specified by the `@Order` annotation in the source code. Using the default two column layout, the Scout layouting engine uses the first fields to fill up the first column before the remaining fields are assigned to the second column. In general the Scout layouting engine tries to balance the number of fields over all available columns. For the general box this rule has the effect that the picture field (this is the first field according to its order value) is assigned to the left column and all other fields are assigned to the right column.

After having added all the fields to the general box of the person form we can now fill the individual tabs of the `DetailsBox` container. We start with adding the content to the tabs "Work" and "Notes" as described below.

Now add the string fields listed below to the "Work" tab as inner classes of the container field `WorkBox`. Use the Scout content assist to add the fields and select *String Field* as the type of each field.

- Class `PositionField`, using label "Position"
- Class `OrganizationField`, using label "Organization"
- Class `PhoneWorkField`, using label "Phone"
- Class `EmailWorkField`, using label "E-Mail"

The "Notes" tab represented by the container field `NotesBox` only contains a single string field. This field will not need a label, span 4 rows of the logical grid and hold a multi line text according to [Listing 33](#).

Listing 33. The notes tab box with its multi line text field.

```
@Order(30)
@ClassId("fcb5b155-2c89-4ef8-9a96-ac41e9032107")
public class NotesBox extends AbstractGroupBox {

    @Override
    protected String getConfiguredLabel() {
        return TEXTS.get("Notes");
    }

    @Order(10)
    @ClassId("ce791f14-fca6-4f11-8476-89cbf905eb2e")
    public class NotesField extends AbstractStringField {

        @Override
        protected int getConfiguredGridH() {
            return 4;
        }

        @Override
        protected boolean getConfiguredLabelVisible() {
            return false;
        }

        @Override
        protected boolean getConfiguredMultilineText() {
            return true;
        }
    }
}
```

Next is the implementation of the address box in the "Contact Info" tab. The address box is realized as a single column group box that holds a street field, a city field and a country field. According to the form layout defined in [Designing the Person Form](#) the city field and the country field will be located on the same logical row and in the same cell of the logical grid.

In the Scout default layout each form field uses up a single cell of the logical grid. Whenever we like to be more economical with the space occupied by several fields, we can work with a Scout sequence box. Inner fields of a sequence box will be arranged on a single row from left to right and the spacing between the inner fields will be minimal.

Taking advantage of these properties we implement the location box as a sequence field according to [Listing 34](#). To further optimize screen real estate we also switch to on-field labels for the city field and the country field.

Listing 34. The content of the address box.

```
@Order(10)
```

```

@ClassId("736450dd-ba89-43cd-ba52-bcd31196b462")
public class AddressBox extends AbstractGroupBox {

    @Override
    protected boolean getConfiguredBorderVisible() {
        return false;
    }

    @Override
    protected int getConfiguredGridH() { ①
        return 3;
    }

    @Override
    protected int getConfiguredGridW() { ①
        return 1;
    }

    @Override
    protected int getConfiguredGridColumnCount() { ②
        return 1;
    }
    @Order(10)
    @ClassId("a9137ad1-af9d-4fef-a69d-3e3d9ce48f21")
    public class StreetField extends AbstractStringField {

        @Override
        protected String getConfiguredLabel() {
            return TEXTS.get("Street");
        }
    }

    // use a sequence box for horizontal layout ③
    @Order(20)
    @ClassId("a278333c-057e-4c1d-a442-0c1dd62fdca7")
    public class LocationBox extends AbstractSequenceBox {

        @Override
        protected String getConfiguredLabel() {
            return TEXTS.get("Location");
        }

        @Override
        protected boolean getConfiguredAutoCheckFromTo() { ④
            return false;
        }

        @Order(10)
        @ClassId("3ea6ac2a-976e-4c7f-b04b-ec0d7d1ae5ec")
        public class CityField extends AbstractStringField {

```

```

@Override
protected String getConfiguredLabel() {
    return TEXTS.get("City");
}

@Override
protected byte getConfiguredLabelPosition() {
    return LABEL_POSITION_ON_FIELD; ⑤
}
}

@Override(20)
@ClassId("d4dfce4f-019b-4a61-ba78-347ef67cf80f")
public class CountryField extends AbstractSmartField<String> {

    @Override
    protected String getConfiguredLabel() {
        return TEXTS.get("Country");
    }

    @Override
    protected byte getConfiguredLabelPosition() {
        return LABEL_POSITION_ON_FIELD;
    }

    @Override
    protected Class<? extends ILookupCall<String>> getConfiguredLookupCall()
    {
        return CountryLookupCall.class;
    }
}
}
}

```

- ① Makes the address box to occupy 1 column and 3 rows.
- ② The content in the address box will use a single column layout.
- ③ Extending a Scout sequence box will place the inner fields of the **LocationBox** on a single row.
- ④ Disables the default check if the value of the first field in the sequence box is less than the value in the second field.
- ⑤ On field labels do not take any additional space and are shown in the field itself.

While string fields are used for the street field and the city field, the country field is implemented as a smart field. Scout smart fields can be viewed as a powerful drop down lists with search-as-you-type support. In the case of the country field the smart field is backed by the lookup class **CountryLookupCall** that we already used for the country smart column in the person page.

After the address box the "Contact Info" box contains the three fields mentioned below. Use the Scout content assist to add the fields and select *String Field* as the type of each field.

- Class `PhoneField`, using label "Phone"
- Class `MobileField`, using label "Mobile"
- Class `EmailField`, using label "E-Mail"

We have now completed the implementation of the form layout and added all form fields of the person form. The application is now in a state where we can verify the layout of the person form and check the handling of the different input fields. (Re)start the application and enter some values into the various fields of the person form.

To view and enter person data with the form we have yet to add the interaction with the database in the backend of the "Contacts" application. This is the topic of the next section.

Person Form Handler and Person Service

This section shows how we can integrate the person form created in the previous sections with the "Contacts" backend application to load and store person data with the database.

Most of the necessary infrastructure such as the transfer objects between the frontend and the backend application has already been created by the Scout form wizard. In the text below we will first discuss the setup created by the new form wizard and then add the missing code snippets to interact with the database.

On the frontend side, the Scout new form wizard has also created the two form handler classes `ModifyHandler` and `NewHandler`. By convention a `ModifyHandler` is used to change existing data and a `NewHandler` implements the creation of new data.

Form handler classes provide a number of callback methods that are invoked at various stages during the life cycle of the form. The implementation created by the Scout wizard includes the methods `execLoad` and `execStore` for each form handler. In these methods the form fetches data from the Scout backend application and/or sends new data to the backend server.

Adapt the default implementation of the form handlers according to [Listing 35](#).

Listing 35. The new handler and modify handler for the person form.

```
public class PersonForm extends AbstractForm {

    public class ModifyHandler extends AbstractFormHandler {

        @Override
        protected void execLoad() {
            IPersonService service = BEANS.get(IPersonService.class); ❶
            PersonFormData formData = new PersonFormData();
            exportFormData(formData); ❷
            formData = service.load(formData); ❸
            importFormData(formData); ❹

            getForm().setSubTitle(calculateSubTitle()); ❺
        }

        @Override
        protected void execStore() {
            IPersonService service = BEANS.get(IPersonService.class);
            PersonFormData formData = new PersonFormData();
            exportFormData(formData);
            service.store(formData); ❻
        }
    }

    public class NewHandler extends AbstractFormHandler {

        @Override
        protected void execStore() {
            IPersonService service = BEANS.get(IPersonService.class);
            PersonFormData formData = new PersonFormData();
            exportFormData(formData);
            formData = service.create(formData); ❼
            importFormData(formData);
        }
    }

    protected String calculateSubTitle() {
        return StringUtility.join(" ",
            getFirstNameField().getValue(),
            getLastNameField().getValue());
    }
}
```

- ❶ Obtains a reference to the person service located on the Scout backend application.
- ❷ All form field values are transferred to the form data. In this case the person primary key property will be transferred to the form data. Remember that we have set this key in the "Edit" context menu.
- ❸ The form data (including the person primary key) is sent to the **load** method. The load method

returns the person data from the backend.

- ④ The field values in the form data are loaded into the form fields of the person form.
- ⑤ The sub title on the view tab of the form is updated to reflect the name of the person.
- ⑥ Calls the `store` method of the person service providing the updated person data.
- ⑦ Calls the `create` method of the person service providing the new person data.

With the implementation provided in [Listing 35](#) the classes `ModifyHandler` and `NewHandler` orchestrate the complete roundtrip between the frontend and the backend of the "Contacts" application.

The only part that is now missing is the implementation of the form service methods `create`, `load` and `store` on the backend of the "Contacts" application. For these methods we can again rely on the default implementations created by the Scout new form wizard.

Modify the person service methods according to [Listing 36](#).

Listing 36. The `PersonService` methods to load, create and update person data.

```
public class PersonService implements IPersonService {

    @Override
    public PersonFormData create(PersonFormData formData) {
        if (!ACCESS.check(new CreatePersonPermission())) {
            throw new VetoException(TEXTS.get("InsufficientPrivileges"));
        }

        // add a unique person id if necessary
        if (StringUtil.isNullOrEmpty(formData.getPersonId())) {
            formData.setPersonId(UUID.randomUUID().toString());
        }

        SQL.insert(SQLs.PERSON_INSERT, formData); ❶

        return store(formData); ❷
    }

    @Override
    public PersonFormData load(PersonFormData formData) {
        if (!ACCESS.check(new ReadPersonPermission())) {
            throw new VetoException(TEXTS.get("InsufficientPrivileges"));
        }

        SQL.selectInto(SQLs.PERSON_SELECT, formData); ❸

        return formData;
    }

    @Override
    public PersonFormData store(PersonFormData formData) {
        if (!ACCESS.check(new UpdatePersonPermission())) {
            throw new VetoException(TEXTS.get("InsufficientPrivileges"));
        }

        SQL.update(SQLs.PERSON_UPDATE, formData); ❹

        return formData;
    }
}
```

- ❶ The SQL insert statement adds a new person entry in the database. Only the primary key is used to create this entry.
- ❷ To save all other person attributes provided in the form data, the `store` method is reused.
- ❸ The SQL select into transfers the person data from the database into the form data.
- ❹ The SQL update statement transfers all person attributes provided in the form data to the person table.

What have we achieved?

In the fourth step of the "Contacts" tutorial we have added the person form to add, view and change persons. Using the person form as an example we have learned how to implement complex form layouts using the Scout layouting mechanism, Scout container fields and individual form field properties.

We have also seen how we can use context menus to integrate the forms in the user interface of the application and have implemented the interaction of the frontend with the backend application including the persistence of person data in the database.

The "Contacts" application is in a clean state again and you can (re)start the backend and the frontend of the application and verify the result in your browser. We can now verify the creation of new person entries and the modification of existing person data in the current state of the "Contacts" application. The created person form is shown in [Figure 24](#). In case you copied some code snippets from the tutorial, you may see the text "undefined text {...}" in some labels in the person field. You may want to define these texts using the Scout content assist for defining new texts as was already presented earlier in this tutorial.

The screenshot shows a web browser window titled 'Contacts Application' with the URL 'localhost:8082/?dl=outline-11770&i=contacts'. The application has a dark blue sidebar with a search icon and a menu containing 'Contacts', 'Persons', and 'Organizations'. The main content area is titled 'Person Alice' and includes 'Quick access', 'Options', and 'User' links. There are 'OK' and 'Cancel' buttons at the top. The form contains a large illustration of Alice and the White Rabbit from 'Alice's Adventures in Wonderland' sitting at a table. To the right of the illustration are form fields for 'Gender' (radio buttons for Male and Female, with Female selected), 'First name' (filled with 'Alice'), 'Last name' (empty), and 'Date of birth' (filled with '11/26/1865'). Below the illustration are three tabs: 'Contact info' (selected), 'Work', and 'Notes'. Under 'Contact info', there are fields for 'Street', 'Location' (filled with 'Daresbury, Cheshi United Kingdor'), 'Phone', 'Mobile', and 'Email'.

Figure 24. The "Contacts" application with the person form at the end of tutorial step 4.

Form Field Validation and Template Fields

This tutorial step introduces two additional concepts that are used in most Scout applications. Form field validation and template fields. Form field validation helps to keep data quality high and template fields are used to increase the code quality of a Scout application.

In addition to just retrieving and storing new data, a business application should also help the user to maintain the quality of the entered data. To validate user input, the Scout framework offers form field validation. Simple input validation is possible on the level of individual fields as shown in [Simple Form Field Validation](#). Scout also offers mechanisms to validate field values on the level of container fields or on the level of a form as shown in [Complex Form Field Validation](#). In the text below we add a number of form field validations that implement this approach for the person form.

In [Creating Template Fields](#) we refactor the picture field code into a template field that can later be re-used for the organization form. To edit the image URL we add a simple edit form to the refactored picture field in [Adding a simple URL Input Form to the Picture Field](#).

In [More Template Fields](#) we outline the creation of additional template fields and provide a summary of this tutorial step in [What have we achieved?](#).

Simple Form Field Validation

This section explains the form field validation on the level of a single field. As an example we will use the email address field defined in the "Contact Info" tab. The validation implemented in [Listing 37](#) checks the length and the format of the entered email address.

```
@Order(40)
@ClassId("5f9d9363-8e57-4151-b281-7d401e64702c")
public class EmailField extends AbstractStringField {

    private static final String EMAIL_PATTERN = ①
        "^[_A-Za-z0-9-\\+](\\.[_A-Za-z0-9-]+)*@" +
        "[A-Za-z0-9-](\\.[A-Za-z0-9-]+)*(\\.[A-Za-z]{2,})$";

    @Override
    protected String getConfiguredLabel() {
        return TEXTS.get("Email");
    }

    @Override ②
    protected int getConfiguredMaxLength() {
        return 64;
    }

    @Override ③
    protected String execValidateValue(String rawValue) {
        if (rawValue != null && !Pattern.matches(EMAIL_PATTERN, rawValue)) {
            throw new VetoException(TEXTS.get("BadEmailAddress")); ④
        }

        return rawValue; ⑤
    }
}
```

- ① Email verification is performed against a simple regular expression.
- ② This prevents the field from accepting more than 64 characters. The return value should match the size of the corresponding table column.
- ③ Method `execValidateValue` is called during validation of the new field value.
- ④ If the value violates any business rules, a `VetoException` should be thrown.
- ⑤ If the new value passes all business rules the method returns the value.

In the next section we use the address box to demonstrate the joint validation of several fields.

Complex Form Field Validation

Often the values of several fields have to be considered jointly to evaluate if the entered data is actually valid. As an example we will add a more complex form field validation on the level of the `AddressBox` group box widget that takes into account the data entered into the street, city, and country fields.

The implemented validation for the address box example should enforce the following set of business rules.

- Only valid countries should be allowed
- If a city is provided a country must also be provided
- If street information is provided, both a city and a country must be provided
- The address may be empty

The simplest rule is about entering only valid countries. This rule is already implemented as the country smart field only allows the user to select a single entry of the list of valid countries. A possible implementation to enforce the other rules is provided in [Listing 38](#).

Listing 38. The validation of the fields in the address box

```
@Order(10)
@ClassId("736450dd-ba89-43cd-ba52-bcd31196b462")
public class AddressBox extends AbstractGroupBox {

    @Order(10)
    @ClassId("a9137ad1-af9d-4fef-a69d-3e3d9ce48f21")
    public class StreetField extends AbstractStringField {

        @Override ①
        protected void execChangedValue() {
            validateAddressFields(); ②
        }
    }

    @Order(20)
    @ClassId("a278333c-057e-4c1d-a442-0c1dd62fdca7")
    public class LocationBox extends AbstractSequenceBox {

        @Order(10)
        @ClassId("3ea6ac2a-976e-4c7f-b04b-ec0d7d1ae5ec")
        public class CityField extends AbstractStringField {

            @Override
            protected void execChangedValue() {
                validateAddressFields(); ②
            }
        }

        @Order(20)
        @ClassId("d4dfce4f-019b-4a61-ba78-347ef67cf80f")
        public class CountryField extends AbstractSmartField<String> {

            @Override
            protected void execChangedValue() {
                validateAddressFields(); ②
            }
        }
    }

    protected void validateAddressFields() {
        boolean hasStreet = StringUtility.hasText(getStreetField().getValue());
        boolean hasCity = StringUtility.hasText(getCityField().getValue());

        getCityField().setMandatory(hasStreet); ③
        getCountryField().setMandatory(hasStreet || hasCity);
    }
}
```

① This method is called after the value of this field has been changed.

- ② After changing the street, the city or the country recompute which address fields are mandatory.
- ③ The city becomes mandatory if the street field is not empty. The country is mandatory if the street or the city is not empty.

Whenever the content of the street field, the city field, or the country field is changed the mechanism implemented above triggers a re-evaluation of the mandatory status of the city field and the country field. As the Scout default form validation ensures that every mandatory field receives some content the application prevents the user from entering address data that does not satisfy the business rules mentioned above.

The verification of user input can also be triggered before the form is closed. This behavior can be implemented by overriding method `execValidate` on the form level. As an example we use this mechanism to make sure that a user can only enter persons that have at least some name information.

Now add this validation to the person form using the implementation provided in [Listing 39](#).

Listing 39. The validation of the first and last names on the form level

```
public class PersonForm extends AbstractForm {  
  
    @Override ①  
    protected boolean execValidate() {  
        boolean noFirstName = StringUtility.isNullOrEmpty(getFirstNameField().getValue());  
        boolean noLastName = StringUtility.isNullOrEmpty(getLastNameField().getValue());  
  
        if (noFirstName && noLastName) {  
            getFirstNameField().requestFocus(); ②  
  
            throw new VetoException(TEXTS.get("MissingName")); ③  
        }  
  
        return true; ④  
    }  
}
```

- ① This method is called during the form validation and before the form is stored/closed.
- ② Place the focus on the first name field.
- ③ In case both the first name and the last name fields are empty throw a `VetoException`, this will fail the validation.
- ④ The return value indicates if the validation has passed successfully or not.

As we have now implemented a number of form field validations we are now ready to test the result in the running application. Re-start the "Contacts" application and try to trigger the different validation rules. [Figure 25](#) shows the response of the user interface when trying to save invalid person data.

The screenshot shows a web browser window with the URL `localhost:8082/?dl=outline-11770&i=contacts`. The browser displays a 'Person' form. A modal dialog box is open in the center, listing validation errors:

- The following fields are required:**
 - Contact info: Location: City
 - Contact info: Location: Country
- The following fields are invalid:**
 - Contact info: Email: Bad email address

The form itself has several fields: Gender (radio buttons for Male and Female), First name, Last name, Date of birth, and Contact info (with sub-fields for Location: City and Country). The Email field has a red error message 'Bad email address' next to it. The form also has 'OK' and 'Cancel' buttons at the top.

Figure 25. The form field validation implemented for the person form.

Creating Template Fields

In this section we show how to refactor a group of fields into a Scout template field that is ready for reuse. As an example we refactor the picture field into a template field. Later in tutorial step [Adding the Company Form](#) we can then reuse this field in the company form to show the company's logo.

The generic approach to refactor a form field into a template field is listed below.

1. Create an empty field data class in the shared module
2. Create the template field class in the client module
3. Copy the existing field code to the template field
4. Let the original field extend the new template field and fix imports

For refactoring the picture field we can exactly follow these steps. To create the empty field data class perform the following steps.

1. Expand the shared module of the "Contacts" application
2. Navigate into folder `src/generated/java`
3. Add a new package `org.eclipse.scout.contacts.shared.common`
4. Create class `AbstractUrlImageFieldData` in this package as shown in [Listing 40](#)

Listing 40. The empty form data class for the picture template field.

```
package org.eclipse.scout.contacts.shared.common;

public abstract class AbstractUrlImageFieldData {
}
```

We are now ready to implement the template field class according to the following steps.

1. Navigate to the client module of the "Contacts" application
2. Select package `org.eclipse.scout.contacts.client.common` in folder `src/main/java`
3. Create class `AbstractUrlImageField` using `Ctrl + N`
4. Update the implementation according to [Listing 41](#)

Listing 41. The refactored picture field.

```
@ClassId("73a4276f-77b2-4ad2-b414-7f806284bdb3")
@FormData(value = AbstractUrlImageFieldData.class, ❶
    sdkCommand = SdkCommand.CREATE,
    defaultSubtypeSdkCommand = DefaultSubtypeSdkCommand.CREATE)
public abstract class AbstractUrlImageField extends AbstractImageField {

    private String url; ❷

    @FormData ❷
    public String getUrl() {
        return url;
    }

    @FormData ❷
    public void setUrl(String url) {
        this.url = url;
        updateImage();
    }

    @Override
    protected boolean getConfiguredLabelVisible() {
        return false;
    }

    @Override
    protected int getConfiguredGridH() {
        return 5;
    }

    @Override
    protected boolean getConfiguredAutoFit() {
        return true;
    }

    @Override
    protected String getConfiguredImageId() {
        return Icons.User;
    }

    protected void updateImage() {
        setImageUrl(this.url);
    }
}
```

❶ The link to the corresponding field data class.

❷ Field `PictureUrlField` is refactored into the property `url` value. To transfer the content of this property to the field data object we need to add annotation `@FormData` to its getter and setter

methods.

The next step is to replace the original code of the picture field with the newly created template field. Delete the field `PictureUrlField` and remove all the code from the field `PictureField` and let `PictureField` extend the newly created template field as shown in [Listing 42](#).

Listing 42. The refactored picture field.

```
@Order(10)
@ClassId("e7efc084-fe7a-462f-ba23-914e58f7b82d")
public class MainBox extends AbstractGroupBox {

    @Override
    protected void injectMenusInternal(OrderedCollection<IMenu> menus) {
        BEANS.get(ContactsHelper.class).injectReadOnlyMenu(menus);
    }

    @Order(10)
    @ClassId("b20aad47-e070-4f3c-bafc-ddbaa3ae2a4c")
    public class GeneralBox extends AbstractGroupBox {

        @Order(10)
        @ClassId("d80625e3-b548-47e4-9cae-42d70aaa568f")
        public class PictureField extends AbstractUrlImageField { ①
        }

        // additional form field

    }
}
```

① The implementation of the picture field is now provided by the template field `AbstractUrlImageField`.

As the last step we need to slightly modify the SQL statement that loads and stores the picture URL information. The reason for the change is the replacement of the picture url field by an url property defined as a member of the picture field. For this change perform the steps listed below.

1. Open class `SQLs` in the Java editor.
2. In string `PERSON_SELECT` change the token `'pictureUrl'` with `'picture.url'`
3. In string `PERSON_UPDATE` change the token `'pictureUrl'` with `'picture.url'`

Based on the picture field example we have now walked through the complete process to turn normal fields into template fields. This process remains the same for refactoring container fields into template fields.

Adding a simple URL Input Form to the Picture Field

Using the refactored picture template field we want the user to be able to enter and update the URL

of the shown picture. For this we add a simple form with a single field and a context menu to the picture template field using the Scout new form wizard as shown in [Figure 26](#).

- Verify that you use the correct source folder and package name.
- In the *Name* field enter "PictureUrl".
- In section *Additional Components* deselect all checkboxes.
- Click [**Finish**] to let the wizard implement the form.



Figure 26. Creating the picture URL form with the new form wizard.

Now adapt the content of the URL form according to [Listing 43](#). As you can see, there is no roundtrip to a backend server and the form only contains a single editable field.

Listing 43. The form to edit the picture URL

```
@ClassId("3b30ebf1-e8fe-4dd3-8124-5f5038b1d47c")
public class PictureUrlForm extends AbstractForm {

    @Override
    protected String getConfiguredTitle() {
        return TEXTS.get("PictureURL");
    }

    public void startModify() {
        startInternal(new ModifyHandler());
    }
}
```

```

}

public UrlField getUrlField() {
    return getFieldByClass(UrlField.class);
}

public InfoField getInfoField() {
    return getFieldByClass(InfoField.class);
}

@Order(10)
@ClassId("6c5e0da2-cf04-402f-9784-43e3a138796b")
public class MainBox extends AbstractGroupBox {

    @Order(10)
    @ClassId("fdcc7087-a693-45e8-a889-3725b0995558")
    public class UrlBox extends AbstractGroupBox {

        @Order(10)
        @ClassId("32b71aa6-1109-4b39-996f-f35a677faa06")
        public class UrlField extends AbstractStringField {

            @Override
            protected boolean getConfiguredLabelVisible() { ①
                return false;
            }

            @Override
            protected boolean getConfiguredStatusVisible() {
                return false;
            }

            @Override
            protected int getConfiguredGridW() {
                return 2;
            }
        }
    }

    @Order(20)
    @ClassId("999c32e9-ca87-4b5c-a907-29d7a7400abf")
    public class InfoField extends AbstractHtmlField {

        @Override
        protected boolean getConfiguredLabelVisible() {
            return false;
        }

        @Override
        protected boolean getConfiguredStatusVisible() {
            return false;
        }
    }
}

```

```

@Override
protected int getConfiguredGridW() {
    return 2;
}

@Override
protected boolean getConfiguredGridUseUiHeight() {
    return true;
}

@Override
protected void execInitField() {
    setValue(HTML.fragment(HTML.icon(Icons.Info), HTML.bold(" " + TEXTS.get(
    "PleaseNote") + ": "), TEXTS.get("SecurityUrlRestrictedMsg")).toHtml());
}
}

@Order(20)
@ClassId("4e15ce0e-502c-4290-aeca-e83359f3bc5b")
public class OkButton extends AbstractOkButton {
}

@Order(30)
@ClassId("f278815a-f4cf-4e86-a057-66cb7ce43fc3")
public class CancelButton extends AbstractCancelButton {
}

public class ModifyHandler extends AbstractFormHandler { ②
}
}

```

① No label is needed as the name of the field is already provided by the title of the form.

② As no round trip to the backend is required the modify handler can remain empty.

We can now add an "Edit URL" menu to the picture template field. The implementation of the edit context menu is provided in [Listing 44](#).

Listing 44. The "Edit URL" menu for the refactored picture field

```
public abstract class AbstractUrlImageField extends AbstractImageField {

    @Order(10)
    @ClassId("99c1c12a-84d4-4c1a-a009-dfd2b7b55ded")
    public class EditURLMenu extends AbstractMenu {

        @Override
        protected String getConfiguredText() {
            return TEXTS.get("EditURL");
        }

        @Override
        protected Set<? extends IMenuType> getConfiguredMenuTypes() {
            return CollectionUtility.<IMenuType> hashSet(
                ImageFieldMenuType.ImageUrl,
                ImageFieldMenuType.ImageId,
                ImageFieldMenuType.Null);
        }

        @Override
        protected void execAction() {
            PictureUrlForm form = new PictureUrlForm();
            String oldUrl = getUrl();

            if (StringUtility.hasText(oldUrl)) { ❶
                form.getUrlField().setValue(oldUrl);
            }

            form.startModify();
            form.waitFor(); ❷

            if (form.isFormStored()) { ❸
                setUrl(form.getUrlField().getValue());
                getForm().touch();
            }
        }
    }
}
```

- ❶ If we already have an URL for the picture prefill the url field in the form with its value.
- ❷ Method `waitFor` makes the application wait until the user has closed the form.
- ❸ Only store the new URL if the user has saved a new value. Storing the value will refresh the picture in the user interface.

Based on the example with the picture field we have now walked through the complete process to turn normal fields into template fields. This process remains the same for refactoring container fields into template fields.

More Template Fields

To reduce the amount of copy & paste for the implementation of the company form in the next tutorial step, we recommend that you refactor the following fields into templates.

- Email field
- Address group box field
- Notes group box field

You can follow the steps described in the previous section for the picture field. To be able to copy & paste the code in the following tutorial step you may use the following class names.

- `AbstractEmailField` for the email template field
- `AbstractAddressBox` for the address group template field
- `AbstractNotesBox` for the notes tab template field

Note that both the `AbstractAddressBox` and the `AbstractNotesBox` need their own form data object, whereas the `AbstractEmailField` does not.

Replacing the concrete fields with the template fields in the person form will result in a number of compile errors in the field getter methods of the person form. In the case of the "Contacts" application these getter methods are not needed and can simply be deleted.

Moving from concrete fields to template fields also implies some minor changes as we have seen with the picture template field. Therefore make sure to modify the SQL statements in class `SQLs` accordingly.

- Replace token `':street'` by `':addressBox.street'`
- Replace token `':city'` by `':addressBox.city'`
- Replace token `':country'` by `':addressBox.country'`
- Replace token `':notes'` by `':notesBox.notes'`

What have we achieved?

In this step of the "Contacts" tutorial we have covered two important concepts for implementing business applications.

- Validation of user input on the level of fields, components and the complete form
- Creation and usage of template fields to minimize copy & paste where possible

The "Contacts" application is in a clean state again and you can (re)start the backend and the frontend of the application and verify the result in your browser. Using the created picture template field we can now update the image in the picture form as shown in [Figure 27](#).



Figure 27. The person form with the refactored picture template field including a menu (red square) and a URL edit form.

In the next tutorial step we are going to implement the company form to enter and edit company information. For the creation of this form we can reuse the template fields that we have created.

Adding the Company Form

This section describes the implementation of the organization form. For the implementation of the organization form we can apply many of the concepts we have learned in the previous sections. As a result, the descriptions of this section can be kept on a much higher level.

The sketch shows a form layout for an organization. At the top left is a large rectangular area with a diagonal 'X' inside, representing a picture field. To its right are two text input fields: 'Name' and 'Homepage'. Below the picture field are two tabs: 'Contact Info' and 'Notes'. The 'Contact Info' tab is active, showing a sub-form with four fields: 'Street' (text input), 'Phone' (text input), 'Location' (a dropdown menu with 'City' selected), and 'Country' (a dropdown menu with a downward arrow). To the right of the 'Location' and 'Country' fields is an 'Email' text input field.

Figure 28. The sketch of the organization form layout.

Considering the layout sketch for the organization form shown in [Figure 28](#) we can already see how we can reuse the following fields / templates.

- The picture field
- The address box with street, city and country including its validation
- The email field with its validation
- The complete "Notes" tab

For the remaining fields "Name", "Homepage" and "Phone" we will use simple string fields with matching label texts.

We can now implement the company form according to the following steps.

1. Expand folder `src/main/java` in the client module in the package explorer
2. Select package `org.eclipse.scout.contacts.client.organization` and hit `Ctrl + N`
3. Enter "form" into the search field of the wizard selection and double click on proposal *Scout Form*
4. Use "OrganizationForm" as class name and make sure to select all check boxes in the lower part of the wizard

5. Click **[Finish]** to start the Scout form wizard.

After creating the initial form class using Scout's new form wizard the form layout can be implemented according to [Listing 45](#).

Listing 45. The layout implementation of the organization form

```
public class OrganizationForm extends AbstractForm {

    private String organizationId;

    @FormData
    public String getOrganizationId() {
        return organizationId;
    }

    @FormData
    public void setOrganizationId(String organizationId) {
        this.organizationId = organizationId;
    }

    @Override
    public Object computeExclusiveKey() {
        return getOrganizationId();
    }

    @Override
    protected String getConfiguredTitle() {
        return TEXTS.get("Organization");
    }

    @Override
    protected int getConfiguredDisplayHint() {
        return IForm.DISPLAY_HINT_VIEW;
    }

    @Override
    protected void execInitForm() {
        BEANS.get(ContactsHelper.class).handleReadOnly(getOkButton());
    }

    @Order(10)
    @ClassId("e7efc084-fe7a-462f-ba23-914e58f7b82d")
    public class MainBox extends AbstractGroupBox {

        @Override
        protected void injectMenusInternal(OrderedCollection<IMenu> menus) {
            BEANS.get(ContactsHelper.class).injectReadOnlyMenu(menus);
        }
    }
}
```

```

@Order(10)
@ClassId("b20aad47-e070-4f3c-bafc-ddbaa3ae2a4c")
public class GeneralBox extends AbstractGroupBox {

    @Order(10)
    @ClassId("d80625e3-b548-47e4-9cae-42d70aaa568f")
    public class PictureField extends AbstractUrlImageField { ①

        @Override
        protected int getConfiguredGridH() { ②
            return 4;
        }

        @Override
        protected double getConfiguredGridWeightY() { ③
            return 0;
        }
    }

    @Order(20)
    @ClassId("4c1a0dea-6c04-4cad-b26b-8d5cc1b786a9")
    public class NameField extends AbstractStringField {

        @Override
        protected String getConfiguredLabel() {
            return TEXTS.get("Name");
        }

        @Override
        protected boolean getConfiguredMandatory() { ④
            return true;
        }
    }

    @Order(30)
    @ClassId("68008603-257f-45dc-b8ea-d1e066682205")
    public class HomepageField extends AbstractStringField {

        @Override
        protected String getConfiguredLabel() {
            return TEXTS.get("Homepage");
        }
    }
}

@Order(20)
@ClassId("4e48c196-22e4-4e22-965a-5e305af5e6a9")
public class DetailsBox extends AbstractTabBox {

    @Order(10)
    @ClassId("c6c9e644-2ab3-436e-9d8a-bdcc5482eb5b")

```

```

public class ContactInfoBox extends AbstractGroupBox {

    @Override
    protected String getConfiguredLabel() {
        return TEXTS.get("ContactInfo");
    }

    @Order(10)
    @ClassId("2a10bd00-de56-4a97-a5b2-6a8a0aae925f")
    public class AddressBox extends AbstractAddressBox { ⑤
    }

    @Order(20)
    @ClassId("504a4845-d307-4238-a2e9-9e785c1477ac")
    public class PhoneField extends AbstractStringField {

        @Override
        protected String getConfiguredLabel() {
            return TEXTS.get("Phone");
        }
    }

    @Order(30)
    @ClassId("0b4d059d-ec81-4e93-9a99-2512d734ebac")
    public class EmailField extends AbstractEmailField { ⑥
    }
}

@Order(20)
@ClassId("85f4dfb0-f375-4e90-be92-b59e9bc2ebcf")
public class NotesBox extends AbstractNotesBox { ⑦
}

@Order(30)
@ClassId("97c3ceed-d005-47da-b44d-def4b07f92ab")
public class OkButton extends AbstractOkButton {
}

@Order(40)
@ClassId("d63bfcd6-7464-4e4f-a07e-eb1173a77f8c")
public class CancelButton extends AbstractCancelButton {
}
}
}

```

- ① We reuse the picture template field to display the company logo.
- ② We reduce the number of rows for the company logo compared to the person picture.
- ③ We do not allow the general box to grow or shrink vertically

- ④ We configure the company name field to be mandatory for an organization.
- ⑤ As-is reuse of the address template box.
- ⑥ As-is reuse of the email template field.
- ⑦ As-is reuse of the notes tab box.

To be able to open the organization form we need to link the form to the user interface. Following the pattern for the person form we define the context menus "Edit" and "New" for the organization table and a menu "Create new organization" under the "Quick access" top level menu.

The implementation of the organization form is completed by providing the logic to interact with the database in the organization service according to [Listing 46](#). The technical setup exactly follows the implementation of the person service.

Listing 46. The `OrganizationService` methods to load, create and update organization data.

```
public class OrganizationService implements IOrganizationService {

    @Override
    public OrganizationFormData create(OrganizationFormData formData) {
        if (!ACCESS.check(new CreateOrganizationPermission())) {
            throw new VetoException(TEXTS.get("InsufficientPrivileges"));
        }

        if (StringUtility.isNullOrEmpty(formData.getOrganizationId())) {
            formData.setOrganizationId(UUID.randomUUID().toString());
        }

        SQL.insert(SQLs.ORGANIZATION_INSERT, formData);

        return store(formData);
    }

    @Override
    public OrganizationFormData load(OrganizationFormData formData) {
        if (!ACCESS.check(new ReadOrganizationPermission())) {
            throw new VetoException(TEXTS.get("InsufficientPrivileges"));
        }

        SQL.selectInto(SQLs.ORGANIZATION_SELECT, formData);

        return formData;
    }

    @Override
    public OrganizationFormData store(OrganizationFormData formData) {
        if (!ACCESS.check(new UpdateOrganizationPermission())) {
            throw new VetoException(TEXTS.get("InsufficientPrivileges"));
        }

        SQL.update(SQLs.ORGANIZATION_UPDATE, formData);

        return formData;
    }
}
```

Method `prepareCreate` is not needed for the creation of a new organization and we can remove it from `OrganizationService` and `IOrganizationService`. Therefore, the implementation of the method `execLoad` in the new handler of the organization form can also be removed.

With these implementations of the organization form and organization service the "Contacts" application can now also be used to maintain a list of organizations.

What have we achieved?

In the sixth step of the "Contacts" tutorial we have added the Scout form to edit and create organizations. The focus of this part of the tutorial was on re-using previous work and applying the concepts that have been introduced in previous tutorial steps.

The "Contacts" application is in a clean state again and you can (re)start the backend and the frontend of the application and verify the result in your browser. As shown in [Figure 29](#) company data can now be viewed and entered in the user interface.



Figure 29. The "Contacts" application with the newly created organization form.

Linking Organizations and Persons

In this step we modify the user interface to represent the 1:n relationship between organizations and persons. For the implementation of this 1:n relation we follow the Scout standard pattern.

In the "Contacts" application any person can be assigned to a single organization. This fact is represented in the database schema created using the statement `SQLs.PERSON_CREATE_TABLE`.

We will therefore need to be able to assign a person to an existing organization by selecting an existing organization in the field. For this we modify the organization field on the person to a smart field. To display the assigned organizations we will also modify the person page accordingly.

In addition we would like to be able to easily access all persons assigned to a specific organization. Using the existing organization page we will add a child page that will then show all associated persons. This will result in a drill-down functionality for the organization page.

The implementation of the features described above can be achieved by the the following steps.

- Creating an Organization Lookup Call ([Creating an Organization Lookup Call](#))
- Using the Lookup Call in the Person Form and the Person Table ([Using the Lookup Call in the Person Form and the Person Table](#))
- Link the Person Page to Organizations ([Link the Person Page to Organizations](#))

This last tutorial step ends with a short review in [What have we achieved?](#)

Creating an Organization Lookup Call

Before we can change the organization field on the person form from a string field to a smart field we need a organization lookup call that provides the necessary data to the smartfield. We have been using this approach for the country field already. The difference to the lookup call for countries lies in the fact that we no longer have a static list of entries but need to fetch possible the organizations dynamically. We will therefore need to access the database to provide the data to the lookup call.

As this is a common requirement the Scout framework comes with the base class `AbstractSqlLookupService` and a default mechanism to route lookup calls from the frontend sever to database calls on the backend server. The necessary infrastructure can be created using the Scout lookup wizard according to the steps described below.

1. Expand folder `src/main/java` in the client module in the package explorer
2. Select package `org.eclipse.scout.contacts.shared.organization` and hit `Ctrl + N`
3. Enter "lookup" into the search field of the wizard selection and double click on proposal *Scout LookupCall*
4. Use "OrganizationLookupCall" as class name
5. Enter "String" as the key class and use service super class "AbstractSqlLookupService" in the wizard

6. Verify that the fields in the wizard match the values provided in [Figure 30](#)
7. Click **[Finish]** to start the Scout lookup call wizard.

Figure 30. Using the Scout lookup call wizard for creating class `OrganizationLookupCall`.

The Scout wizard will then create the lookup class `OrganizationLookupCall` and the corresponding lookup service with the interface `IOrganizationLookupService` and its initial implementation `OrganizationLookupService`. We will only need to provide some implementation for the lookup service. The service interface and the lookup call class can be used as provided by the Scout wizard. [Listing 47](#) shows the code generated for the lookup call

Listing 47. The `OrganizationLookupCall` implemented by the Scout wizard.

```
@ClassId("22789824-ad89-4208-bc11-5c08b56ce998")
public class OrganizationLookupCall extends LookupCall<String> {

    private static final long serialVersionUID = 1L;

    @Override
    protected Class<? extends ILookupService<String>> getConfiguredService() {
        return IOrganizationLookupService.class;
    }
}
```

We are now ready to implement method `getConfiguredSqlSelect` of the organization lookup service.

Open class `OrganizationLookupService` in the Java editor and change the implementation according to [Listing 48](#).

Listing 48. The `OrganizationService` methods to load, create and update organization data.

```
public class OrganizationLookupService
    extends AbstractSqlLookupService<String>
    implements IOrganizationLookupService {

    @Override
    protected String getConfiguredSqlSelect() {
        return SQLs.ORGANIZATION_LOOKUP; ❶
    }
}
```

❶ We only need to return a single SQL statement for lookup services that extend `AbstractSqlLookupService`

The SQL statement that backs the lookup service is provided in [Listing 49](#). Lookup services can provide data for three different use cases. The most straightforward case is the mapping of a key to a specific lookup row. Next is the case where the lookup service returns a number of lookup rows that match a provided substring and finally the case where the lookup service simply returns all available rows.

Listing 49. The SQL statement to provide the data for the organization lookup service.

```
String ORGANIZATION_LOOKUP = ""
    + "SELECT    organization_id, "
    + "          name "
    + "FROM      ORGANIZATION "
    + "WHERE     1 = 1 "
    + "<key>     AND organization_id = :key</key> " ❶
    + "<text>    AND UPPER(name) LIKE UPPER(:text||'%') </text> " ❷
    + "<all></all>"; ❸
```

- ❶ The where clause to be used for a search for a specific key
- ❷ The where clause to be used when some search text is provided
- ❸ The where clause that defines the full set of lookup rows

Using the Lookup Call in the Person Form and the Person Table

Now we can use the organization lookup call to transform the organization field in the "Work" tab of the person form into a smart field. To do this we open class `PersonForm` in the Java editor and navigate to its inner class `WorkBox`. Then, update the implementation of the `OrganizationField` according to [Listing 50](#)

Listing 50. The organization smart field in the "Work" tab backed by the `OrganizationLookupCall`.

```
@Order(20)
@ClassId("8e18a673-aca5-44a2-898f-60a744e4467a")
public class WorkBox extends AbstractGroupBox {

    @Order(20)
    @ClassId("cd4a7afd-e0ac-4c79-bf2e-819aa491db27")
    public class OrganizationField extends AbstractSmartField<String> { ❶

        @Override
        protected String getConfiguredLabel() {
            return TEXTS.get("Organization");
        }

        @Override ❷
        protected Class<? extends ILookupCall<String>> getConfiguredLookupCall() {
            return OrganizationLookupCall.class;
        }
    }
}
```

❶ The `OrganizationField` now extends a Scout smart field

❷ The smart field is backed by the newly created organization lookup call

This change has the effect, that now we can assign an organization in the person form by typing a substring of the organizations name into the organization field. The conversion of the field into a smart field has the additional benefit that only valid organizations can be selected that respect the referential integrity defined by the database.

As a next step we also modify the organization column of the person page. For this open class `PersonTablePage` in the Java editor and navigate to its inner class `Table`. Then, change the implementation of `OrganizationColumn` according to [Listing 51](#).

Listing 51. The organization smart column in the person page.

```
@ClassId("3fa1374b-9635-441b-b2f8-feb24b50740a")
public class Table extends AbstractTable {

    @Order(9)
    @ClassId("2e53e50e-5bd5-421e-8bca-fc50f27d790b")
    public class OrganizationColumn extends AbstractSmartColumn<String> {

        @Override
        protected String getConfiguredHeaderText() {
            return TEXTS.get("Organization");
        }

        @Override
        protected Class<? extends ILookupCall<String>> getConfiguredLookupCall() {
            return OrganizationLookupCall.class;
        }
    }
}
```

Using the created organization lookup calls we have now completed the modifications on the person form and also used the lookup call to display the a person's organization in the person page. The next section will focus on the necessary modifications and new components to re-use the person page as a sub page of the organization page.

Link the Person Page to Organizations

In this section we will implement a drill-down functionality on the organization page. The goal is to let the user of the application expand a row in the organization page to provide access to the persons of the organization.

Scout node pages are useful when we want to display different entities that are related to a specific entry in a parent page. In the "Contacts" demo application this mechanism is used to link both persons and events to an organization as shown in [Figure 31](#). Note that this is a screenshot of the "Contacts" demo application, not the tutorial application that we are building here.



Figure 31. A drill-down on an organization in the "Contacts" demo application provides access to related persons and events.

In the "Contacts" demo application this hierarchical page structure is implemented as follows.

- Organization page implemented in class `OrganizationTablePage`
 - A node page implemented in class `OrganizationNodePage`
 - Person page implemented in class `OrganizationTablePage`
 - Event page implemented in class `EventTablePage`

For the "Contacts" tutorial application we will create the exact same structure but only add the person page as child page to the organization node page.

To implement this sequence of linked pages we will follow the dependencies of the linked classes. We start with adapting method `getPersonTableData` in the person service by adding an organization id parameter. Using this parameter we can then restrict the person search to the subset that is linked to the specified organization. For this change we first update the person service interface as shown in [Listing 52](#).

Listing 52. The updated method `getPersonTableData` for the person service interface.

```
@ApplicationScoped
@TunnelToServer
public interface IPersonService {

    PersonTablePageData getPersonTableData(SearchFilter filter, String organizationId);
    ①

    PersonFormData create(PersonFormData formData);

    PersonFormData load(PersonFormData formData);

    PersonFormData store(PersonFormData formData);
}
```

① Add parameter `organizationId`

We now adapt the method implementation in the person service according to Listing 53.

Listing 53. Method `getPersonTableData` to access the database and map the data into a page data object.

```
public class PersonService implements IPersonService {

    @Override
    public PersonTablePageData getPersonTableData(SearchFilter filter, String
organizationId) {
        PersonTablePageData pageData = new PersonTablePageData();
        StringBuilder sql = new StringBuilder(SQLs.PERSON_PAGE_SELECT);

        // if an organization is defined, restrict result set to persons that are
        linked to it
        if (StringUtility.hasText(organizationId)) {
            sql.append(String.format("WHERE LOWER(organization_id) LIKE LOWER('%s') ",
                organizationId));
        }

        sql.append(SQLs.PERSON_PAGE_DATA_SELECT_INTRO);
        SQL.selectInto(sql.toString(), new NVPair("page", pageData));

        return pageData;
    }
}
```

Having modified the person service we add a organization id property to the person page. We can then populate this property when the person page is attached to the organization node page. Finally we can use in method `execLoadData` according to Listing 54.

Listing 54. Add the possibility to restrict the list of persons to those assigned to a specific organization.

```
@PageData(PersonTablePageData.class)
@ClassId("23c10251-66b1-4bd6-a9d7-93c7d1aededede")
public class PersonTablePage extends AbstractPageWithTable<Table> {

    private String organizationId; ❶

    public String getOrganizationId() {
        return organizationId;
    }

    public void setOrganizationId(String organizationId) {
        this.organizationId = organizationId;
    }

    @Override
    protected void execLoadData(SearchFilter filter) {
        importPageData(BEANS.get(IPersonService.class)
            .getPersonTableData(filter, getOrganizationId())); ❷
    }

    @ClassId("3fa1374b-9635-441b-b2f8-feb24b50740a")
    public class Table extends AbstractTable {

        @Order(20)
        @ClassId("8ac358f2-de17-4b2b-93f3-73e21a7415d8")
        public class NewMenu extends AbstractMenu {

            @Override
            protected void execAction() {
                PersonForm form = new PersonForm();
                form.getOrganizationField().setValue(getOrganizationId()); ❸
                form.addFormListener(new PersonFormListener());
                // start the form using its new handler
                form.startNew();
            }
        }
    }
}
```

- ❶ This property lets the person page remember an organization key
- ❷ Provides the organization key to the person search on the backend server
- ❸ If the user creates a new person below an organization pre-fill the corresponding field

In the cases where the modified person page is shown as a child page of the organization page we can now improve the usability of the page's new menu. When creating a person under an existing organization we create the new person with a pre-filled organization id. See the modified `execAction` method in `NewMenu` of Listing 54.

The next step in the setup of the page hierarchy is the creation of the organization node page. Node pages allow to define a list of child pages that typically represent different entities. As mentioned before we will only have the person page as a child page in the "Contacts" tutorial application.

To create the organization node page follow the steps listed below.

1. Expand folder `src/main/java` in the client module in the package explorer
2. Select package `org.eclipse.scout.contacts.client.organization` and hit `Ctrl + N`
3. Enter "scout page" into the search field of the wizard selection and double click on proposal *Scout Page*
4. Add "Organization" to the class name field
5. Switch the super class field to "AbstractPageWithNodes"
6. Verify that the fields in the wizard match the values provided in [Figure 32](#)
7. Click [**Finish**] to start the Scout page wizard



Figure 32. Creating the organization node page.

After the wizard has created the initial implementation of the node page open class `OrganizationNodePage` in the Java editor and adapt its implementation according to [Listing 55](#).

Listing 55. The complete implementation of the class `OrganizationNodePage`.

```
@ClassId("f074181d-462a-40dc-b7cd-46bb4e50e7fb")
public class OrganizationNodePage extends AbstractPageWithNodes {

    private String organizationId; ❶

    public String getOrganizationId() {
        return organizationId;
    }

    public void setOrganizationId(String organizationId) {
        this.organizationId = organizationId;
    }

    @Override ❷
    protected void execCreateChildPages(List<IPage<?>> pageList) {
        PersonTablePage personTablePage = new PersonTablePage();
        personTablePage.setOrganizationId(getOrganizationId()); ❸
        pageList.add(personTablePage);
    }
}
```

❶ The organization id property that represents the selected organization in the parent page

❷ Method `execCreateChildPages` defines the list of child pages

❸ Define the organization id property for the person child page

We have now created an organization node page that contains a person page as its child page. The only missing step to create the discussed page hierarchy is the link between the organization page with the organization node page. Create this missing link by adding method `execCreateChildPage` to the organization page as shown in [Listing 56](#).

Listing 56. Add the organization node page as a child page to the organization page.

```
@PageData(OrganizationTablePageData.class)
@ClassId("18f7a78e-0dd0-4e4e-9234-99892bb4459f")
public class OrganizationTablePage extends AbstractPageWithTable<Table> {

    @Override
    protected IPage<?> execCreateChildPage(ITableRow row) {
        OrganizationNodePage childPage = new OrganizationNodePage();
        childPage.setOrganizationId(getTable().getOrganizationIdColumn().getValue(row));
        return childPage;
    }
}
```

The difference between Scout table pages and node pages is also reflected in the different signatures of `AbstractPageWithTable.execCreateChildPage` and `AbstractPageWithNodes.execCreateChildPages`. Table pages can have a single child page while node

pages may contain a list of child pages.

What have we achieved?

In the seventh step of the "Contacts" tutorial we have introduced a typical Scout user interface pattern for 1:n relationships. We have created a dynamic lookup call and used the lookup call to provide the data for a smart field and a smart column.

To implement a drill-down functionality for the organization table we have created a page hierarchy using the existing organization page and person page. To link the two table pages we have also created and integrated a Scout node page.

The "Contacts" application is in a clean state again and you can (re)start the backend and the frontend of the application and verify the result in your browser. As shown in [Figure 33](#) the organization specific person data is now presented in a hierarchical form in the navigation area of the application.



The screenshot shows the Scout Contacts Application interface. The left sidebar contains a navigation menu with the following items: "Contacts", "Persons", "Organizations", "Alice's Adventures in Wonderland", and "BSI Business Systems Integrati...". The "Organizations" item is expanded, showing a sub-menu with "Persons". The main content area displays a table of persons. The table has columns: "First name", "Last name", "City", "Country", and "Organization". The table contains 7 rows of data. At the bottom of the table, there is a filter bar with a search icon, a "Filter by..." input field, and status indicators: "26 rows loaded", "Reload data", and "No row selected", "Select all".

First name	Last name	City	Country	Organization
Gegor	Bauer	Aarau	Switzerland	BSI Business ...
Alexandre	Schroder	Zürich	Switzerland	BSI Business ...
André	Wegmüller	Walkringen	Switzerland	BSI Business ...
Catherine	Crowden	Schoenried	Switzerland	BSI Business ...
Cédric	Amstalden	Stands	Switzerland	BSI Business ...
Christian	Braun	Kloten	Switzerland	BSI Business ...

Figure 33. The linked person page only shows persons related to the parent organization page.

Additional Concepts and Features

This last part of the "Contacts" tutorial discusses the gap between the tutorial application and the complete "Contacts" demo application. In a number of aspects the full "Contacts" demo application comes with additional features and improvements over the functionality of its tutorial version.

The sections below highlight the following areas.

- Theming and Styling ([Theming and Styling](#))
- Modular Scout Applications ([Modular Scout Applications](#))
- Infrastructure ([Infrastructure](#))

Theming and Styling

- theme switching form for top level options menu [Figure 34](#)
- project specific theme "dark"



Figure 34. Scout application can change between different themes at runtime.

Modular Scout Applications

- separate entity "event"
- additional pattern for managing m:n relationships using table fields
- contributions to core:
 - menu to desktop
 - column to person page
 - tab to person form

- event page as child page of organization page

Infrastructure

- maven module that combines frontend and backend into a single war file
- docker integration
 - docker-maven-plugin in pom.xml
 - Dockerfile

Git configuration

If you want to add the created application to a Git repository there might some configurations be helpful. If there are no plans to use Git, this chapter can be skipped.

E.g. it is best practice to exclude some files from adding to a Git repository. These exclusions can be configured by creating a file named `.gitignore` in the root folder of the repository (see the [Git Documentation](#) for details). Here is a sample file that might be used as starting point:

```
# Git
*.orig

# Maven
target/
.surefire-*
.flattened-pom.xml

# Node
node_modules/
test-results/

# Do not check in any log files
*.log
```



Do you want to improve this document? Have a look at the [sources](#) on GitHub.