Eclipse Scout
an Introduction

Matthias Zimmermann

Version 6.0.0-SNAPSHOT

Table of Contents

Preface

Today, the Java platform is widely seen as the primary choice for implementing enterprise applications. While many successful frameworks support the development of persistence layers and business services, implementing front-ends in a simple and clean way remains a challenge. This is exactly where Eclipse Scout fits in. The primary goal of Scout is to make your life as a developer easier and to help organisations to save money and time. For this, the Scout framework covers most of the recurring front-end aspects such as user authentication, client-server communication and the user interface. This comprehensive scope reduces the amount of necessary boiler plate code, and let developers concentrate on understanding and implementing business functionality.

The purpose of this book is to get the reader familiar with the Scout framework. In this book Scout's core features are introduced and explained using many practical examples. And as both the Scout framework and Scout applications are written in Java, we make the assumption that you are familiar with the language too. Ideally, you have worked with Java for some time now and feel comfortable with the basic language features.

In the first part of the book a general introduction into the runtime part of the framework and the tooling - the Scout SDK - is provided. After the mandatory "Hello World!" application, the book walks you though a complete client server application including database access. The focus of the book's second part is on the front-end side of Scout applications. First, an overview of the Scout client model is introduced before Scout's most important UI components are described based on the Scout widget demo application. To cover the the server-side of Scout applications, an additional part of the book is planned to be released jointly with version 5.0 of the Scout framework. And finally, we intend to amend the book regarding building, testing and continuous integration for Scout applications.

Last but not least, we thank you for your interest in Scout, for being part of our community and for your friendly support of new community members. To allow for contributions to this book, the technical setup and the book's licence have been selected to minimize restrictions. According to the terms of the Creative Commons (CC-BY) license, you are allowed to freely use, share and adapt this book. All source files of the book including the Scout projects described in the book are available on github. For the first edition of this book, we did already receive a number of bug reports and comments that were pointing out mistakes, inconsistencies and suggestions for changes. This feedback is very valuable to us as it helps to improve both the book's content and the quality for all future readers. We hope that this book helps you to get started quickly and would love to get your feedback.

Chapter 1. Introduction

1.1. What is Scout?

Scout is an open source framework for building business applications. The Scout framework covers most recurring aspects of a classical client server architecture with a strong focus on the application's front-end. With its multi-device capability, a Scout client applications may run simultaneously as a rich client, in the browser and on mobile and tablet devices.

To different groups of people, Scout means different things. End users are interested in a good usability, the management cares about the benefits a new framework can offers to the organisation and developers want to know if a framework is simple to use and helps them to solve practical issues. This is why the text below describes Scout from the perspective of these three roles.

1.1.1. End User Perspective

End users of enterprise applications care about friendly user interfaces (UI) and well designed functionality that support them in their everyday work. Depending on the current context/location of an end user, either desktop, web or mobile clients work best. If working in the office, a good integration of the enterprise software with Lotus Notes or Microsoft Office often help to boost the users productivity. As office software is typically installed locally on the users PC, integrating this software also requires a desktop client for the enterprise application. When a user is working on a computer outside of his company where the enterprise client is not installed (or the user lacks the permissions to install any software), the natural choice is to work with a web application. And when the user is on the move or sitting in a meeting, the only meaningful option is to work with a mobile device.

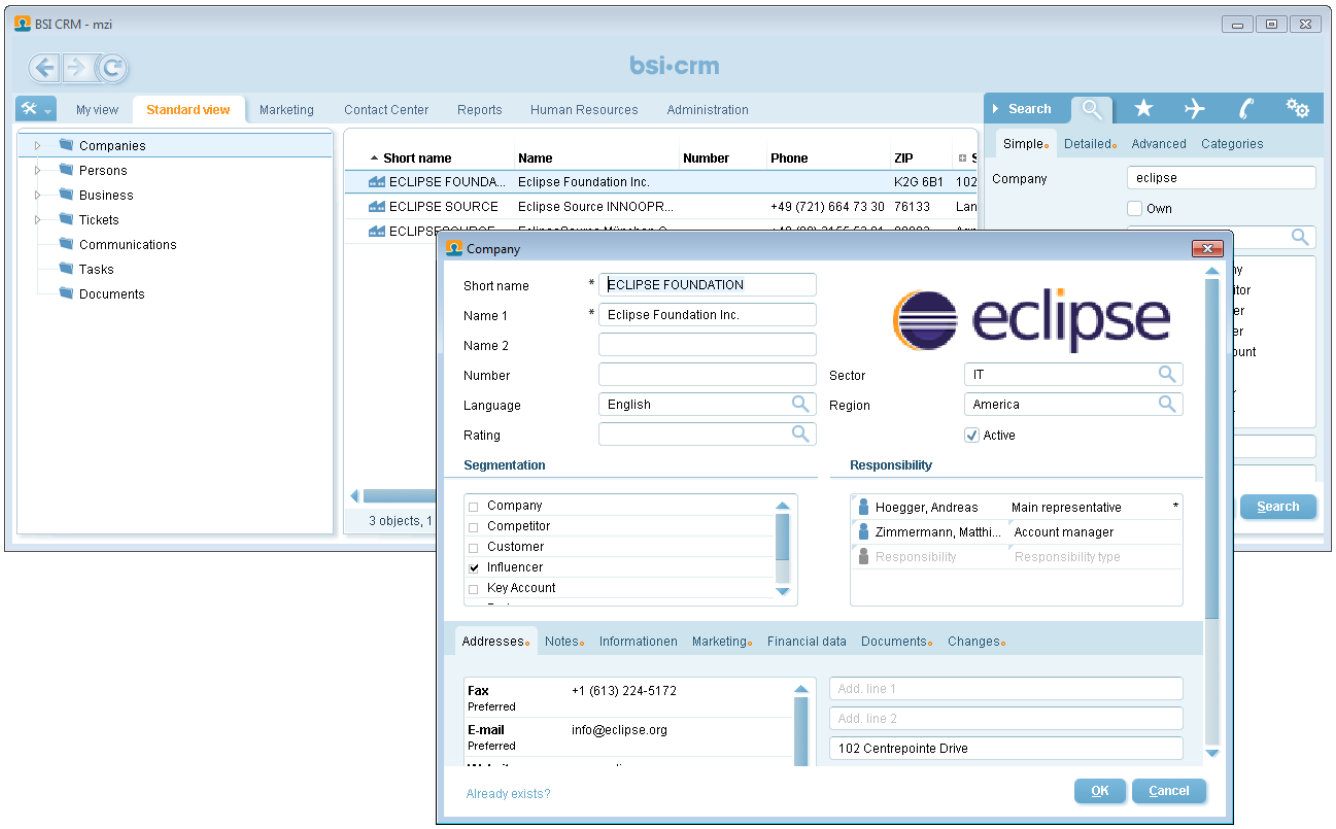


Figure 1. The desktop client of a Scout enterprise application.

To provide a concrete example, we briefly describe a real world enterprise application based on Scout. A first screenshot of a Scout desktop client is provided in Figure 000. The screenshot provides an overview of the layout of a customer relationship management (CRM) solution. On the left hand side, an entity class such as companies can be selected. Once an entity such is selected, a form is presented on the right hand side to enter the search criteria. After entering Òeclipseó into the company search field, the list of matching companies is presented. Using the context menu on a specific company, the corresponding company dialog can be opened for editing.

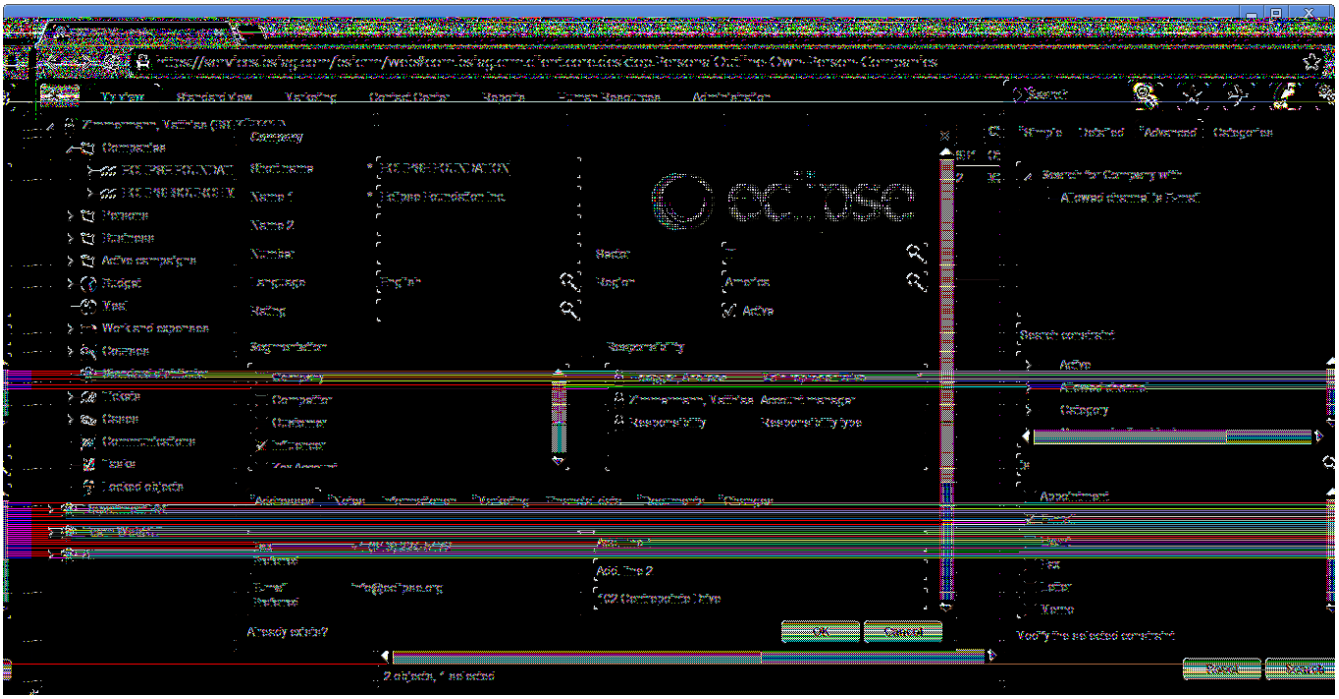


Figure 2. A Scout enterprise application running in a web browser.

In Figure 000 a screenshot of the web client of the CRM Scout application is shown. When comparing the screenshots of the desktop client with the web application it is interesting to note how Scout applications offer a consistent look and feel for the two clients. This is important as it makes the end user feel Òat homeÓ on the web client.

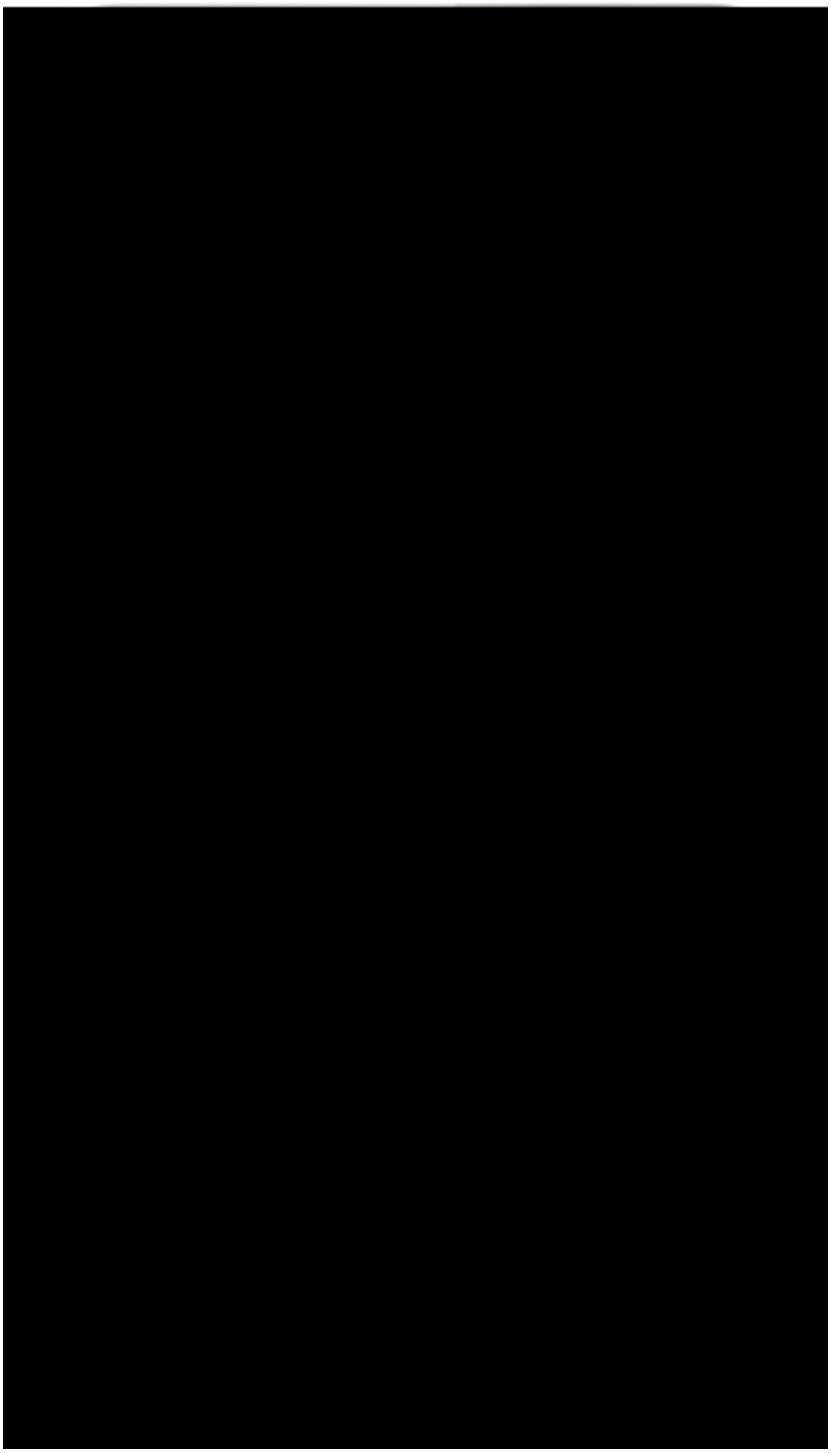


Figure 3. The same Scout enterprise application running on a mobile device.

Finally, Figure 000 provides a screenshot of the now familiar CRM application. In contrast to desktop and web applications, most tablets and mobile phones are controlled using touch features instead of mouse clicks. In addition, less elements may be presented on a single screen compared to desktop devices. These two aspects makes it impractical to directly reuse the desktop user interface on mobile devices. The look and feel still relates to the desktop and web clients but is optimized to the different form factor of the mobile device. And the end user benefits from the identical

behaviour and the the known functionality of the application.

Comparing the company table shown in the background of Figure 000 with Figure 000 it can be observed that the multi-column table of the desktop client has been transformed into a list on the mobile device. In addition, the context menu ÒNew companyÓ is now provided as a touch button. As the navigation in the application and the offered choices remain the same for Scout desktop and mobile applications, the end user feels immediately comfortable working with Scout mobile applications.

1.1.2. Management Perspective

For the management, Scout is best explained in terms of benefits it brings to the organisation in question. This is why we are going to concentrate on a (typical) application migration scenario here. Let us assume that to support the companyÕs business, a fairly large landscape of multi-tier applications has to be maintained and developed. Including host systems, client server applications with desktop clients, as well as applications with a web based front-end.

Figure 4. A typical application landscape including a service bus and a Scout application.

Usually, these applications interact with each other through a service bus as shown in Figure 000. Often, some of the applications that are vital to the organisationÕs core business have grown historically and are based on legacy technologies. And for technologies that are no longer under active development it can get difficult to find staff having the necessary expertise or motivation. Sometimes, the organisation is no longer willing to accept the costs and technology risks of such mission critical applications.

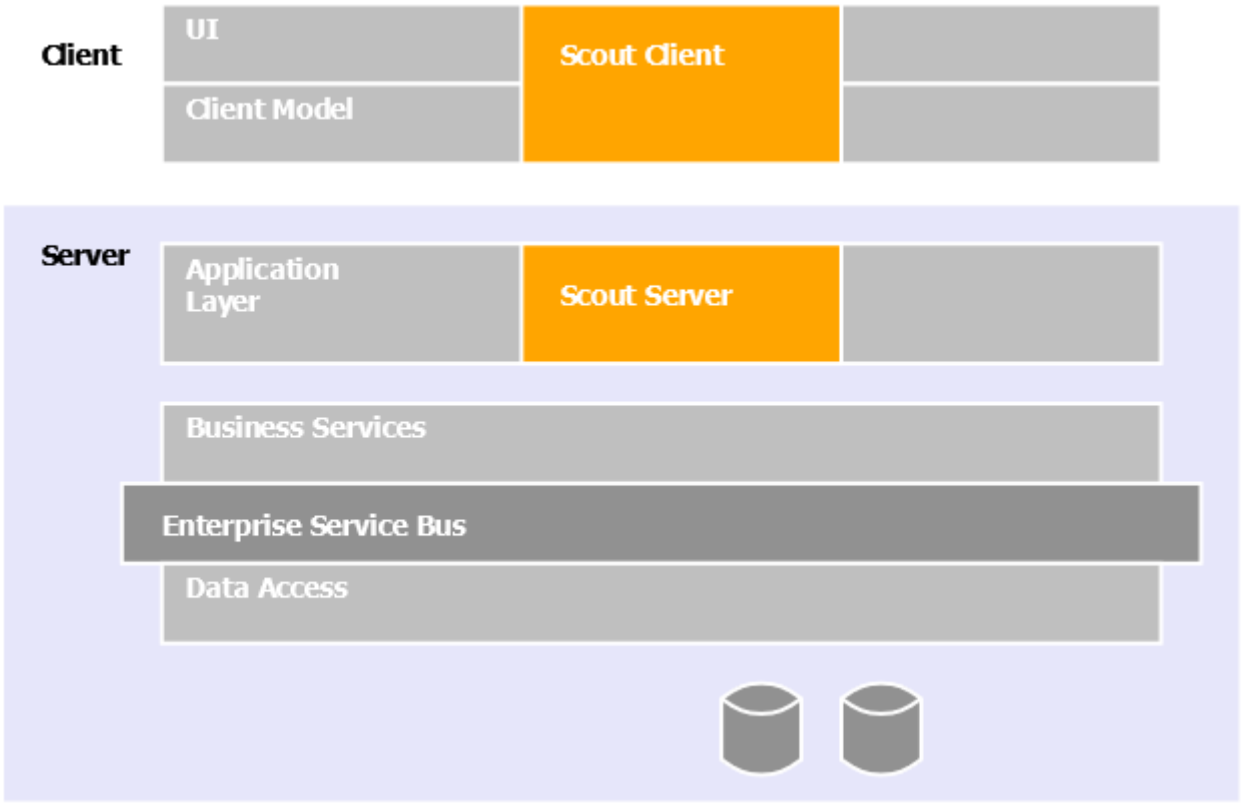


Figure 5. The integration of a Scout application in a typical enterprise setup.

In this situation, the company needs to evaluate if it should buy a new standard product or if the old application has to be migrated to a new technology stack. Now let us assume, that available products do not fit the company's requirements well enough and we have to settle for the migration scenario. In the target architecture, a clean layering similar to the one shown in Figure 000 is often desirable.

While a number of modern and established technologies exist that address the backend side (data bases, data access and business services), the situation is different for the UI layer and the application layer. The number of frameworks to develop web applications with Java is excessively large. [1: Web application framework comparison: http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks#Java.], but the choice between desktop application technologies in the Java domain is restricted to three options only. Swing, SWT and JavaFX. Both Eclipse SWT and Java Swing are mature and well established but Swing is moving into 'maintenance only' mode and will be replaced by JavaFX. However, the maturity of the new JavaFX technology in large complex enterprise applications is not yet established. Obviously, deciding for the right UI technology is a challenge and needs to be made very carefully. Reverting this decision late in a project or after going into production can get very expensive and time consuming.

Once the organisation has decided for a specific UI technology, additional components and frameworks need to be evaluated to cover client server communication, requirements for the application layer, and integration into the existing application landscape. To avoid drowning in the integration effort for all the elements necessary to cover the UI and the application layer a 'lightweight' framework is frequently developed. When available, this framework initially leads to desirable gains in productivity. Unfortunately, such frameworks often become legacy by themselves. Setting up a dedicated team to actively maintain the framework and adapt to new

technologies can reduce this risk. But then again, such a strategy is expensive and developing business application frameworks is usually not the core business of a company.

Can we do better? To implement a business application that covers the UI and the application layer as shown in Figure 000, Eclipse Scout substantially reduces both risk and costs compared to the inhouse development presented above. First or all, Scout is completely based on Java and Eclipse. Chances are, that developers are already familiar with some of these technologies. This helps in getting developers up to speed and keeping training costs low.

On the UI side, Scoutˆs multi-device support almost allows to skip the decision for a specific UI technology. Should a particular web framework become the de-facto standard in the next years, it will be the responsibility of the Scout framework to provide the necessary support. Existing Scout applications can then switch to this new technology with only minimal effort. This is possible because the Scout developers are designing and building the UI of an application using Scoutˆs client model. And this client model is not linked to any specific UI technology. Rather, specific UI renderers provided by the Scout framework are responsible to draw the UI at runtime.

As Scout is an open source project, no licence fees are collected. Taking advantage of the growing popularity of Scout, free community support is available via a dedicated forum. At the same time, professional support is available if the organisation decides for it.

As the migration of aging applications to current technology is always a challenge, it surely helps to have Scout in the technology portfolio. Not only is it a low risk choice, but also boosts developer productivity and helps to motivate the development team. Additional reasons on why Scout helps to drive down cost and risks are discussed in Section Why Scout?.

1.1.3. Developer Perspective

From the perspective of application developers, Scout offers a Java based framework that covers the complete client server architecture. This implies that Ñ once familiar with the Scout framework Ñ the developer can concentrate on a single framework language (Java) and a single set of development tools.

As Scout is completely based on Java and Eclipse, Scout developers can take full advantage of existing knowledge and experience in these domains. And to make learning Scout as simple as possible, Scout includes a comprehensive software development kit (SDK), the Scout SDK. The Scout SDK helps to create a robust initial project setup for client server applications and includes a large set of wizards for repetitive and error prone tasks.

On the client-side Scoutˆs flexible client model allows the developer to create a good user experience without having to care about specific UI technologies. The reason for this can be found in Scoutˆs client architecture that cleanly separates the UI model from the UI technology. In Scout (almost) every UI component is implemented four times. First the implementation of the UI model component and then, three rendering components for each UI technology supported by Scout. For desktop clients these are the Swing and the SWT technologies, and for the web and mobile support this is Eclipse RAP which in turn takes care of the necessary JavaScript parts.

Not having to worry about Swing, SWT or JavaScript can significantly boost the productivity. With one exception. If a specific UI widget is missing for the user story to be implemented, the Scout

developer first needs to implement such a widget. Initially, this task is slightly more complex than not working with Scout. For custom widgets the Scout developer needs to implement both a model component and a rendering component for a specific UI technology. But as soon as the client application needs to be available on more than a single frontend, the investment already pays off. The developer already did implement the model component and only needs to provide an additional rendering component for the new UI technology. In most situations the large set of Scouts UI components provided out-of-the box are sufficient and user friendly applications are straight forward to implement. Even if the application needs to run on different target devices simultaneously.

Client-server communication is an additional aspect where the developers is supported by Scout. Calling remote services in the client application that are provided by the Scout server looks identical to the invocation of local services. The complete communication including the transfer of parameter objects is handled fully transparent by the Scout framework. In addition, the Scout SDK can completely manage the necessary transfer objects to fetch data from the Scout server that is to be shown in dialog forms on the Scout client. The binding of the transferred data to the form fields is done by the framework.

Although the Scout SDK wizards can generate a significant amount of code, there is no one-way code generation and no meta data in a Scout application. Just the Java code. [2: With the exception of the plugin.xml and MANIFEST.MF files required for Eclipse plugins.]. Developers preferring to write the necessary code manually, may do so. The Scout SDK parses the application's Java code in the background to present the updated Scout application model to the developers preferring to work with the Scout SDK.

Finally, Scout is an open source framework hosted at the Eclipse foundation. This provides a number of interesting options to developers that are not available for closed source frameworks. First of all, it is simple to get all the source code of Scout and the underlying Eclipse platform. This allows for complete debugging of all problems and errors found in Scout applications. Starting from the application code, including the Scout framework, Eclipse and down to the Java platform.

Scout developer can also profit from an increasing amount of free and publicly available documentation, such as this book or the Scout Wiki pages. And problems with Scout or questions that are not clearly addressed by existing documentation can be discussed in the Scout forum. The forum is also a great place for Scout developers to help out in tricky situation and learn from others. Ideally, answered questions lead to improved or additional documentation in the Scout Wiki.

At times, framework bugs can be identified from questions asked in the forum. As all other enhancement requests and issues, such bugs can be reported in Bugzilla by the Scout developer. Using Bugzilla, Scout developers can also contribute bug analysis and patch proposals to solve the reported issue. With this process, Scout developers can actively contribute to the code base of Eclipse Scout. This has the advantage, that workarounds in existing Scout applications can be removed when an upgrade of the Scout framework is made.

Having provided a significant number of high quality patches and a meaningful involvement in the Scout community, the Scout project can nominate a Scout developer as a new Scout committer. Fundamentally, such a nomination is based on the trust of Scout committers in the candidate. To quote the official guidelines. [3: Nominating and electing a new Eclipse Scout committer:

http://wiki.eclipse.org/Development_Resources/HOWTO/Nominating_and_Electing_a_New_Committer#Guidelines_for_Nominating_and_Electing_a_New_Committer.] for nominating and electing a new committer:

> A Committer gains voting rights allowing them to affect the future of the Project. Becoming a Committer is a privilege that is earned by contributing and showing discipline and good judgment. It is a responsibility that should be neither given nor taken lightly, nor is it a right based on employment by an Eclipse Member company or any company employing existing committers.

After a successful election process (existing committers voting for and not against the candidate) the Scout developer effectively becomes a Scout committer. With this new status, the Scout developer then gets write access to the Eclipse Scout repositories and gains voting rights and the possibility to shape the future of Scout.

1.2. Why Scout?

Most large organizations develop and maintain enterprise applications that have a direct impact on the success of the ongoing business. And at the same time, those responsible for the development and maintenance of these applications struggle with this task. It is a big challenge to adapt to changing business demands and complying with the latest legal requirements in time. And the increasing pressure to lower recurring maintenance costs does not make the situation any easier.

It often seems that too many resources are required to keep a heterogeneous set of legacy technologies alive. In this situation, modernizing mission critical applications can help to improve over the current situation. For the target platform stack, Java is a natural choice as it is mature, widely adopted by in the industries and unlikely to become legacy in the foreseeable future. While for the back-end side of enterprise applications well-known and proven frameworks do exist, the situation on the client side is less clear. Unfortunately, user interface (UI) technologies often have lifetimes that are substantially shorter than the lifetimes of larger mission critical applications. This is particularly true for the web, where many of today's frameworks will no longer be relevant in five or more years.

Enter Eclipse Scout. This open source framework covers most of the recurring needs that are relevant to the front-end development of business applications. And Scout forces a clean separation between the user interface and the specific UI technology used for rendering. This has two major benefits. First, Scout developers implement the user interface against an abstraction layer, which helps to focus on the business functionality and saves development time. And second, long term maintenance costs are lower, as the Scout code remains valid even when the rendering technology needs to be exchanged. Therefore, Scout helps to improve the productivity of the development teams and reduces the risk of major application rewrites.

To provide a first impression on the scope and goals of the Scout framework, a number of scenarios where Scout typically contributes to your projects success are listed below .

- You are looking for a reasonable client side framework for your business application.

- You need an application that works on the desktop, in browsers and on mobiles devices.
- You don't have the time to evaluate and learn a new UI technology.
- You need a working prototype application by the end of the week.
- Your application's expected lifespan is 10 years or more.

That Scout should help in the last two situations mentioned above seems to be contradictory at first but is just based on a simple principle. Where possible, the Scout framework provides abstractions for areas/topics. [4: Example areas/topics that are abstracted by the Scout framework are user interface (UI) technologies, databases, client-server communication or logging.] that need to be implemented for business applications again and again. And for each of these abstractions Scout provides a default implementation out of the box. Typically, the default implementation of such an abstraction integrates a framework or technology that is commonly used.

When needing a working prototype application by the end of the week, the developer just needs to care about the desired functionality. The necessary default implementations are then automatically included by the Scout tooling into the Scout project setup. The provided Scout SDK tooling also helps to get started quickly with Scout. It also allows to efficiently implement application components such as user interface components, server services or connections to databases.

In the case of applications with long lifespans, the abstractions provided by Scout help the developer to stay productive and concentrate on the actual business functionality. At the same time, this keeps the code base as independent of specific technologies and frameworks as possible. This is a big advantage when individual technologies incorporated in the application reach their end of life. As all the implemented business functionality is written against abstractions only, no big rewrite of the application is necessary. Instead, it is sufficient to exchange the implementation for the legacy technology with a new one. And often, an implementation for a new technology/framework is already provided by a more recent version of Scout.

1.3. What should I read?

The text below provides guidelines on what to read (or what to skip) depending on your existing background. We first address the needs of junior Java developers that like to learn more about developing enterprise applications. Then, we suggest a list of sections relevant for software wizards that already have a solid understanding of the Eclipse platform, Java enterprise technologies, and real world applications. Finally, the information needs of IT managers are considered.

1.3.1. I know Java

The good news first. This book is written for you! For the purpose of this book we do not assume that you have a solid understanding of the Java Enterprise Edition (Java EE) is required. [5: Java Enterprise Edition: http://en.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition] . However, having prior experience in client server programming with Java will be useful occasionally and having used the Eclipse IDE for Java development before is certainly a plus.

The "bad" news is, that writing Scout applications requires a solid understanding of Java. To properly benefit from this book, we assume that you have been developing software for a year or more. And you should have mastered the Java Standard Edition (Java SE). [6: Java Standard Edition:

http://en.wikipedia.org/wiki/Java_SE] to a significant extent. To be more explicit, you are expected to be comfortable with all material required for the Java Programmer Level I Exam. [7: Level I Exam: docs.oracle.com/javase/tutorial/extra/certification/javase-7-programmer1.html] and most of the material required for Level II. [8: Level II Exam: docs.oracle.com/javase/tutorial/extra/certification/javase-7-programmer2.html].

We now propose to start downloading and installing Scout as described in Appendix Scout Installation and do some actual coding. To do so, please continue with the ÒHello WorldÓ example provided in Chapter ÒHello WorldÓ Tutorial. You can expect to complete this example in less than one hour including the necessary download and installation steps. Afterwards, you might want to continue with the remaining material in ÒGetting StartedÓ. Working through the complete book should take no more than two days.

Once you work with the Scout framework on a regular basis, you might want to ask questions in the Scout forum. [9: Eclipse Scout forum: http://www.eclipse.org/forums/eclipse.scout]. When your question gets answered, please ask yourself if your initial problem could have been solved by better documentation. In that case, you might want to help the Scout community by fixing or amending the Scout wiki pages. [10: Eclipse Scout wiki: http://wiki.eclipse.org/Scout]. Or this book. If you find a bug in Eclipse Scout that makes your life miserable you can report it or even propose a patch. And when your bug is fixed, you can test the fix. All of these actions will add to the healthy grow of the Scout community.

1.3.2. I am a manager

Being a manager and actually reading this book may indicate one of the following situations:

- Your developer tried to convince you that Eclipse Scout can help you with implementing business applications in a shorter time and for less money. And you did not understand why (again) a new technology should work better than the ones you already use.
- Your are a product manager of a valuable product that is based on legacy technology. And you are now evaluating options to modernize your product.
- Think about your current situation. There must be a reason why you are checking out this book.

To learn about Scout and about its benefits first go through Section What is Scout? and Section Why Scout?. Then, flip through Section The ÒContactsÓ Application to get an impression of the ÒMy ContactsÓ application. In case you like the idea that your developers should be able to build such an application in a single day, you might want to talk to us. [11: To contact the Scout team, use the feedback provided on the Scout homepage: https://eclipse.org/scout.].

Chapter 2. “Hello World” Tutorial

The ÒHello WorldÓ chapter walks you through the creation of an Eclipse Scout client server application. When the user starts the client part of this application, the client connects to the server. [12: The Scout server part of the ÒHello WorldÓ application will be running on a web server.] and asks for some text content that is to be displayed to the user. Next, the server retrieves the desired information and sends it back to the client. The client then copies the content obtained from the server into a text field widget. Finally, the client displays the message obtained from the server in a text field widget.

The goal of this chapter is to provide a first impression of working with the Scout framework using the Scout SDK. We will start by building the application from scratch and then weÒll deploy the complete application to a Tomcat web server.

Based on this simple ÒHello WorldÓ applications a large number of Scout concepts can be illustrated. Rather than including such background material in this tutorial, this information is provided separately in Chapter [cha-helloworld_background]. This tutorial is also available in the Scout wiki. [13: ÒHello WorldÓ wiki tutorial: http://wiki.eclipse.org/Scout/Tutorial/4.0/HelloWorld].

2.1. Installation and Setup

Before you can start with the ÒHello WorldÓ example you need to have a complete and working Scout installation. For this, see the step-by-step installation guide provided in Appendix Scout Installation. Once you have everything installed, you are ready to create your first Scout project.

2.2. Create a new Project

Start your Eclipse IDE and select an empty directory for your workspace as shown in Figure Start the Eclipse IDE with a new project folder. This workspace directory will then hold all the project code for the `Hello World` application. Once the Eclipse IDE is running it will show the Java perspective.

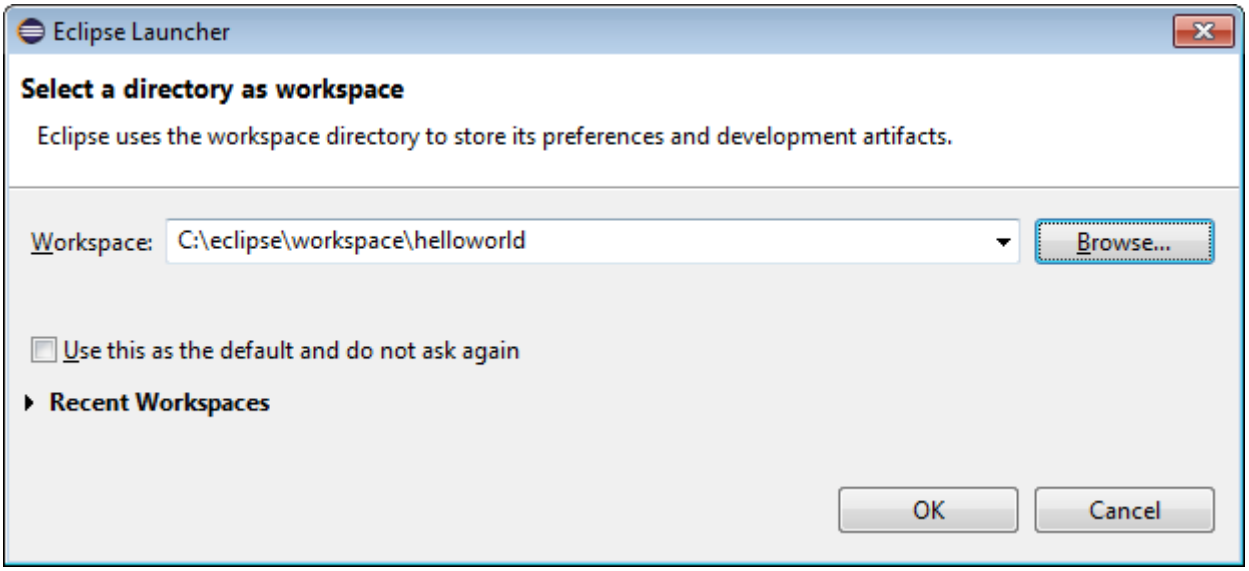


Figure 6. Select a new empty folder to hold your project workspace

To create a new Scout project select the menu **File | New | Project...** and type ÒScout ProjectÓ in the wizard search field. Select the Scout Project wizard and press **[Next]**. The *New Scout Project* wizard is then started as shown in Figure New Scout Project Wizard.

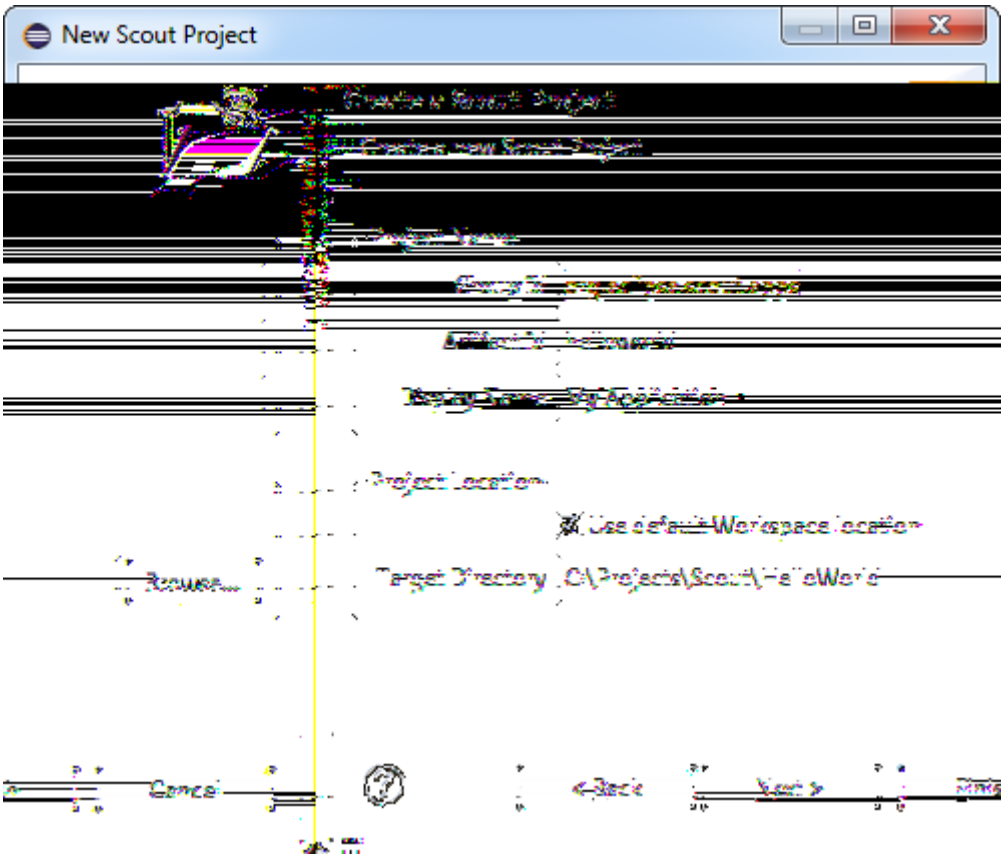


Figure 7. The new Scout project wizard.

In the *New Scout Project* wizard you have to enter a `group id`, `artifact id` and a `display name` for your Scout project. As the created project will make use of Apache Maven please refer to the Maven naming conventions to choose `group id` and `artifact id` for your project. The `artifact id` will then also be the project name in the Eclipse workspace. The `display name` is used as the application name presented to the user (e.g. in the Browser title bar).

For the `Hello World` application just use the already prefilled values as sown in Figure New Scout Project Wizard. Then, click the **[Finish]** button to let the Scout SDK create the initial project code for you.

Depending on your Eclipse installation some `Maven plugin connectors` may be missing initially. In that case a dialog as shown in Figure Maven plugin connector installation dialog may be shown. To continue click on **[Finish]** to resolve the selected connectors. Afterwards confirm the installation, accept the license and the message that some content has not been signed. Finally, the installation of the maven plugin connectors requires a restart of the Eclipse IDE.

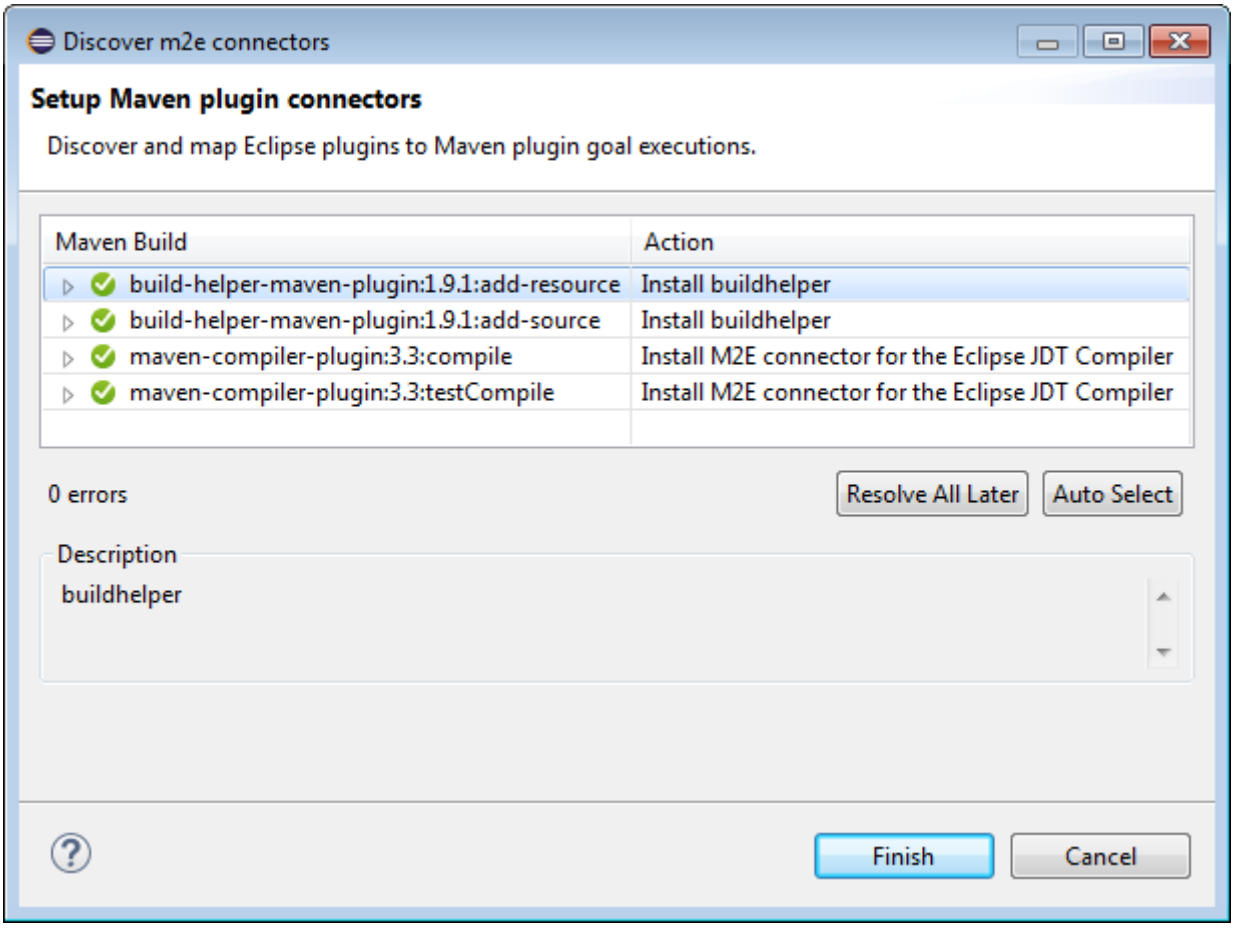


Figure 8. The Maven plugin connector installation dialog.

After the *New Scout Project* wizard has created the initial Maven modules for the `Hello World` application these modules are compiled and built by the Eclipse IDE. In case of a successful Eclipse Scout installation your Eclipse IDE should display all created Maven modules in the Package Explorer and have an empty Problems view as shown in Figure The Hello World Maven Modules.

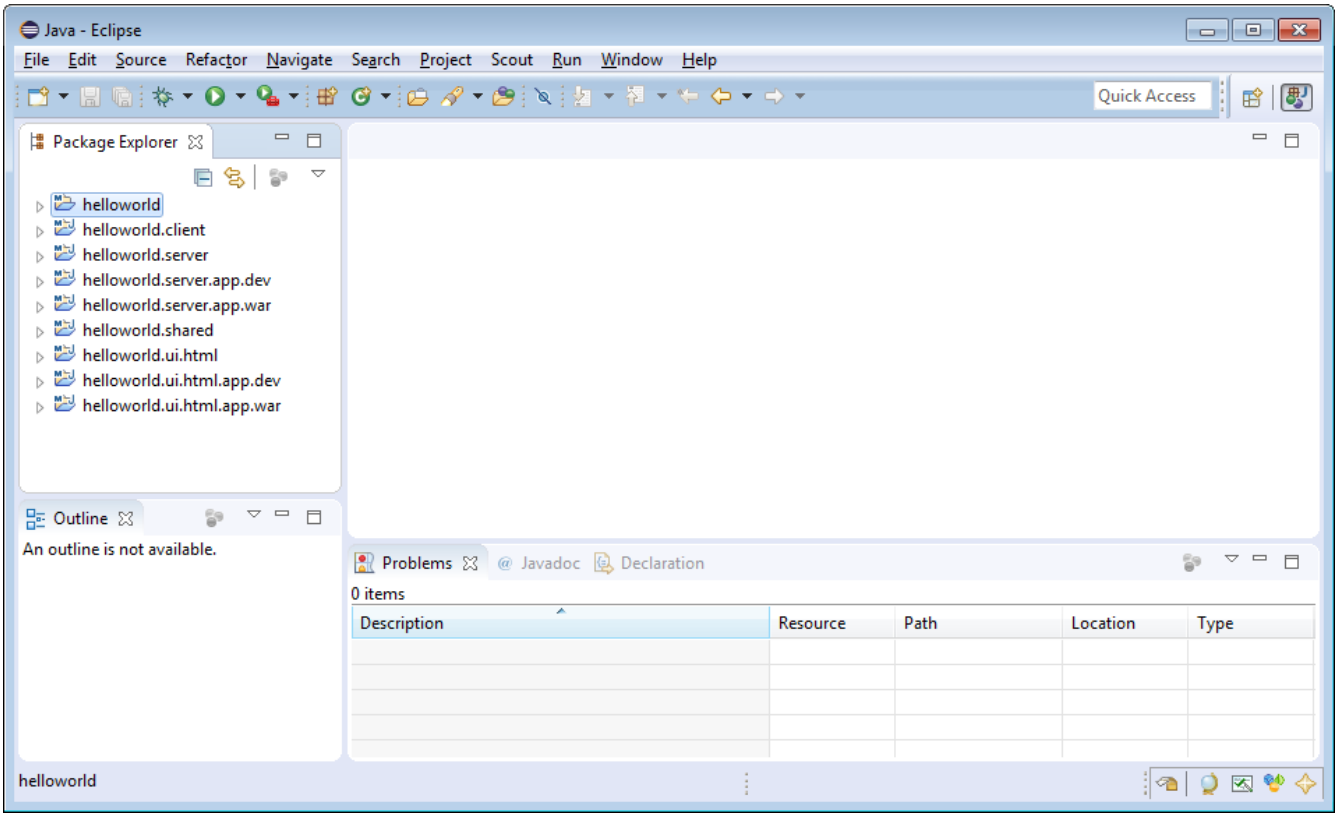


Figure 9. The inital set of Maven modules created for the Hello World application.

2.3. Run the Initial Application

After the initial project creation step we can start the Scout application for the first time. For this, the following three steps are necessary

1. Start the Scout backend server
2. Start the Scout frontend server
3. Open the application in the browser

To start the Scout backend server we first select the `[webapp] dev server.launch` file in the Package Explorer view of the Eclipse IDE and then use the *Run As* menu as shown in Figure Starting the Hello World application.

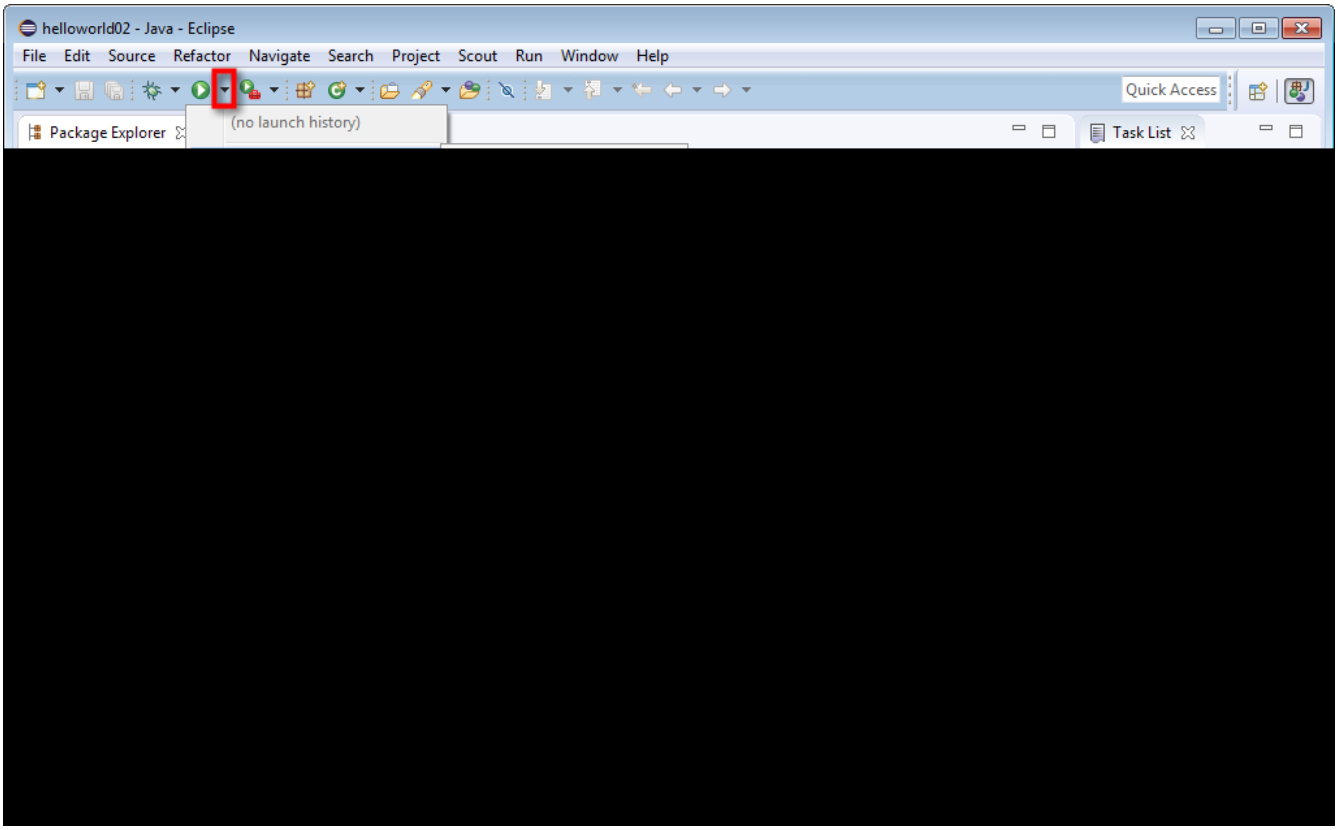


Figure 10. Starting the Hello World application.

Starting the Scout frontend server works exactely the same. But first select the `[webapp] dev ui.launch` file in the Eclipse IDE. This launch file is located under module `helloworld.ui.html.app.dev` in the Package Explorer.

During startup of the Scout applications you should see console output providing information about the startup. After having successfully started the Scout backend and frontend servers the Hello World application can then be accessed by navigating to http://localhost:8082/ in your favorite web browser.

The running Hello World application should then be started in your browser as shown in Figure The Hello World application.

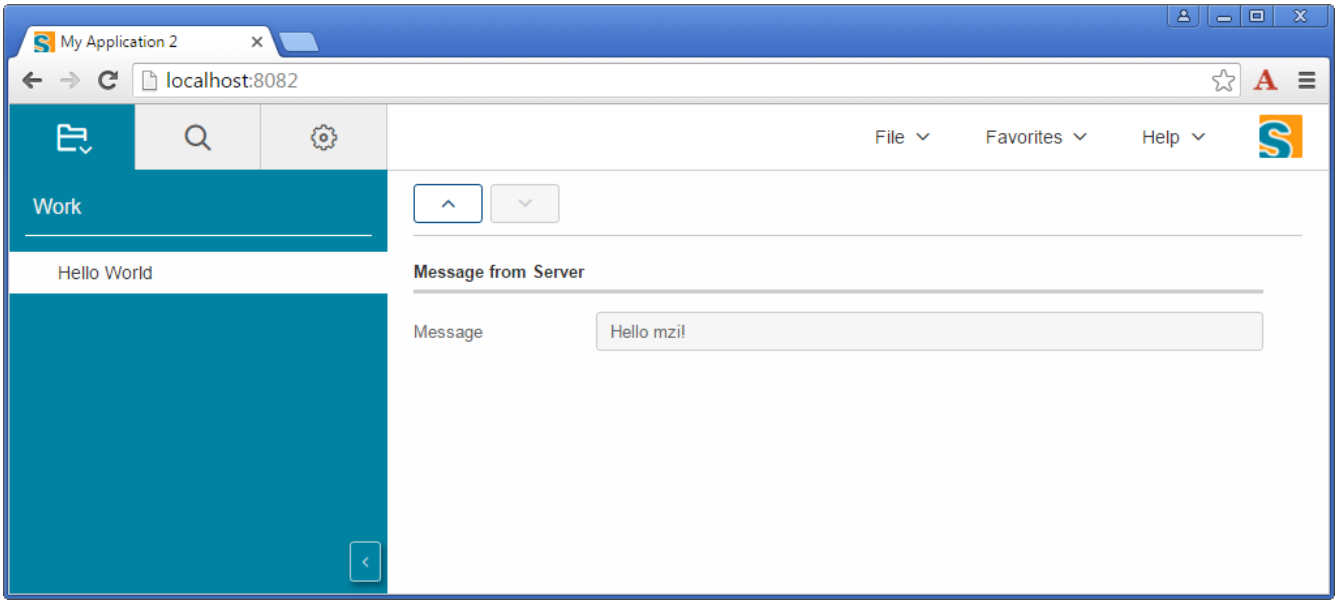


Figure 11. The Hello World application in the browser.

2.4. Export the Application

At some point during the application development you will want to install your software on a machine that is intended for productive use. This is the moment where you need to be able to build and package your Scout application in a way that can be deployed to an application server.

As Scout applications just need a servlet container to run, Scout applications can be deployed to almost any Java appliction servers. For the purpose of this tutorial we will use Apache Tomcat.

2.4.1. Verify the Container Security Settings

First you need to decide if the users of your application should communicate via HTTPS with the Scout frontend server. We strongly recommended this setup for any productive environment. This is why even the Scout ÒHello WorldÓ example is configured to use HTTPS.

As a default Tomcat installation is configured to use HTTP only, we need to first verify if the installtion is properly configured for HTTPS too. In case HTTPS support is already enabled for your Tomcat installation, you may skip this section.

Otherwise, check out the configuration process described in the Tomcat Documentation to enable SSL/TLS.

2.4.2. Create and Install a Self-Signed Certificate

This section describes the creation and usage of a self-signed certificat in a localhost setting.

1. Create a keystore file with a self-signed certificate
2. Uncomment/adapt the HTTPS connector port in TomcatÕs server.xml configuration
3. Export the self-signed certificate from the keystore
4. Import the self-signed certificate into the Java certificate store

The first step is to create a self-signed certificate using the keytool provided with the Java runtime. The example command line below will create such a certificate using the alias `tomcat_localhost` and place it into the keystore file `tomcat_localhost.jks`

```
keytool.exe -genkey -keyalg RSA -dname CN=localhost -alias tomcat_localhost -keystore
tomcat_localhost.jks -keypass changeit -storepass changeit
```

The second step is to uncomment the HTTPS connector element in the TomcatÕs `server.xml` configuration file. Make sure that parameter `keystoreFile` points to your newly created keystore file (if you are using a windows box, make sure not to use the backslash characters in the path to the keystore). After a restart of Tomcat you should then be able to access Tomcat on https://localhost:8443/manager/html

```
<Connector port="8443" protocol="org.apache.coyote.http11.Http11NioProtocol"
               maxThreads="150" SSLEnabled="true" scheme="https" secure="true"
               clientAuth="false" sslProtocol="TLS"
               keystoreFile="file:///c:/keystore/tomcat_localhost.jks" keystorePass=
"changeit"
/>
```

The third step is to export the newly created self-signed certificate from the `tomcat_localhost.jks` keystore file into the `tomcat_localhost.der` certificate file.

```
keytool.exe -exportcert -alias tomcat_localhost -storepass changeit -keystore
tomcat_localhost.jks -file tomcat_localhost.der
```

In the fourth and last step we add the self-signed certificate to the known certificates of the Java runtime. Make sure that you modify the `cacerts` file of the Java runtime that is used in your Tomcat installation and modify the path to the `cacerts` file accordingly.

```
keytool.exe -import -alias tomcat_localhost -trustcacerts -storepass changeit
-keystore C:\java\jre8\lib\security\cacerts -file tomcat_localhost.der
```

Your Scout application should now properly communicate over HTTPS in your Tomcat installation and after having installed the "Hello World" application to Tomcat it should become available on https://localhost:8443/org.eclipse.scout.apps.helloworld.ui.html/.

In case the Scout frontend server cannot access the Scout backend server your self-signed certificate might be missing in the Java installation. To verify that the certificate has been included in file `cacerts` file use the following command.

```
keytool.exe -list -storepass changeit -keystore C:\java\jre8\lib\security\cacerts |
find "localhost"
```

Once you no longer need the self-signed certificate file in your Java installation make sure to remove the certificate again.

```
keytool.exe -delete -alias tomcat_localhost -storepass changeit -keystore
C:\java\jre8\lib\security\cacerts
```

2.4.3. Update the Scout Application to work with HTTP

If you should prefer to work with HTTP only, you need to modify the security settings of your Scout application. This can be done in module `helloworld.ui.html.app.war` with the two steps described below.

¥ In file `config.properties` (in folder `src/main/resources`) add property

`scout.auth.cookie.session.validate.secure=false` to disable the check for an encrypted channel (HTTPS).

- In file `web.xml` (in folder `src/main/webapp/WEB-INF`) delete the `<secure>true</secure>` flag in the `<cookie-config>` element.

More on this topic can be found in the Scout Architecture Documentation.

2.4.4. Create WAR Files

We are now ready to move the `Hello World` application from our development environment to a productive setup. The simplest option to move our application into the 'wild' is to build it using Maven. This produces two WAR files [14: Web application ARchive (WAR): http://en.wikipedia.org/wiki/WAR_file_format_%28Sun%29].

The first WAR file contains the Scout backend server with all business logic. The second WAR file contains the Scout frontend server that is responsible for communicating with the web browser part of the Scout application.

To start the build right click on the project `helloworld` and select the context menu **Run As → Maven build...** as shown in Figure Maven build. In the dialog that appears enter `clean verify` into the `Goals` field and press **[Run]**.

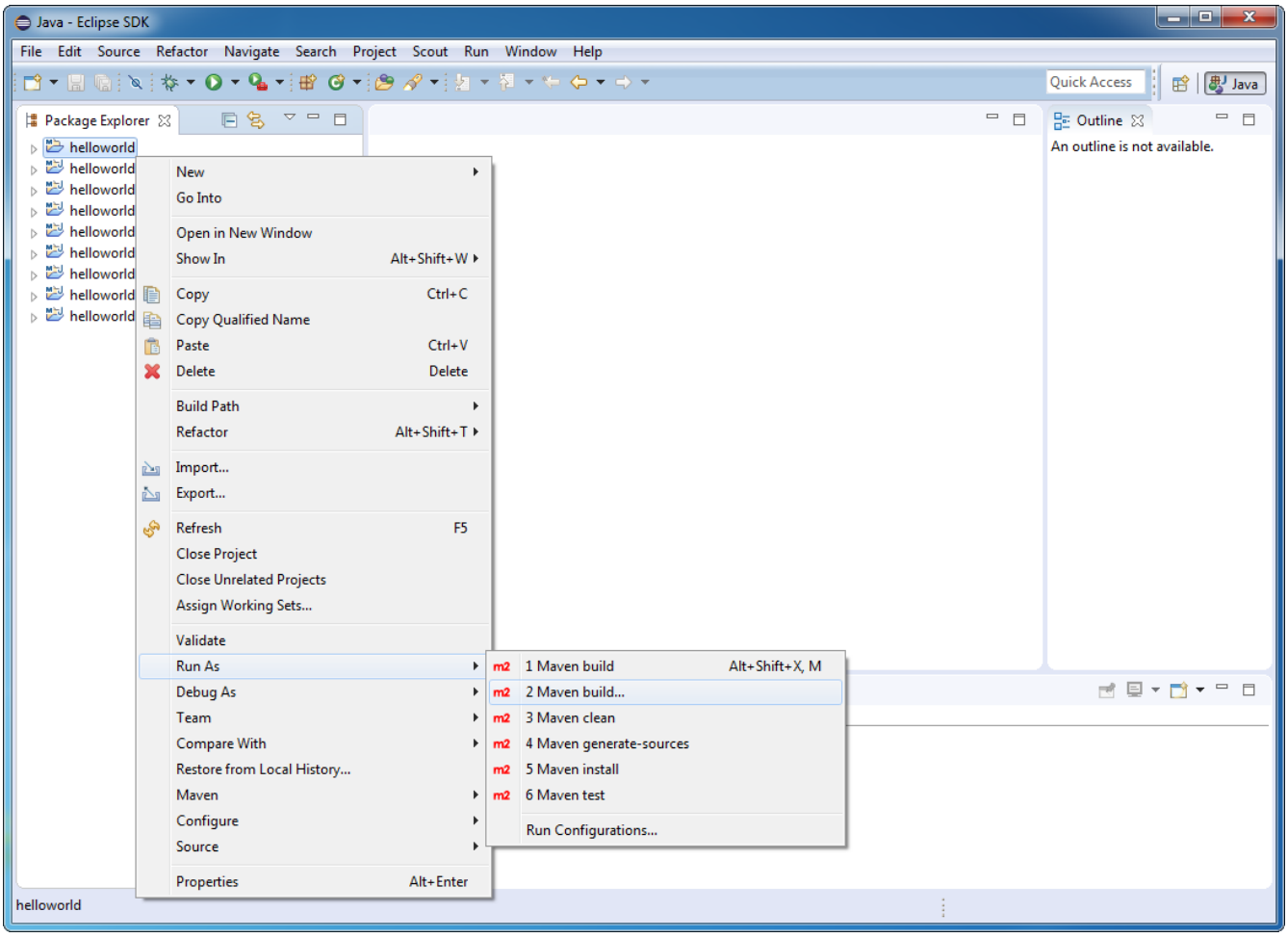


Figure 12. Starting the Maven build.

Afterwards the compilation starts, executes all test cases and bundles the result into two WAR files.

The output of the build is shown in the Console view within Eclipse. As soon as the build is reporting success you can find the built WAR files:

- The Scout backend WAR file `org.eclipse.scout.apps.helloworld.server.war` in folder `workspace_root/helloworld.server.app.war/target`
- The Scout frontend WAR file `org.eclipse.scout.apps.helloworld.ui.html.war` in folder `workspace_root/helloworld.ui.html.app.war/target`

To see the new files within Eclipse you may need to refresh the `target` folder below each project using the F5 keystroke.

2.5. Deploy to Tomcat

As the final step of this tutorial, we deploy the two WAR files representing our ÒHello WorldÓ application to a Tomcat web server. For this, we first need a working Tomcat installation. If you do not yet have such an installation you may want to read and follow the instructions provided in Appendix Apache Tomcat Installation. To verify a running Tomcat instance, type http://localhost:8080/ into the address bar of the web browser of your choice. You should then see the page shown in Tomcat start page.

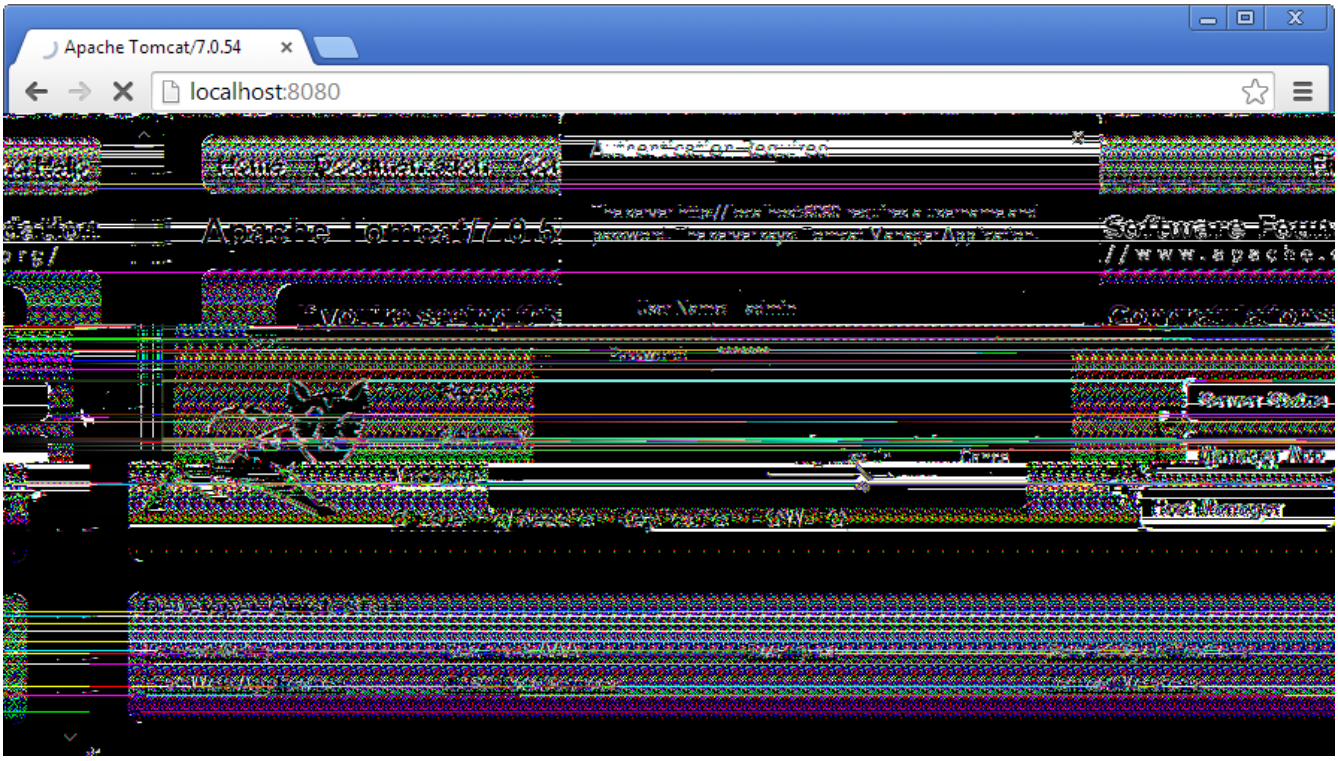


Figure 13. The Tomcat shown after a successful installation. After clicking on the "Manager App" button (highlighted in red) the login box is shown in front. A successful login shows the "Tomcat Web Application Manager".

Once the web browser displays the successful running of your Tomcat instance, switch to its ÒManager AppÓ by clicking on the button highlighted in Tomcat start page. After entering user name and password the browser will display the ÒTomcat Web Application ManagerÓ as shown in Tomcat Manager. If you don't know the correct username or password you may look it up in the file tomcat-users.xml as described in Appendix Directories and Files.

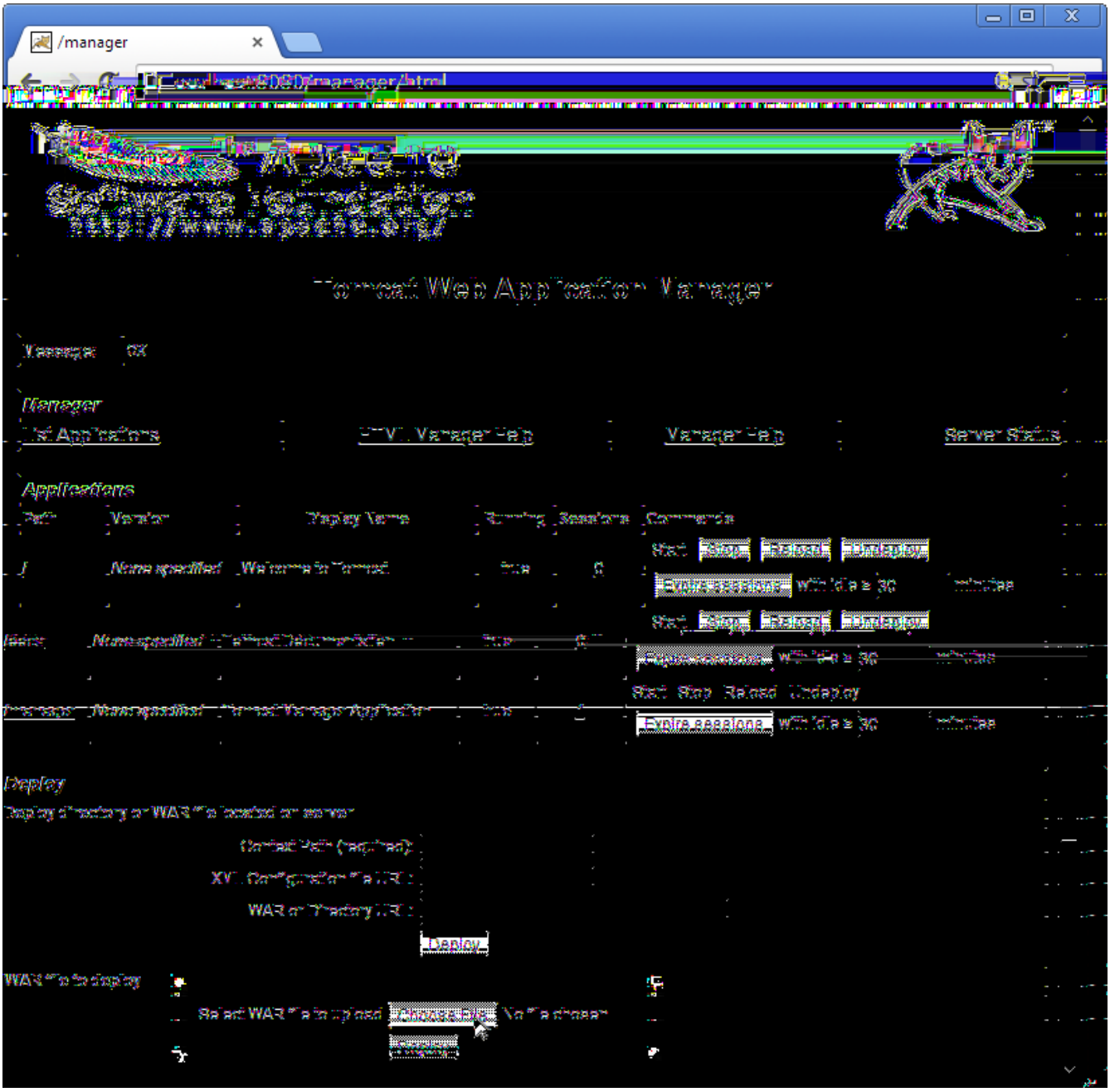


Figure 14. The “Tomcat Web Application Manager”. The WAR files to be deployed can then be selected using button “Choose File” highlighted in red.

After logging into Tomcat's manager application, you can select the WAR files to be deployed using button ÒChoose FileÓ according to the right hand side of Tomcat Manager. After picking your just built org.eclipse.scout.apps.helloworld.server.war and closing the file chooser, click on button ÒDeployÓ (located below button ÒChoose FileÓ) to deploy the application to the Tomcat web server. Then we repeat this step with the second WAR file org.eclipse.scout.apps.helloworld.ui.html.war.

This will copy the selected WAR files into Tomcats webapps directory and unpack its contents into subdirectories with the same name. You can now connect to the application using the browser of your choice and enter the following address:

```
http://localhost:8080/org.eclipse.scout.apps.helloworld.ui.html/
```

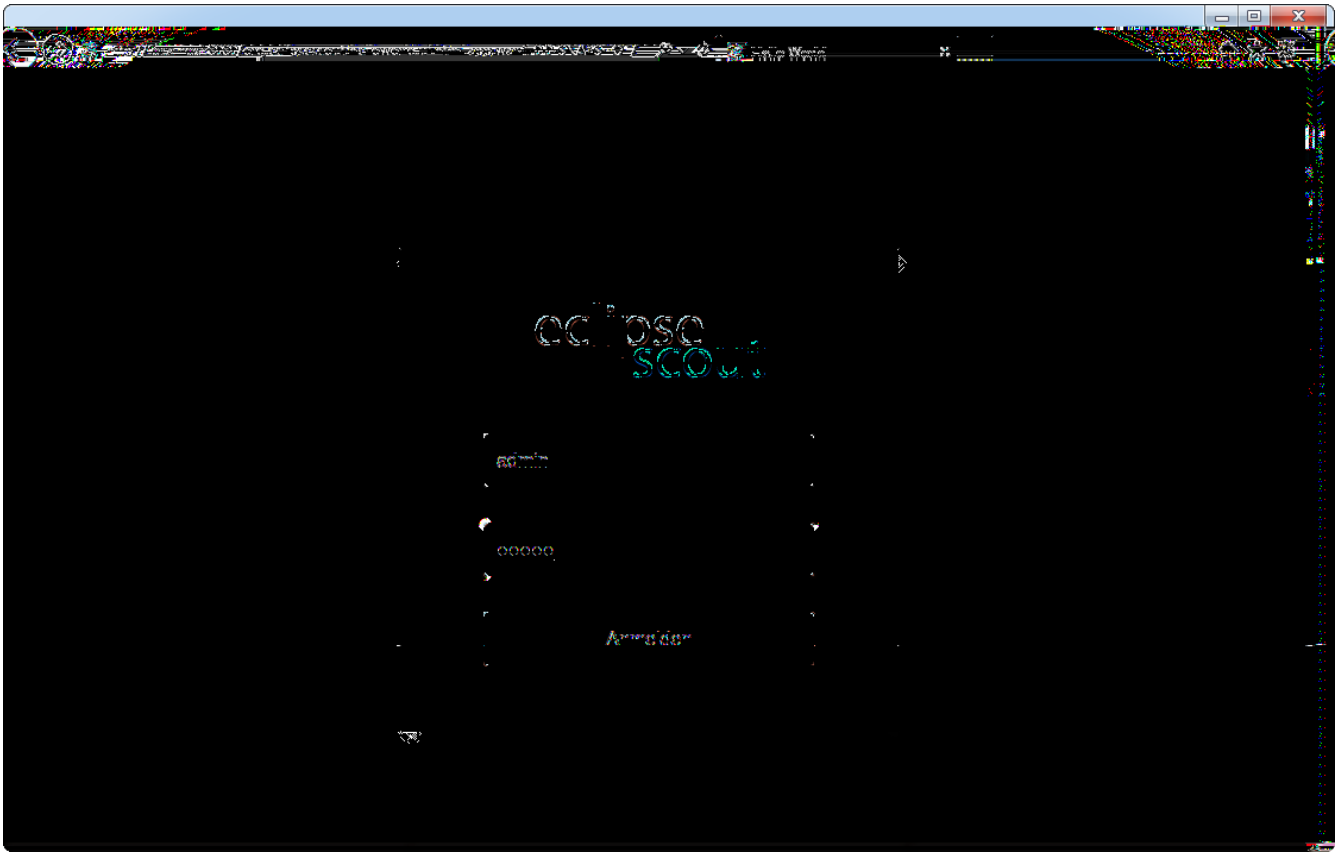


Figure 15. The “Hello World” login page.

Then you will see the login page as shown in Figure Login Page. Two users have been pre defined: Òadmin Ó with password Òadmin Ó and Òscott Ó with password Òtiger Ó. You can find this configuration in the config.properties file of the application.

Please note: In a productive environment it is recommended to deploy the server and the user interface into two different servlet containers running on dedicated machines. This because these two tiers have different requirements on resources, load balancing and access protection. Furthermore, it is strongly recommended to use an encrypted connection (e.g. TLS 1.2 [15: TLS: https://en.wikipedia.org/wiki/Transport_Layer_Security]) between client browsers and the Scout frontend server AND between the Scout frontend and and backend server!

Chapter 3. Scout Tooling

This chapter presents the Scout SDK tooling that is included with the Eclipse Scout. The Scout SDK provides wizards to create new project and application components, adds code assistance to the Java Editor and comes with a NLS editor to manage all translated text entries of the application.

The chapter is organized as follows. Section Motivation for the Tooling describes the goals and benefits of the tooling included. Because the Scout Tooling is based on the Eclipse IDE, section Eclipse IDE tooling provides a short overview of frequently used Eclipse features. A high level description of the Scout tooling is provided in section Scout SDK Overview. Sections Scout Wizards, Scout Content Assistance and Scout NLS Tooling then provide detailed descriptions of the functionality offered by the Scout SDK.

3.1. Motivation for the Tooling

Thanks to this tooling, developing Scout applications is made simpler, more productive and also more robust. Initially, a solid understanding of the Java language is sufficient to start developing Scout applications and only a rough understanding of the underlying Maven/JEE technologies is required.

The Scout SDK also helps developers to become more productive. Many repetitive and error prone tasks run automatically in the background or are taken care of by the component wizards of the Scout SDK.

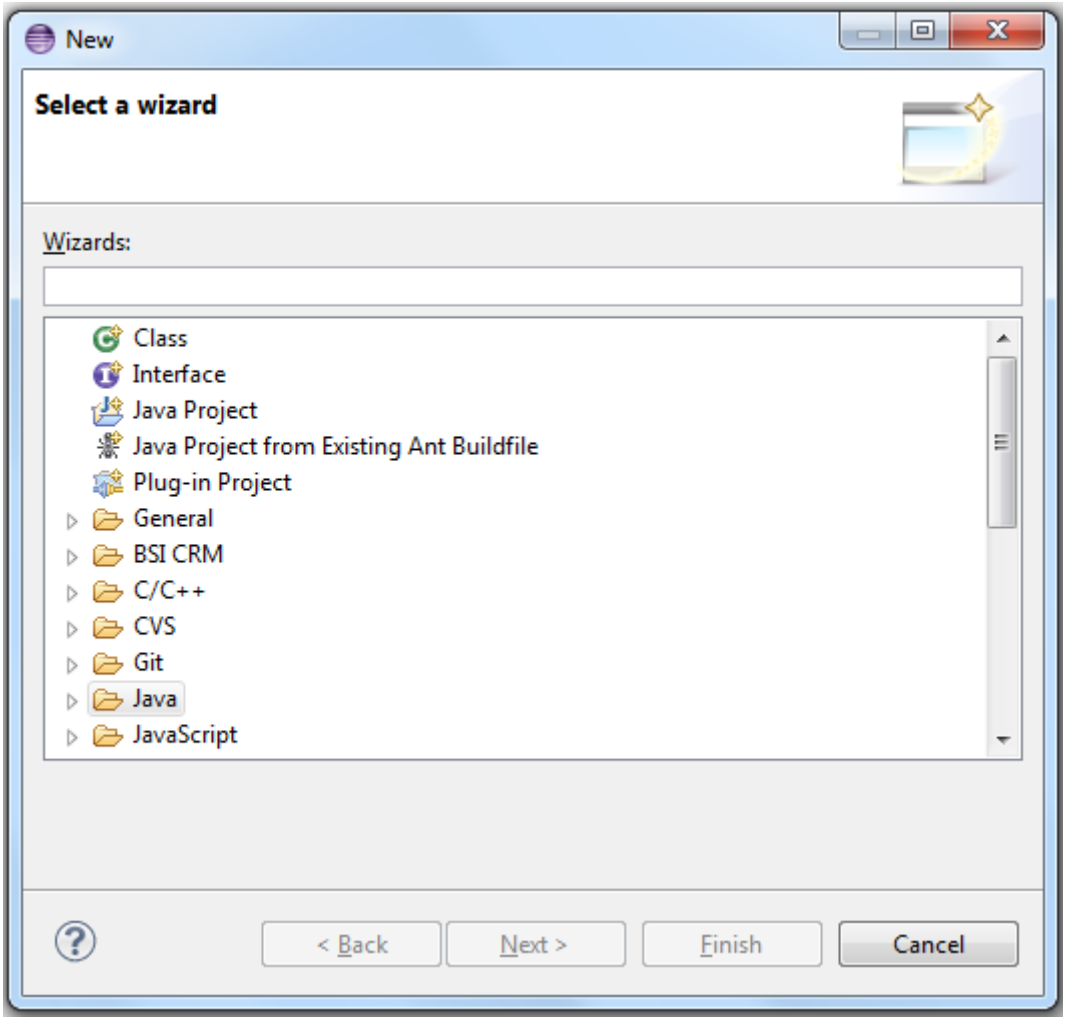
The application code created by the Scout SDK wizards helps to ensure that the resulting Scout application has a consistent and robust code base and is well aligned with the application model defined by the Scout runtime framework.

3.2. Eclipse IDE tooling

The Scout tooling is an extension of the Eclipse IDE. The goal of this section is not to provide a complete overview on the features contained in the Eclipse IDE. It provides a short overview of the important eclipse features, frequently used during the development of a Scout Application. Experimented Eclipse IDE users might skip this section.

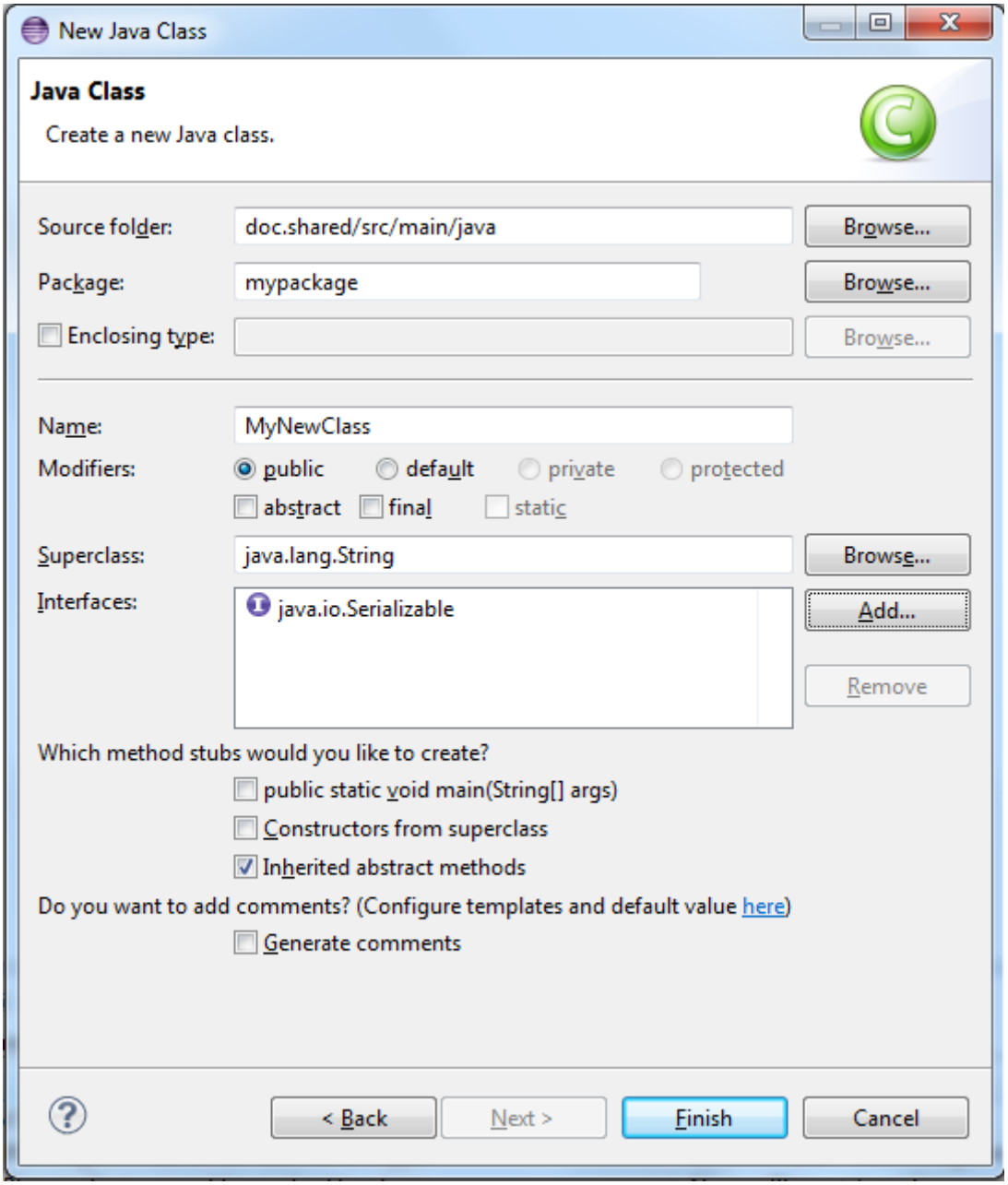
3.2.1. Start the New Wizard

To start the **New Wizard** wizard press `Ctrl`+`N` or use menu **File | New | Other...**. In the first wizard step type the name of the object you want to create into the **Wizards** field as shown in New Wizard.



3.2.2. Create a new Java class

Start the **New Wizard** and type **Class** in the Wizards field. Select **Class** Click on **[Next]** to open the **New class wizard**



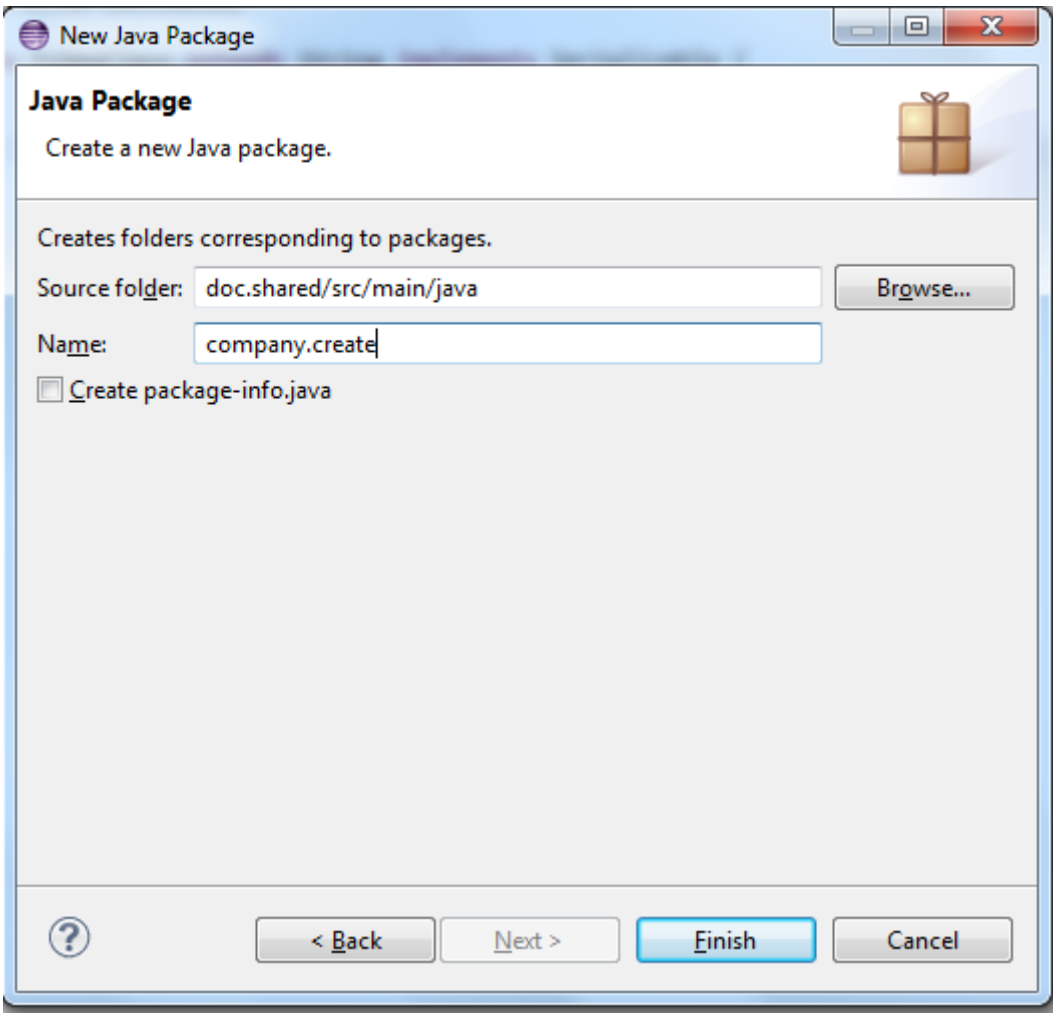
You can choose define the following properties:

- **Source folder**: Click on **[Browse]** to choose the project where the class belongs.
- **Package**: Click on **[Browse]** to choose the package in the given project. If the package does not exist it will be created a new one automatically.
- **Name**: Type the class name
- **Modifiers**: Choose public or default. Or abstract and/or final.
- **Superclass**: Choose the parent class clicking on **[Browse]**
- **Interfaces**: Click on **[Add]** to add the list of interfaces your class implements.
- **Method stubs**: Include methods in your class.
- **Comments**: Generate predefine comments.

Click on **[Finish]** when you are done with the class definition. The java editor will open and you can start editing.

3.2.3. Create a new Java package

Start the **New Wizard** and type **Package** in the Wizards field. Select **Package** Click on **[Next]** to open the **New package wizard**



In the **New package wizard** you can define the following properties:

- **Source folder**: Click on **[Browse]** to choose the project where the package belongs.
- **Name**: Write the name of the package.
- **Package info**: Choose the checkbox if you want package-info

Click on **[Next]** to create the project. The **Project Browser**

3.3. Scout SDK Overview

The Scout SDK tooling helps the Scout developer to quickly create frequently used Scout components. This Scout Tooling is implemented as extensions of the Eclipse IDE in the form of wizards, content assist extension to the Eclipse Java editor and support for dealing with translated texts called NLS support.

Scout Wizards

The Scout SDK tooling includes a number of wizards for the creation of frequently used Scout components. In many cases the execution of such wizards involves the creation/editing of

several source files. In the case of the creation of a new Scout form this includes the form class in the client module of the Scout application, a form data class and a service that communicates Descriptions for the individual wizards are provided in section Scout Wizards.

Content Assist

In the Scout framework the hierarchical organization of Scout components if frequently reflected in the form of inner classes. This allows the Scout tooling to provide context specific proposals in the form of content assist proposals offered in the Java editor of the Eclipse IDE. Examples for this form of the tooling includes the creation of form fields or adding columns and context menus to tables. Content assist support is described in detail in Scout Content Assistance.

NLS Tooling

Eclipse Scout comes with NLS (National Language Support). To support Scout developers in using Scout's NLS (National Language Support) the Scout SDK offers corresponding tooling to work with translated texts. This tooling is described in section Scout NLS Tooling.

3.4. Scout Wizards

The Scout SDK provides a set of wizards to create new Scout projects and various components for your Scout applications.

To start any of these wizards press `Ctrl+N` or use menu **File | New | Other...**. In the first wizard step type "*Scout*" into the **Wizards** field as shown in Scout SDK Wizards.

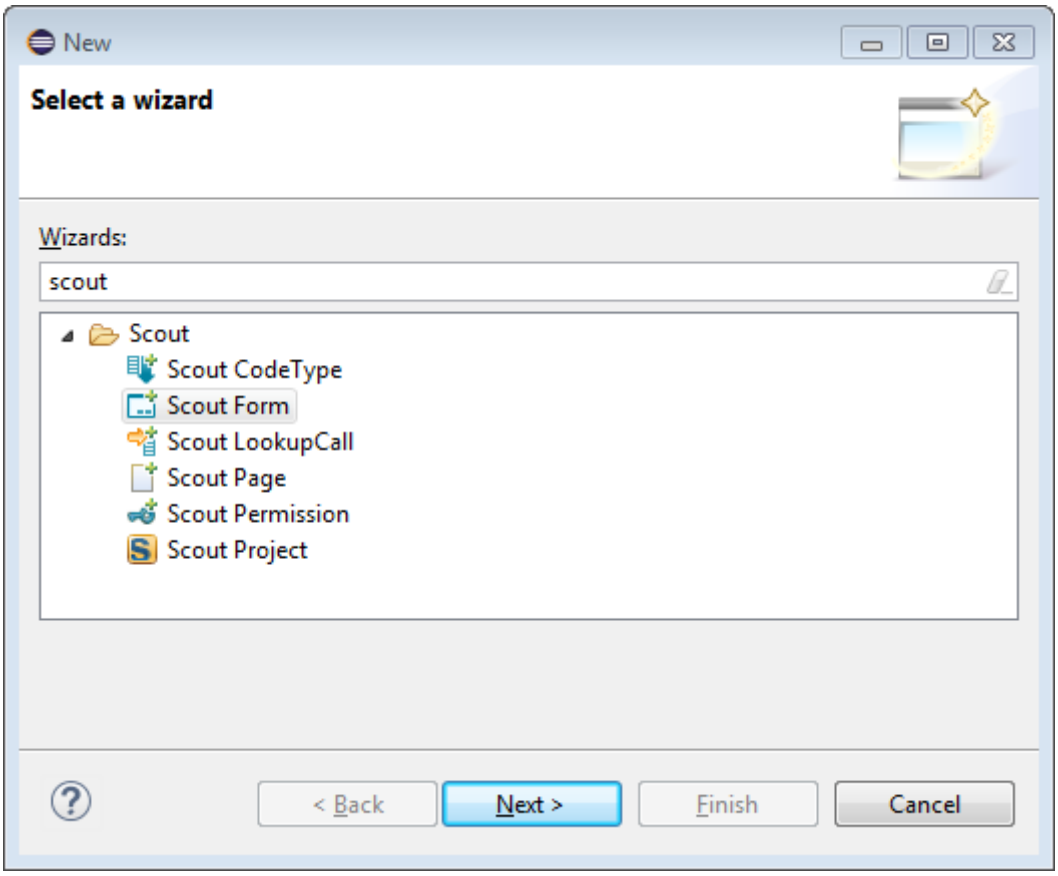


Figure 16. Selecting Scout Wizards in the Eclipse wizard dialog

The Wizards provided by the Scout SDK are introduced and described in the sections listed below.

- New Project Wizard
- New Page Wizard
- New Form Wizard

3.4.1. New Project Wizard

The *New Scout Project* wizard can be used to create a new Scout project from scratch.

To open the wizard press `Ctrl`+`N` or use **File | New | Other...** and type "*Scout*" into the **Wizards** search field. Then, select the entry "*Scout Project*" and click on **[Next]**. This leads to the initial dialog of the *New Scout Project* wizard as shown in The new Project Wizard.

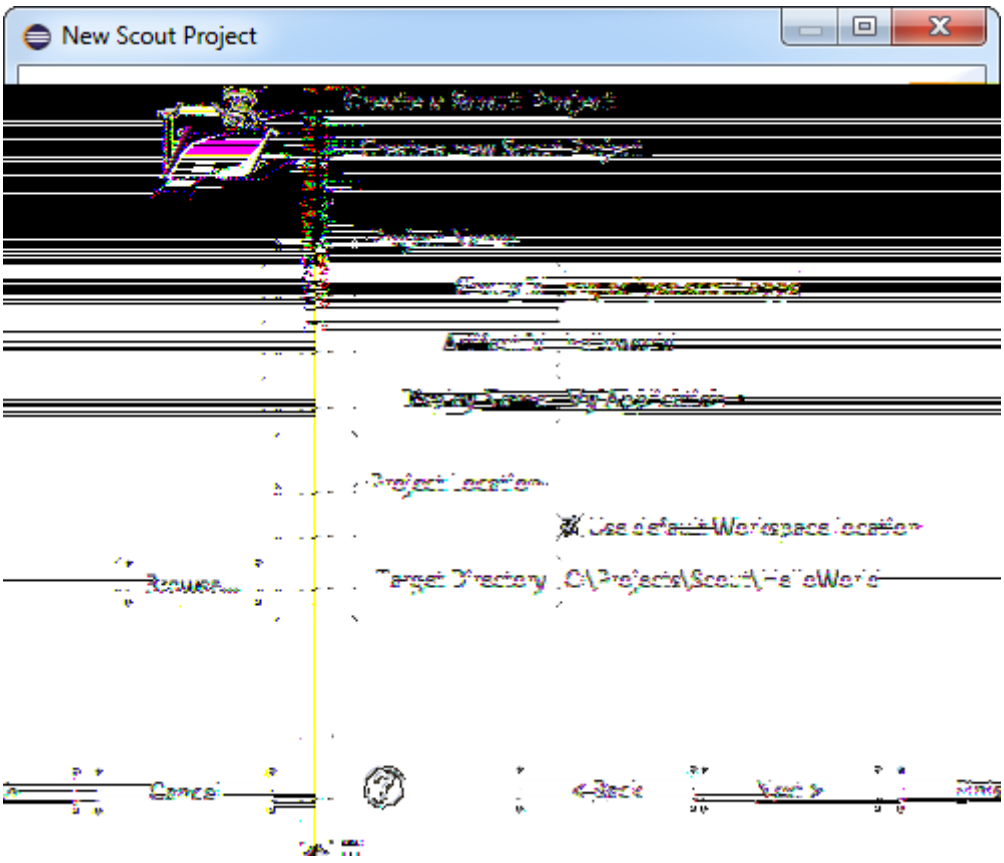


Figure 17. The new Project Wizard

A detailed description of the indiviudal wizard fields of The new Project Wizard is provided in the next section.

By clicking on the **[Finish]** button the wizard is started and a new Scout client server application is created in the form of a Maven multi-module project.

Wizard Fields and Default Values

All fields of the The new Project Wizard are initially filled with default values.

Group Id

Maven groupId used for all created projects. The default value is `org.eclipse.scout.apps`.

Artifact Id

Maven artifactId for the parent project. The additional projects are derived from this name. The default value is **helloworld**.

Display Name

The name of the application presented to the user. This name is shown in the Browser title bar. The default value is *"My Application"*

With the *Project Location* group box, you can control where the project will be created. Unchecked the *Use default Workspace location* checkbox to enter an other value in the *Target Directory* Field. The **[Browse...]** button can help you to find the appropriate path.

Created Components

With the The new Project Wizard wizard a complete Maven multi-module project is created. Using the default artifact Id **helloworld** the following Maven modules are created.

- Maven module **helloworld**
  - Contains the project's parent `pom.xml` file
- Maven module **helloworld.client**
  - Contains model components of the client application in `src/main/java` and model tests in `src/test/java`.
  - The class `HelloWorldForm` in package `org.eclipse.scout.apps.helloworld.client.helloworld` is an example of a model class.
- Maven module **helloworld.shared**
  - Contains components needed in both the client and the server application.
  - For examples see the `IHelloWorldFormService` interface in `src/main/java` and class `HelloWorldFormData` in `src/generated/java`.
  - The `Texts.nls` file that can be opened in the Scout NLS Editor.
- Maven module **helloworld.server**
  - Contains the model components of the server application in `src/main/java` and model tests in `src/test/java`.
  - The class `HelloWorldFormService` in package `org.eclipse.scout.apps.helloworld.server.helloworld` is an example of such a model class.
- Maven module **helloworld.server.app.dev**
  - Contains all components to run the Scout server application from within the Eclipse IDE.
  - The file `config.properties` in folder `src/main/resources` contains the development configuration for the Scout server application.
  - The file `pom.xml` bundles the Jetty web server with the server application.

- The file `server-dev.launch` contains the launch configuration for the Eclipse IDE.

- Maven module **helloworld.server.app.war**
  - Contains all components to create a Scout server WAR file to deploy to an external web server.
  - The file `config.properties` in folder `src/main/resources` contains the server configuration.
  - The file `pom.xml` is used to build the Scout server WAR file.

- Maven module **helloworld.ui.html**
  - Contains servlet filters and the HTML pages as well as custom CSS and JavaScript files for the Scout UI Server.
  - See class `UiServletFilter` in `src/main/java` and folder `WebContent` in `source/main/resources`.

- Maven module **helloworld.ui.html.app.dev**
  - Contains all components to run the Scout UI application from within the Eclipse IDE.
  - The file `config.properties` in folder `src/main/resources` contains the development configuration for the application.
  - The file `web.xml` in folder `src/main/webapp` contains the web configuration for the application.
  - The file `pom.xml` bundles the Jetty web server with the application.
  - The file `ui-html-dev.launch` contains the launch configuration for the Eclipse IDE.

- Maven module **helloworld.ui.html.app.war**
  - Contains all components to create a Scout UI WAR file to deploy to an external web server.
  - The file `config.properties` in folder `src/main/resources` contains the application configuration.
  - The file `web.xml` in folder `src/main/webapp` contains the web configuration.
  - The file `pom.xml` is used to build the Scout UI WAR file.

3.4.2. New Page Wizard

The *New Scout Page* wizard can be used to create a new page and related classes. To start the wizard use **File | New | Other...** or press `Ctrl`+`N`.

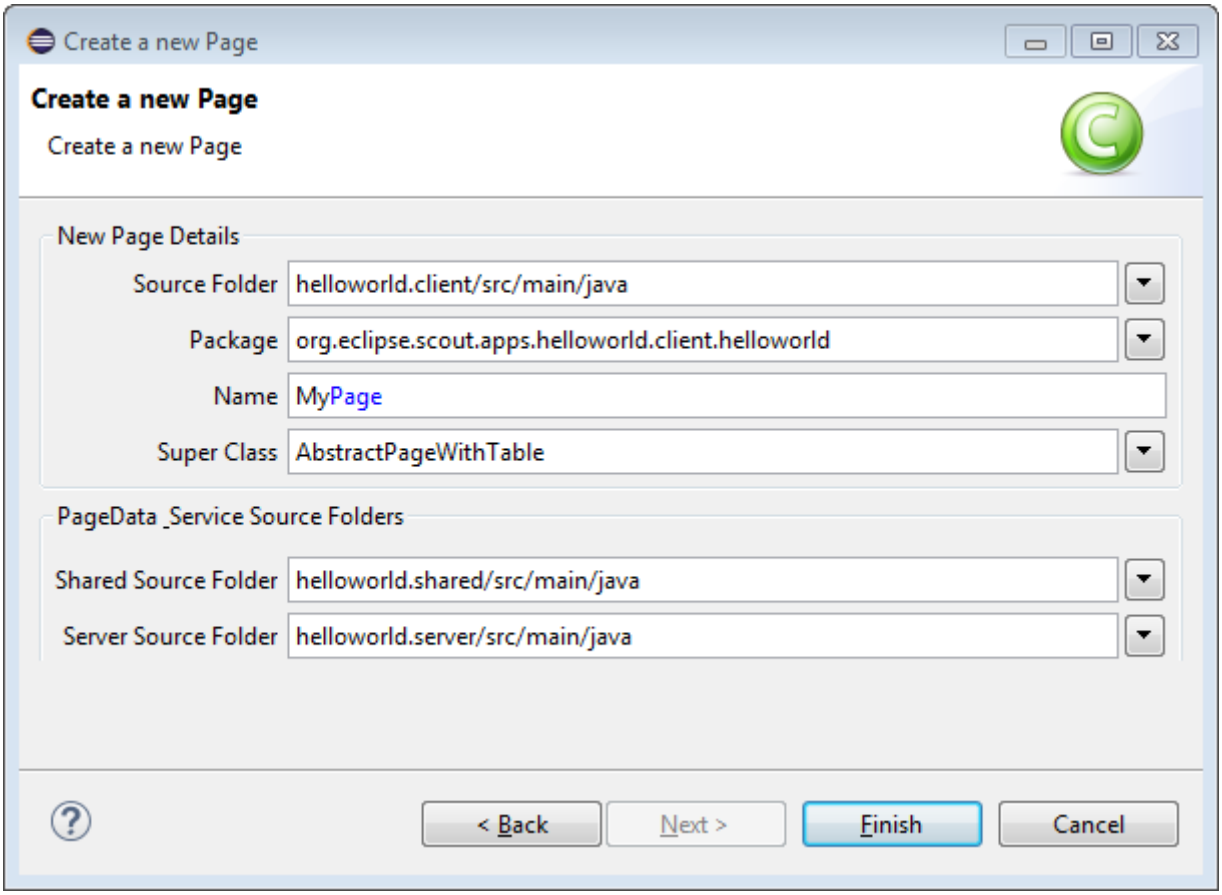


Figure 18. The new Page Wizard

In the case of The new Page Wizard the package `org.eclipse.scout.apps.helloworld.client.helloworld` has been selected in the Package Explorer. The only wizard field that then needs to be filled in manually is the **Name** field.

By clicking on the **[Finish]** button the wizard is started and the specified components are created.

Wizard Fields and Default Values

Most of the fields of the The new Page Wizard will be filled with default values depending on the current context of the IDE. The context can be derived from a package selected in the Package Explorer or from the class in the active Java Editor.

Source Folder

The source folder of the Maven client module used for the creation of the page. The default value is the `src/main/java` folder in the Maven client module.

Package

The Java package that will contain the page class. The Scout SDK will try to guess the package name from the current context and derive matching package names for the Maven shared module.

Name

The name of the page class. According to Scout conventions the class name ends with the suffix `Page`.

Super Class

The super class for the form. `AbstractPageWithTable` is the default value.

Shared Source Folder

The source folder of the Maven shared module used for creation of the page data and the service interface. The default value is the `src/main/java` folder in the Maven shared module.

Server Source Folder

The source folder of the Maven server module used for creation of the service implementation. The default value is the `src/main/java` folder in the Maven server module.

Created Components

In the The new Page Wizard example shown above the Scout SDK will create the following components.

- In Maven module **helloworld.client**
  - The `MyPage` page class in folder `src/main/java` and package `org.eclipse.scout.apps.helloworld.client.helloworld`
- In Maven module **helloworld.shared**
  - The `IMyService` service interface in folder `src/main/java` and package `org.eclipse.scout.apps.helloworld.shared.helloworld`
  - `MyPageData` page data class in folder `src/generated/java` and package `org.eclipse.scout.apps.helloworld.shared.helloworld`
- In Maven module **helloworld.server**
  - The `MyService` implementation in folder `src/main/java` and package `org.eclipse.scout.apps.helloworld.server.helloworld`

3.4.3. New Form Wizard

The *New Form* wizard is be used to create a new form including a form data, permissions and and related service. To start the wizard use **File | New | Other...** or press `Ctrl+N`.

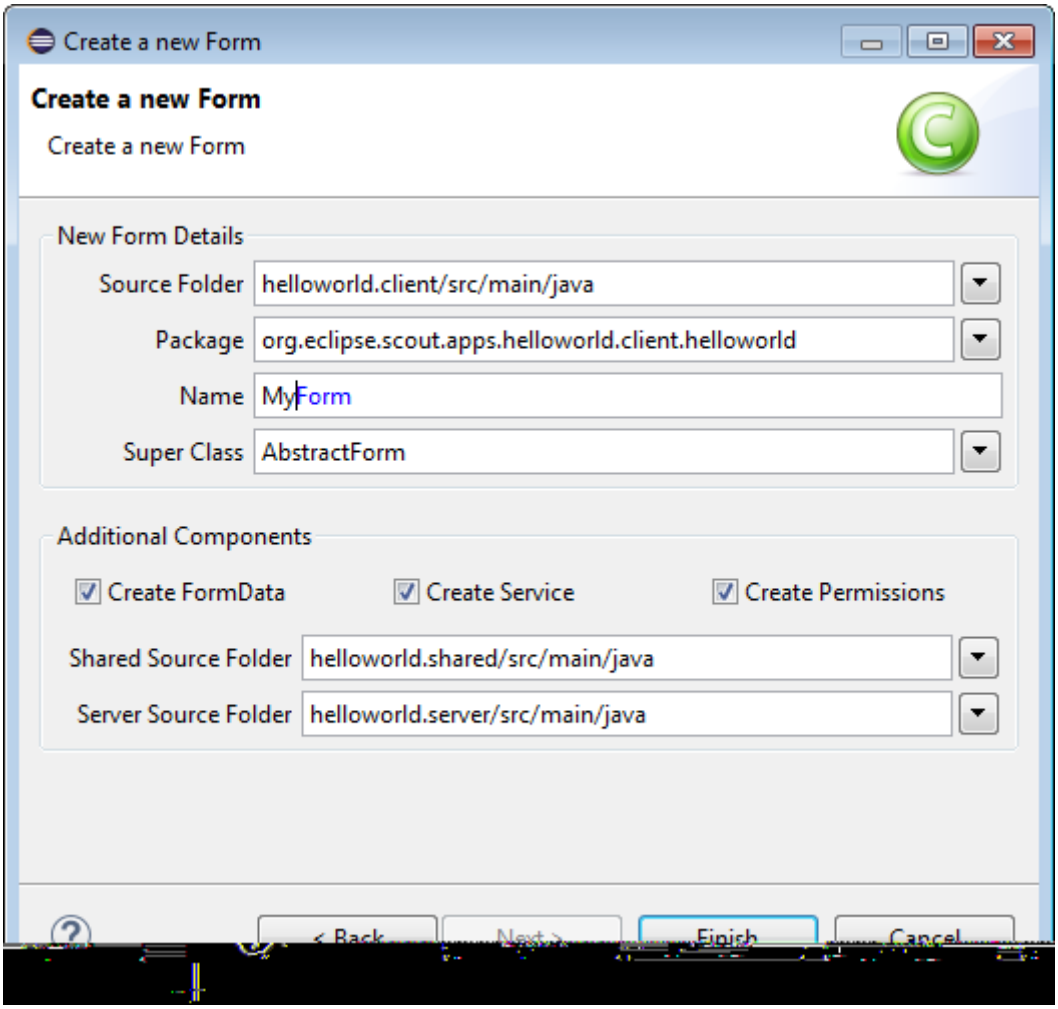


Figure 19. The new Form Wizard

In the case of The new Form Wizard the package `org.eclipse.scout.apps.helloworld.client.helloworld` has been selected in the Package Explorer. The only wizard field that then needs to be filled in manually is the **Name** field.

By clicking on the **[Finish]** button the wizard is started and the specified components are created.

Wizard Fields and Default Values

Most of the fields of the The new Form Wizard will be filled with default values depending on the current context of the IDE. The context can be derived from a package selected in the Package Explorer or from the class in the active Java Editor.

Source Folder

The source folder of the Maven client module used for the creation of the form class. The default value is the `src/main/java` folder in the Maven client module.

Package

The Java package that will contain the form class. The Scout SDK will try to guess the package name from the current context and derive matching package names for the Maven shared and server modules.

Name

The name of the form class. According to Scout conventions the class name ends with the suffix

`Form`.

Super Class

The super class for the form. `AbstractForm` is the default value.

Create FormData

If ticked, a form data class will be created in the shared module.

Create Service

If ticked, a service interface is created in the shared module and a service implementation is created in the Maven server module.

Create Permissions

If ticked, read and update permissions are created in the Maven shared module.

Shared Source Folder

The source folder of the Maven shared module used for creation of the form data, the service interface and the permission classes. The default value is the `src/main/java` folder in the Maven shared module.

Server Source Folder

The source folder of the Maven server module used for the service class creation. The default value is the `src/main/java` folder in the Maven server module.

Created Components

In the The new Form Wizard example shown above the Scout SDK will create the following components.

- In Maven module **helloworld.client**
  - The `MyForm` form class in folder `src/main/java` and package `org.eclipse.scout.apps.helloworld.client.helloworld`
- In Maven module **helloworld.shared**
  - In folder `src/main/java` and package `org.eclipse.scout.apps.helloworld.shared.helloworld`
    - The `IMyService` service interface
    - The `ReadMyPermission` permission class
    - The `UpdateMyPermission` permission class
  - The `MyFormData` form data class in folder `src/generated/java` and package `org.eclipse.scout.apps.helloworld.shared.helloworld`
- In Maven module **helloworld.server**
  - The `MyService` service class in folder `src/main/java` and package `org.eclipse.scout.apps.helloworld.server.helloworld`

3.5. Scout Content Assistance

To create new Scout components that are represented by inner classes in the Scout framework, the Scout tooling extends the Java content assist of the Eclipse Java editor. The offered proposals are context specific. Depending on the current cursor position in the Java editor, possible Scout components are added to the proposal list.

In a class representing a group box in a form, the Scout content assist adds proposals for various form fields are shown and in a table class the content assist adds proposals to add table columns or context menus. are added that trigger the creation of inner classes for form fields, table columns or codes. The Eclipse content assist can be started by typing `Ctrl+Space`.

3.5.1. Create new Form Fields

To add additional form fields to a form the current edit position needs to be inside of a Scout group box. Typing `Ctrl+Space` then provides access to the most frequently used Scout widgets as shown in Figure Proposals in a GroupBox.

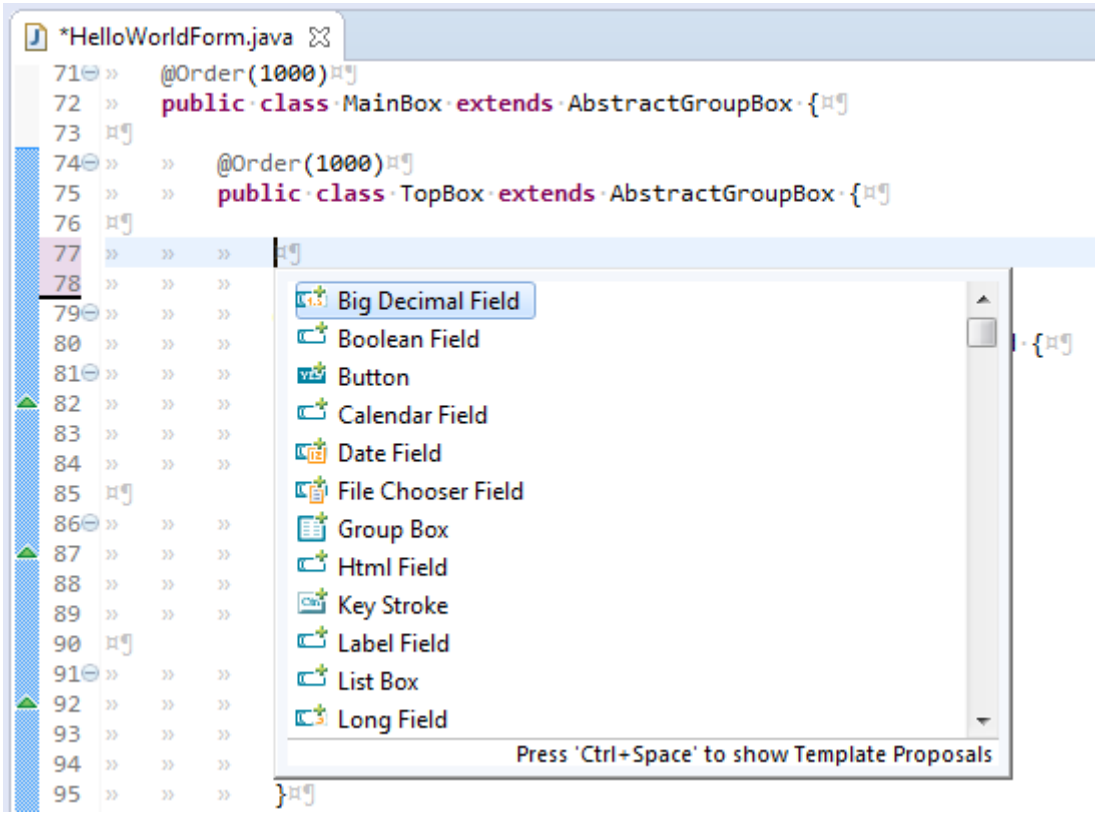


Figure 20. Proposals to create new form fields in a GroupBox

When a template is selected, it is possible to customized it by navigating between the different Edit-Groups with the `Tab` Key (this works exactly like other templates in the Eclipse Editor). With this mechanism you can quickly define the class name, the parent class and other properties. To exit the Edit-Mode just press `Enter`.

3.5.2. Create new Table Columns

For adding new columns in a table the set the current edit position inside a Scout table. The Scout table itself may be located inside of a TableField as shown in Figure Proposals in a Table or can also be located inside of a Scout TablePage.

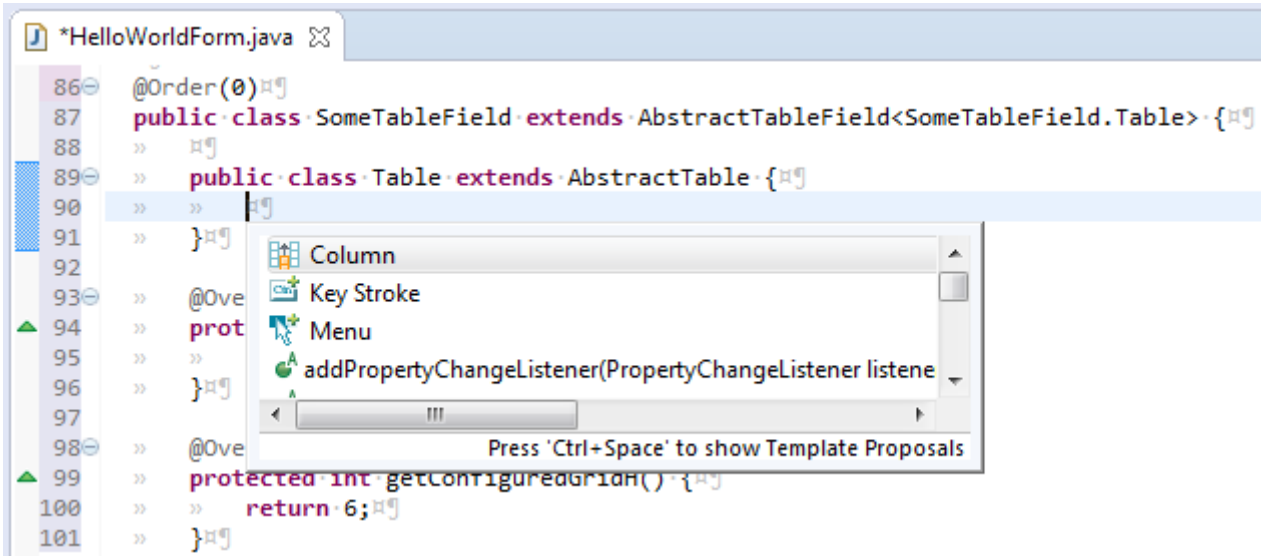


Figure 21. Proposals to create new columns in a Table

Next to adding columns the content assist shown in Figure Proposals in a Table can also be used to add key stroke actions and menus to tables.

3.5.3. Create new Codes

Adding new Codes to an existing CodeType is supported by the content assist as shown in Figure Proposals in a CodeType.

Figure 22. Proposals to create new codes in a CodeType

3.6. Scout NLS Tooling

bla

3.6.1. Adding a new Translated Text Entry

Translated text entries are most frequently added when working in the Java editor view.

When the the current edit position is inside the String parameter of the `TEXTS.get()` code, the content assist (opened with `Ctrl` + `Space`) provides support for the NLS entries as shown in Figure Proposals in a TEXTS.get().

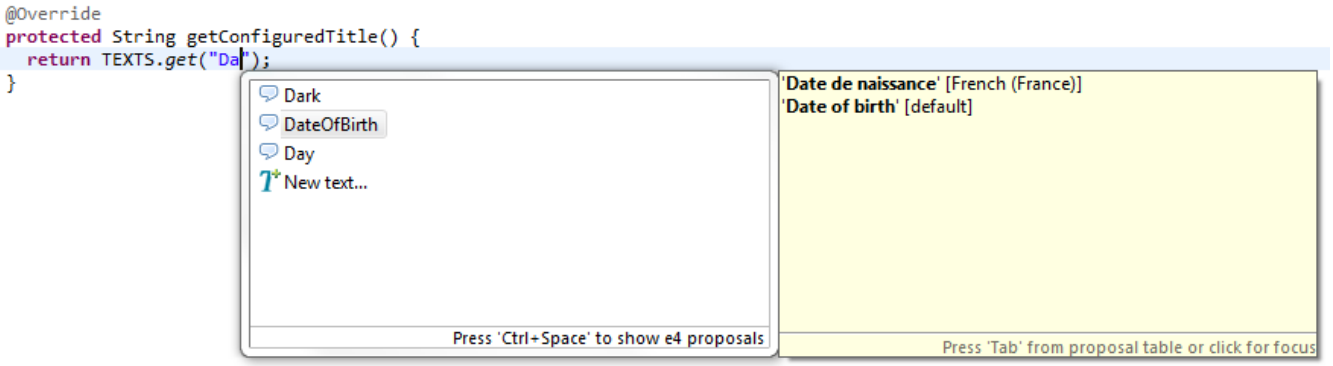
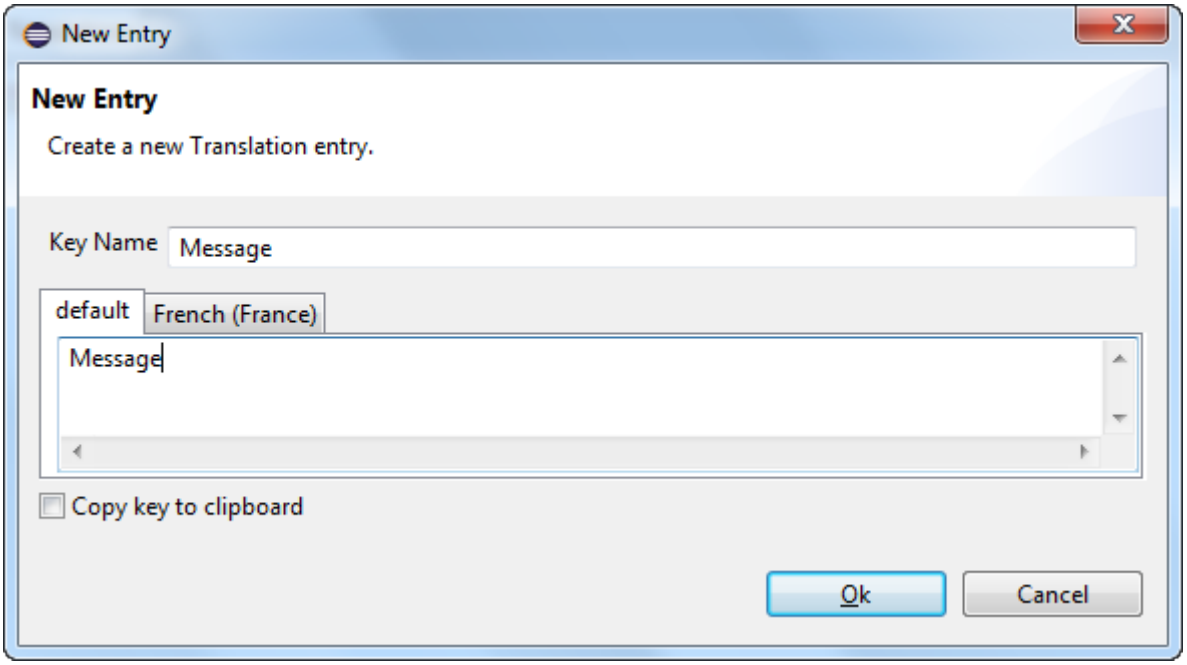


Figure 23. Proposals corresponding to NLS Support.

Selecting one of proposal entries (like "DateOfBirth" in the example) shows the available translations on the right hand side. To select a specific proposal entry you may double click on the entry or hit the `Enter` key. To create a new text entry select **New text...** at the end of the proposal list.

Adding a translated text can then be done in the *New Entry* wizard provided by the Scout SDK as shown in new text entry wizard.



Adding a new text with the New Entry wizard.

Key Name

This field holds the text key that is used to access translated text.

default

This field holds the default translated text for the key. Make sure to at least provide a translated text in this tab.

French (France)

Additional tabs to enter translations for other languages may be present. Adding additional languages is described in the text for the NLS editor.

Copy key to the clipboard

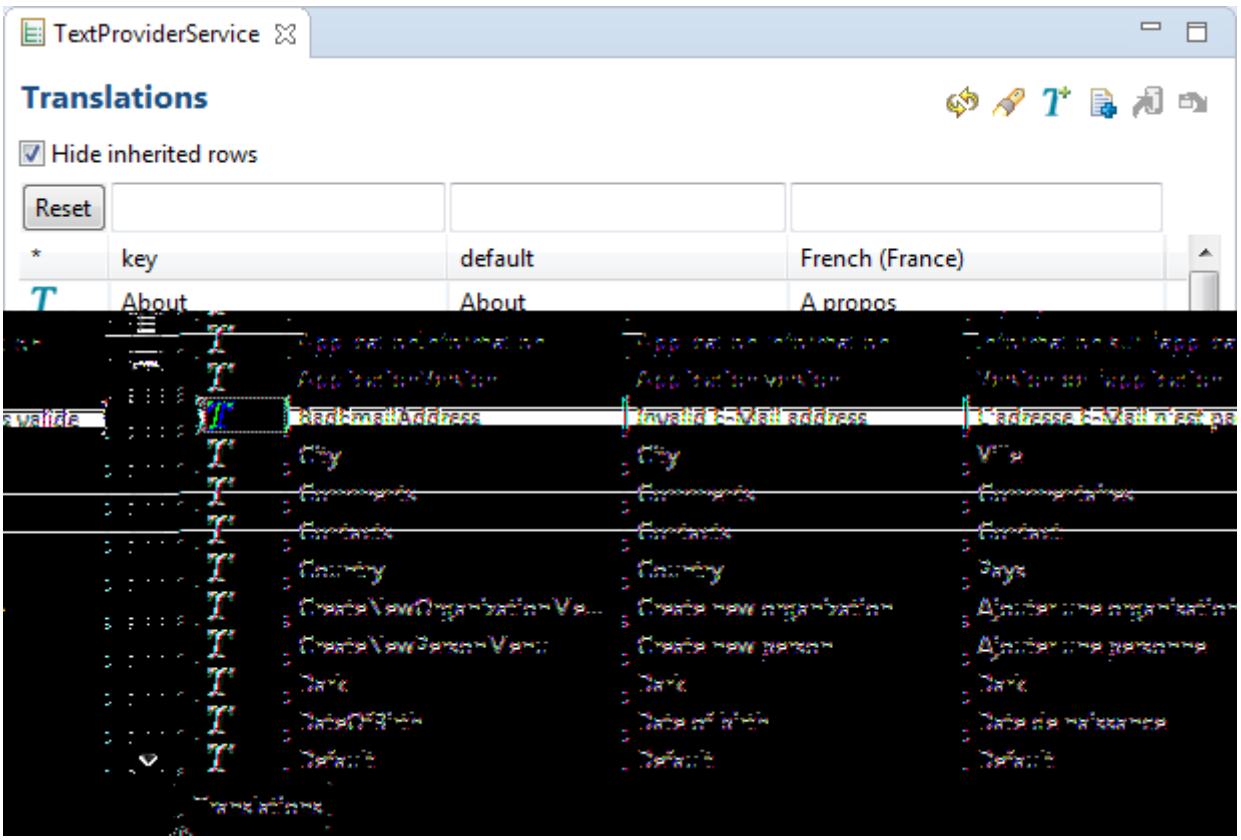
Select this checkbox to copy the key name to the clipboard and paste it later in your code.

3.6.2. The NLS Editor

To manage translated application texts for different languages the Scout SDK includes a NLS editor. This editor helps to efficiently deal to edit all the property files that are used with the default setup of Scout.

The NLS editor can be accessed for each text provider service of a Scout application via the `*.nls` files of the shared Maven modules of the application. In the case of the "Hello World" application you will find the `Texts.nls` file in module `org.eclipse.scout.helloworld.shared`. To open the editor for the "Hello World" application select the `Texts.nls` file first and then use context menu **Open With | NLS Editor**.

The screenshot below shows the opened NLS editor. In the first column the `key` values are shown that are used in accessing translations through `TEXTS.get("key")`. The second columns holds the default translations followed by columns holding the translations for other translated languages.



3.6.3. Action Buttons

Actions on the top right corner:

[icon refresh]	Refresh NLS Project	Reload the content of the editor.
[icon find obj]	Show NLS entry usage	Search in the Java code where the NLS Key is used.
	New entryÉ	Opens the New Text Entry Wizard
[icon fileadd pending]	New languageÉ	Opens the Add a Language Wizard
[icon import]	ImportÉ	Import the NLS entries of an external file
[icon export]	ExportÉ	Export the NLS entries to an external file

Import and Export requires additional components.

Hide inherited rows checkbox

On the top of each columns, the text fields allow you to filter the entries in the table. With the **[Reset]** button on the right you will empty those filters.

The entries in the table can be directly edited by pressing F2 or double-clicking into a text cell.

On each row it is possible to call following context menu:

	Modify Entry	Opens the NLS Entry Wizard
[icon find obj]	Find references to 'Xxx'	Search in the Java code where the NLS Key is used.
	Remove Xxx	Delete the NLS Entry from the files

3.6.4. Default Mapping to Property Files

The mapping between the properties files is registered in the "Text Provider Service" class. Per default the files follow this pattern: `<your application>.shared/src/main/resources/<identifier of the project>/texts/Texts<language>.properties`

where:

- `<identifier of the project>` is a chain of folders following the same convention as the Java source files with the package name. For example the `org.eclipse.contacts.shared` project uses `org/eclipse/scout/contacts/shared` as path.
- `<language>` is an identifier of the language and the country. Some possible file names:
  - `Texts.properties` is the default language

- `Texts_de.properties` is for german
- `Texts_fr_BE` will be for french in Belgium

Chapter 4. A One Day Tutorial

In this chapter we will create the ÒContactsÓ Scout application. The goal of this tutorial application is to learn about the most prominent features of the Eclipse Scout framework using a fully functioning application.

The application is kept small enough to complete this tutorial within less than a day. An extended version of ÒContactsÓ is available as a Scout sample application on Github.

As a prerequisite to this tutorial we assume that the reader has successfully completed the chapters "Hello World Tutorial" and "Import the Scout Demo Applications" as described in the Eclipse Scout user guide. To access the Scout user guide help hit F1 in the Eclipse IDE. This opens the Eclipse help view that includes the Eclipse Scout User Guide as shown in Figure 000.

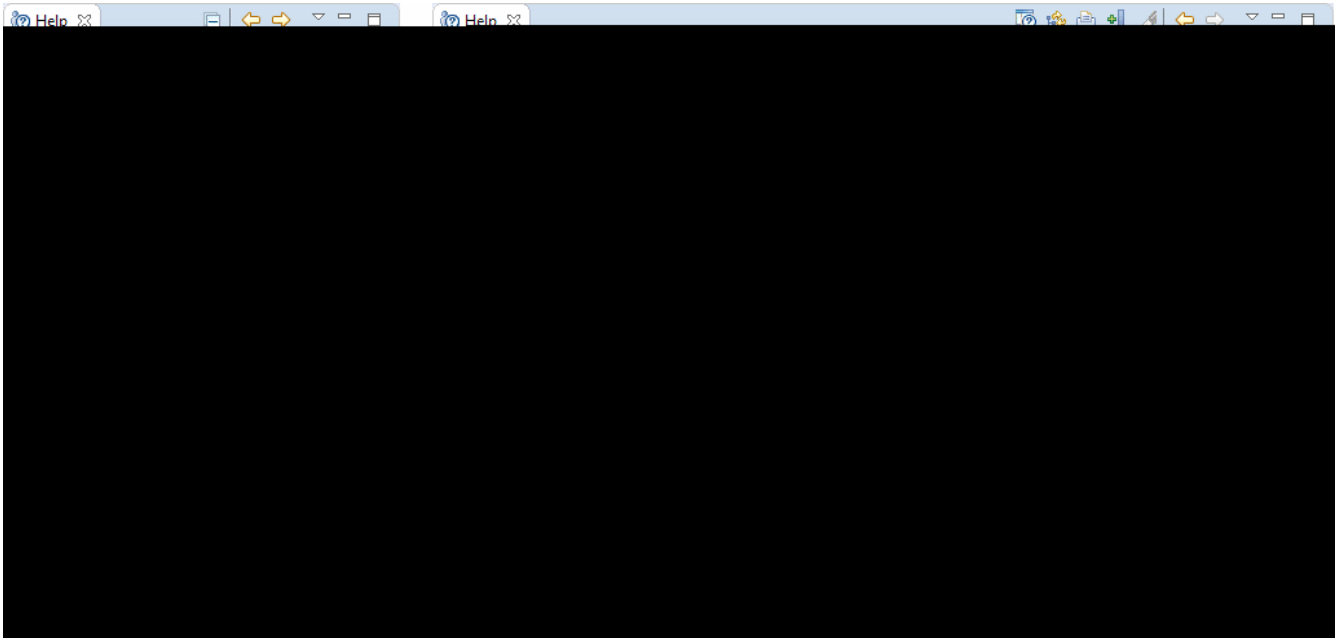


Figure 24. The Eclipse help view including the Eclipse Scout User Guide.

The ÒContactsÓ tutorial is organized as follows. In the first section, the finished ÒContactsÓ application is explained from the user perspective. The remaining sections focus on the individual steps to implement the ÒContactsÓ tutorial application.

4.1. The “Contacts” Application

The ÒContactsÓ 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 ÒContactsÓ demo application as described in the Eclipse Scout User Guide 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. Alternatively, you can also view the source code of the ÒContactsÓ demo application on Github. [16: ÒContactsÓ on Github: https://github.com/BSI-Business-Systems-Integration-AG/org.eclipse.scout.docs/tree/releases/6.0.x/code/contacts].

Figure Figure 000 below shows the ÒContactsÓ application after connecting to the Scout UI application.

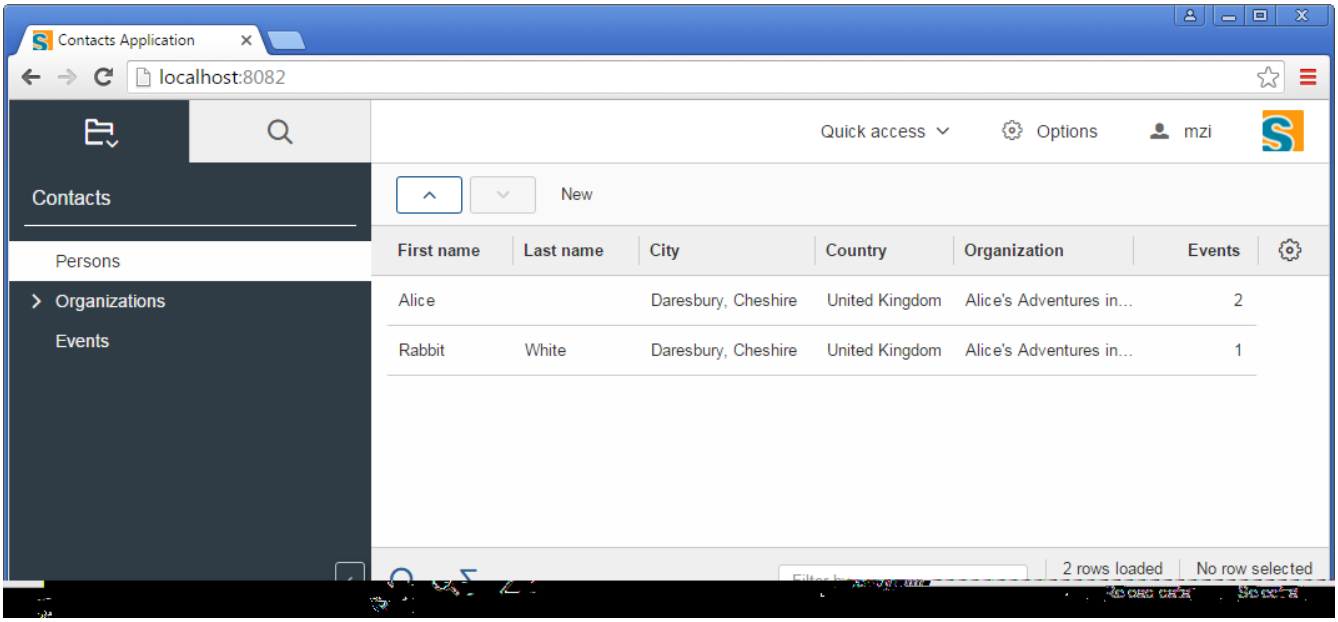


Figure 25. The “Contacts” application with the person page.

The ÒContactsÓ application also 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 000 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 to button with the folder icon is 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 hand 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 000. Selecting a specific person the 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 000.

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 in Figure 000.

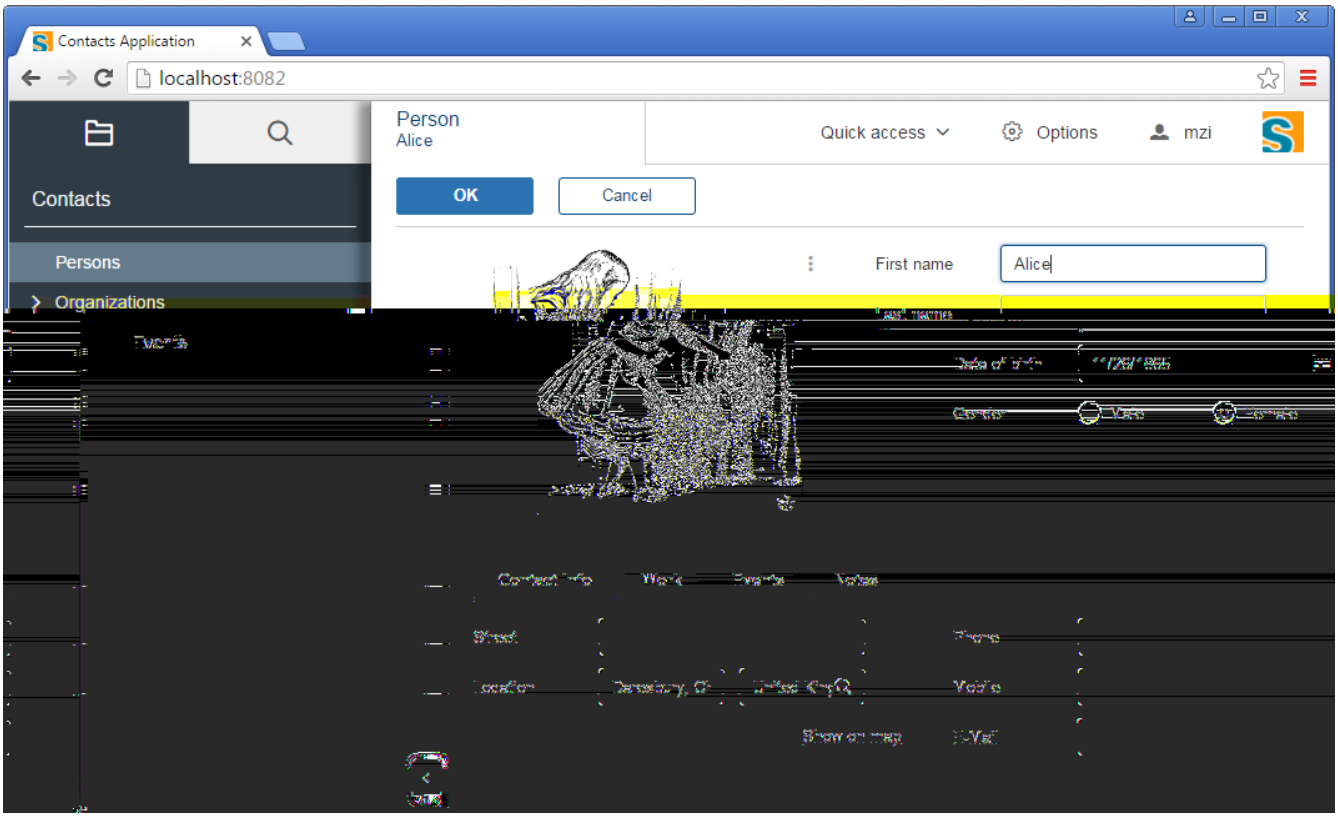


Figure 26. The “Contacts” application with a person opened in a form.

4.2. 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 appliation the tutorial shows how to implement services, logging, databases access, and serveral other aspects of Scout applications.

The tutorial is organized into the sequence of the consecutive steps listed below. Each of the 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

We start with an empty workspace and the "Hello World" project. At the end of step one we have a project setup that matches the "Contacts" application. This includes the organization and naming of Maven modules and the individual Java packages.

Step 2: 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 000.

Step 3: Creating and Accessing the Database

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

Step 4: Adding Forms to Create/Edit Persons and Organizations

Having access to the database is used in this step to add the components that allow a user to create and edit persons and organizations in the user interface of the "Contacts" application.

Step 5: Linking Oranizations 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 hierachical 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 6: Next Steps and Outlook

This step discusses the gap between the tutorial application and the complete "Contacts" demo application and suggests possible next steps to close the gap.

4.3. Setting up the Initial Project

This section deals with setting up the initial workspace and codebase for the "Contacts" application. The goal of this step lies in the project setup that closely matches the "Contacts" application. This includes the organization and naming of Maven modules and the individual Java packages.

The creation up of the initial project setup described in this section consists of the tasks listed below.

- Creating the initial Codebase
- Removing unnecessery Components
- Changes to Class WorkOutline
- Changes to Class Desktop

This first step of the "Contacts" tutorial ends with a review of the results of this first tutorial step in section What have we achieved?.

4.3.1. Creating the initial Codebase

The initial code for the ÒContactsÓ application is generated using the *New Scout Project* wizard as described in Section [sec-wizard_new_project]. For the wizard fiels you may use the following values:

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

To create this initial application click on **[Finish]**. The project wizard then creates a number of

Maven modules as shown in Figure 000.

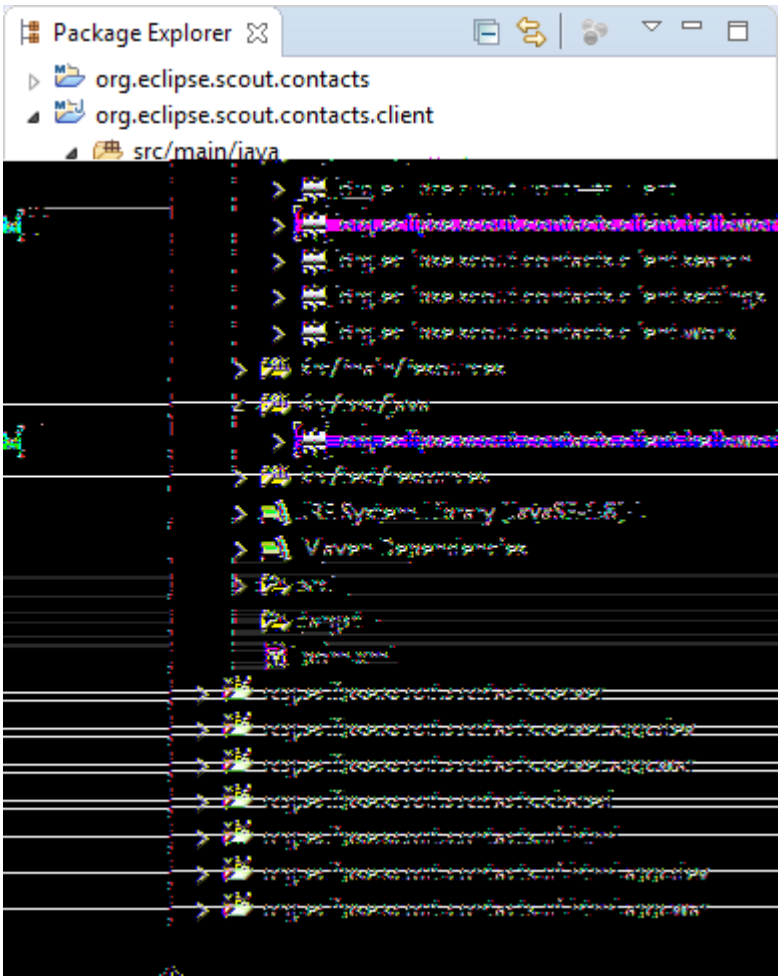


Figure 27. The package explorer with the initial Maven modules created for the "Contacts" application.

4.3.2. Removing unnecessery Components

We start with removing the `*.helloworld` and `\*.settings` packages in all Maven modules of the "Contacts" application. To delete a packages first select an individual package or packages in the Eclipse package explorer as shown in Figure 000 and then hit the `Delete` key. The explicit list of packages to delete per Maven module is provided in the text below.

Client Module `org.eclipse.scout.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 `org.eclipse.scout.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 `org.eclipse.scout.contacts.shared`

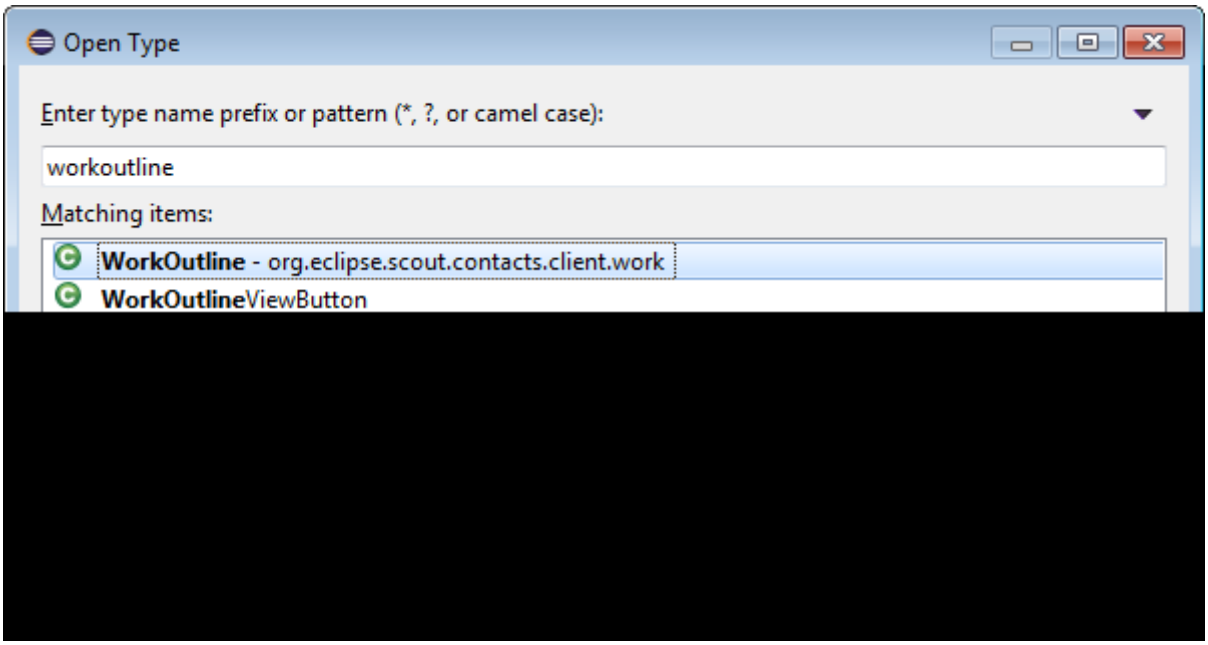
- In folder `src/main/java`
  - Delete package `org.eclipse.scout.contacts.shared.helloworld`
- In folder `src/generated/java`
  - Delete package `org.eclipse.scout.contacts.shared.helloworld`

4.3.3. Changes to Class WorkOutline

Instead of adding a new "Contacts" outline to the application re simply reuse and rename the "Work" outline that was created during the initial project setup. 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 "Contact"
- Change the outline icon to `Icons.Category`

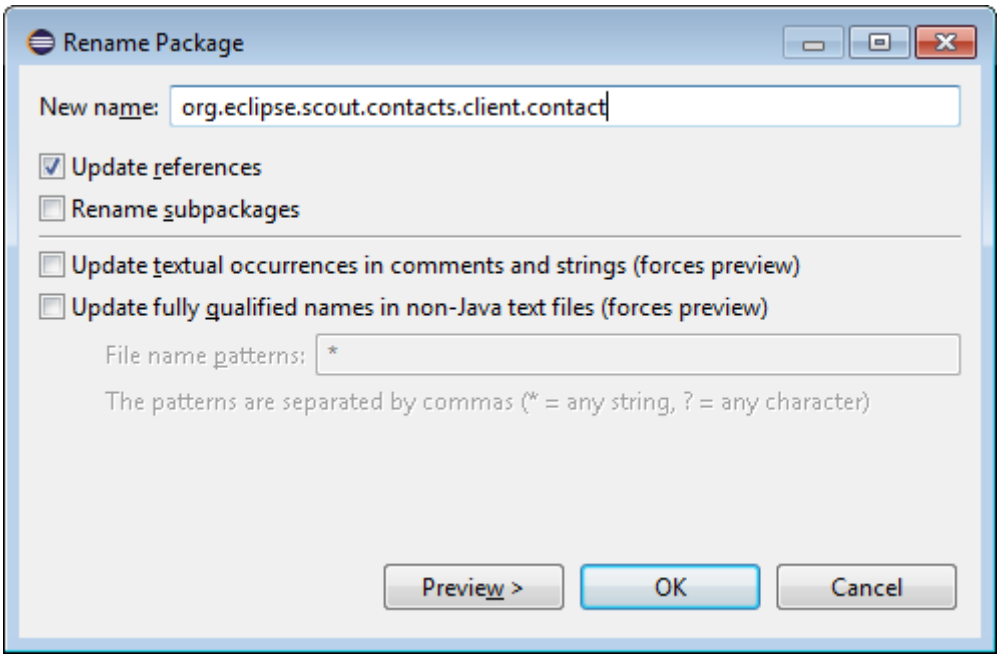
To quickly find the class we open the *Open Type* dialog from the Eclipse IDE [17: Type 'open type' into the *Quick Access* field of the Eclipse IDE or hit `Ctrl`,`Shift`, and `T` at the same time.] and enter 'workoutline' into the search field as shown in Figure 000. In the result list, we select the desired class and click the **[OK]** button to open the file `WorkOutline.java` in the Java editor of the Eclipse IDE.



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 package name and using the Eclipse

menu **Refactor | Rename...**. This opens the package rename dialog as shown in Figure 000.



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 then rename the class by clicking into the class identifier `WorkOutline` and use the Eclipse menu **Refactor | Rename...** again. 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.

Now, we change the outline title in method `getConfiguredTitle` to "Contacts" using the method described in Adding a new Translated Text Entry. Finally we change the return value of method `getConfiguredIconId` to value `Icons.Category`.

Listing 1. Initial implementation of class ContactOutline.

```
public class ContactOutline extends AbstractOutline {

  @Override
  protected void execCreateChildPages(List<IPage<?>> pageList) {
    // pages to be shown in the navigation area of this outline
  }

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

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

Before we conclude the modifications in class ContactOutline let us remove the remaining compile error in class `ContactOutline`. For this remove the line `pageList.add(new HelloWorldPage());` in method `execCreateChildPages`. Then, update the imports using Eclipse menu **Source | Organize Imports** and save the modified class.

4.3.4. Changes to Class Desktop

The second class to adapt for the "Contacts" application is class Desktop. 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 removing `SettingsOutline.class` from the list of return values in `getConfiguredOutlines` as shown in Listing getConfiguredOutlines.

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 inner class SettingOutlineViewButton
- Rename inner class WorkOutlineViewButton to ContactOutlineViewButton
- Change FileMenu to QuickAccesMenu and let `getConfiguredText` return "Quick Access"
- Change BookmarkMenu to OptionsMenu. Let `getConfiguredText` return "Options", and add method `getConfiguredIconId` to return `AbstractIcons.Gear`.
- Change HelpMenu to UserMenuLet. `getConfiguredText` return "", and add method `getConfiguredIconId` to return `AbstractIcons.Person`.

At the end of these changes the inner class structure of class Desktop will look like shown in Listing Desktop.

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

```
public class Desktop extends AbstractDesktop {

  // outline buttons of the application
  @Order(1)
  public class ContactOutlineViewButton extends AbstractOutlineViewButton {
  }

  @Order(2)
  public class SearchOutlineViewButton extends AbstractOutlineViewButton {
  }

  // top level menus for the header area of the application
  @Order(10)
  public class QuickAccessMenu extends AbstractMenu {
  }

  @Order(20)
  public class OptionsMenu extends AbstractFormMenu<OptionsForm> {
  }

  @Order(30)
  public class UserMenu extends AbstractFormMenu<UserForm> {
  }
}
```

The explicit listing for the changed class OptionsMenu is provided in Listing OptionsMenu. Please note that for now your class `OptionsMenu` will just extend `AbstractMenu`.

Listing 4. Initial implementation for the "Options" menu.

```
  @Order(20)
  public class OptionsMenu extends AbstractFormMenu<OptionsForm> {

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

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

  }
```

4.3.5. 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 000.

- Start the backend application
- Start the frontend application
- Open address in your http://localhost:8082/ browser

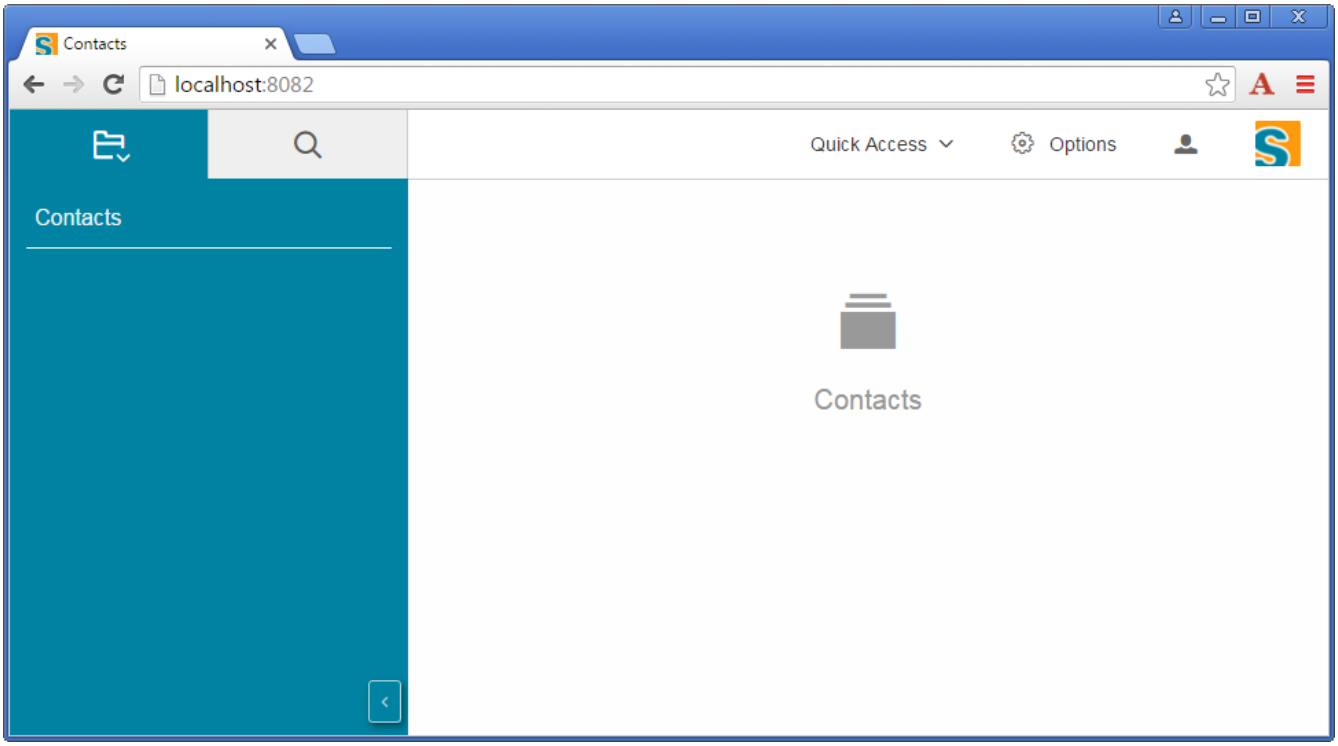


Figure 28. The "Contacts" application at the end of tutorial step 1.

From the coding perspective we now have all necessary maven Modules for "Contacts" application including Java package and class names to match with the complete Scout "Contacts" demo application. This fact is very important as 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.

4.4. 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 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 section Creating and Accessing the Database in the next tutorial step.

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

- Creating additional Packages
- Creating the Country Lookup Call
- Creating the Person Page
- Adding Table Columns to the Page
- 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 section What have we achieved?.

4.4.1. 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 `org.eclipse.scout.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 000
- Make sure the newly created person package is selected in the Eclipse Package Explorer

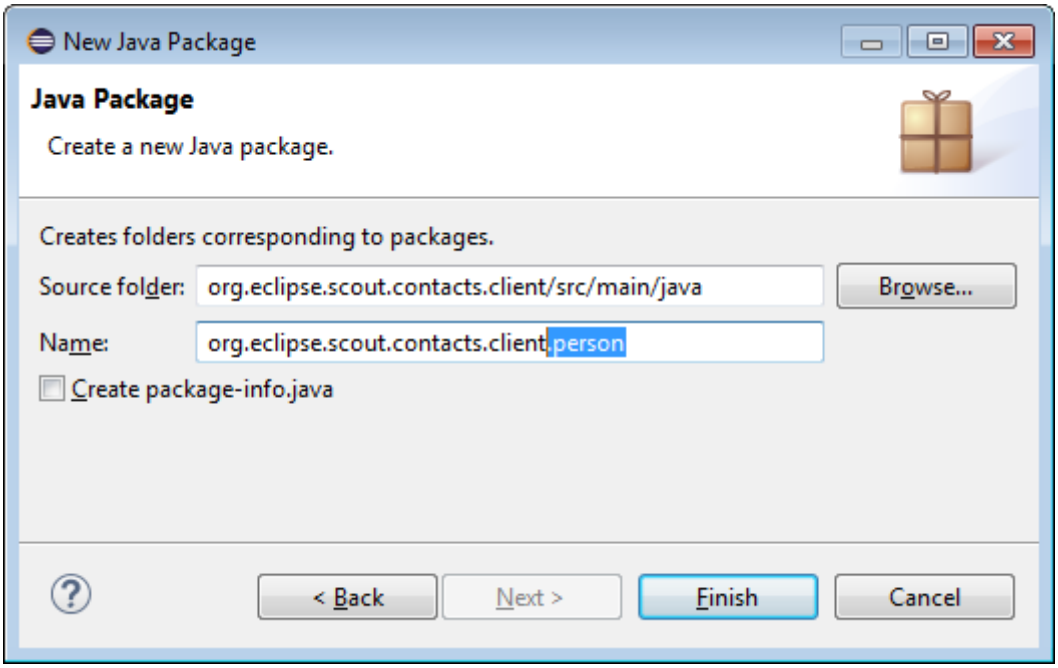


Figure 29. Add the person package to the "Contacts" application.

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`

4.4.2. 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.

Listing 5. The Scout lookup call for countries. This lookup call will be used for the address field.

```
public class CountryLookupCall extends LocalLookupCall<String> { ①

  private static final long serialVersionUID = 1L;

  @Override
  protected List<LookupRow<String>> execCreateLookupRows() { ②
    List<LookupRow<String>> rows = new ArrayList<>();

    for (String countryCode : Locale.getISOCountries()) {
      Locale country = new Locale("", countryCode);
      rows.add(new LookupRow<>(countryCode, country.getDisplayCountry())); ③
    }

    return rows;
  }
}
```

① Makes the `CountryLookupCall` to work with key type `String`

② Defines the set of lookup rows to be used

③ Add a row with the country code as key and the country name as display value

Using the implementation provided in Listing Countries lookup call create class `CountryLookupCall` in package `org.eclipse.scout.contacts.client.common`.

4.4.3. 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.common` 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*

¥ Enter `PersonPage` as the class name and click *Finish* as shown in Figure 000

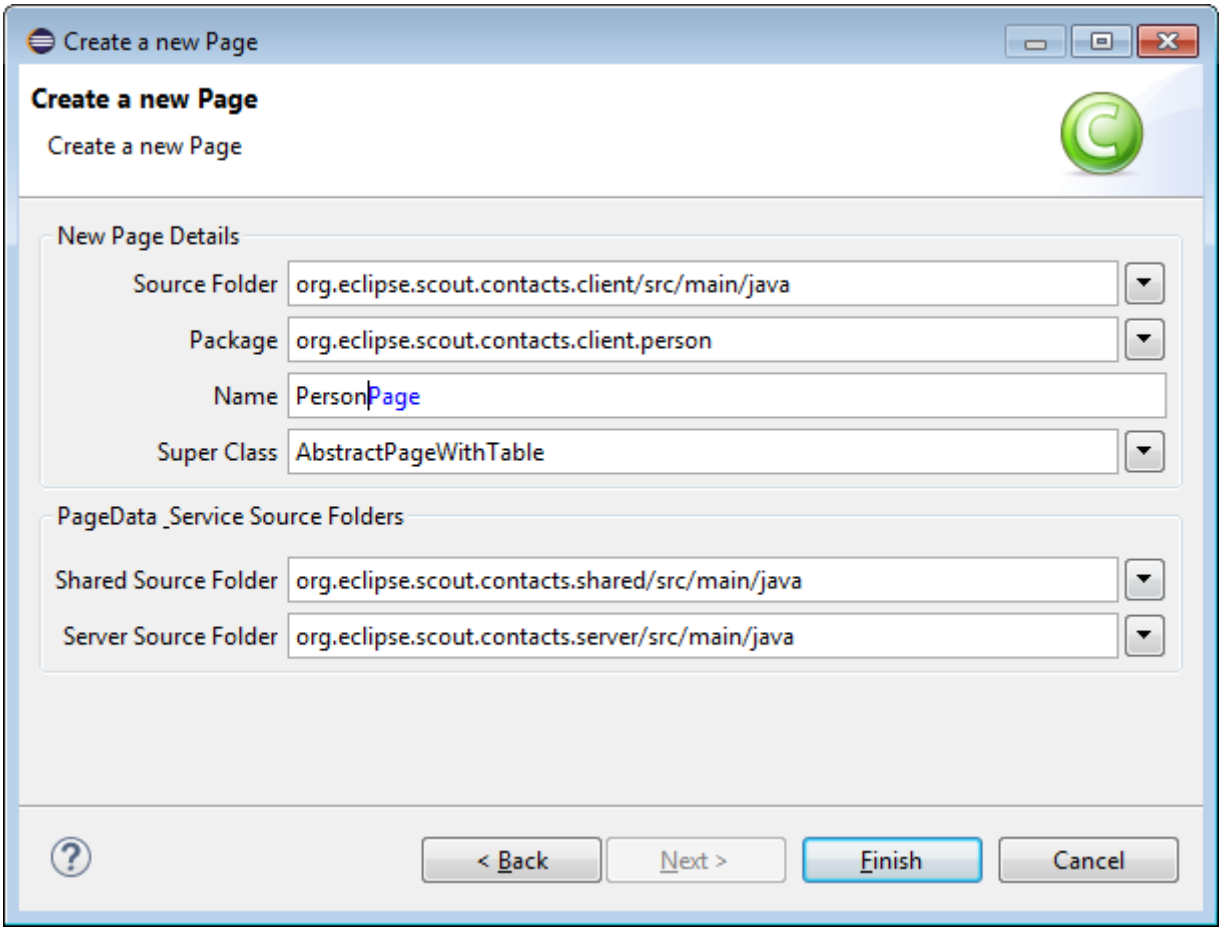


Figure 30. Add the person page to the "Contacts" application.

The Scout *New Page Wizard* then creates an initial implementation for the `PersonPage` class very similar to the listing provided in Listing PersonPage below.

Listing 6. Initial implementation of class PersonPage.

```
@PageData(PersonPageData.class)
public class PersonPage extends AbstractPageWithTable<PersonPage.Table> {

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

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

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

  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 PersonPage.

① Specify the title "Persons" for the page using the Scout NLS tooling (see Adding a new Translated Text Entry)

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.

4.4.4. 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 on the columns to add to a table page.

To decide the columns to add the following criterias 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`

Use the column class names 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 `PersonPage` 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 described in section Create new Table Columns. In the first edit box we type "PersonId" and press `Enter`. To configure this column as an invisible primary key we modify the newly created column class according to Listing PersonIdColumn.

Listing 7. Implementation of the person primary key column PersonIdColumn.

```
@Order(1)
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 attibute as a primary key (or part of a primary key)

We can now add the additional columns `FirstNameColumn`, `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" as described in Adding a new Translated Text Entry.

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

Listing 8. Implementation of the first name column.

```
@Order(2)
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 "City" as the translated text.

In the created class `CountryColumn` we need to update the class to extend

AbstractSmartColumn<String> and add the method getConfiguredLookupCall according to Listing CountryColumn.

Listing 9. Implementation of the country smart column.

```
@Order(5)
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 three columns PhoneColumn, MobilePhoneColumn and EmailColumn that are initially not visible in the user interface. As an example for such a column Listing PhoneColumn is provided below.

Listing 10. Implementation of the (initially invisible) phone column.

```
@Order(6)
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 of the desktop class of the application to the newly created `PersonPage` class. This is what we will do in the following section.

4.4.5. 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 adding the person page

Listing 11. Adding the PersonPage to the ContactOutline.

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

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

Before we can save the class `ContactOutline` we have to update its imports to include the import statement for class `PersonPage`. To update the imports you can either user the menu **Source | Organize Imports** or use the keyboard shortcut `Ctrl+Shift+O`.

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.

4.4.6. 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 client package `org.eclipse.scout.contacts.client.organization`
- Add page `OrganizationPage` with title "Organizations" using the Scout new page wizard

Listing 12. Initial implementation of class OrganizationPage.

```
@PageData(OrganizationPageData.class)
public class OrganizationPage extends AbstractPageWithTable<OrganizationPage.Table> {

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

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

  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 `OrganizationPage` using the Scout new page wizard then looks as shown in Listing OrganizationPage.

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 `OrganizationPage` in method `execCreateChildPages` of the outline class `ContactOutline` as shown in Listing adding the organization page to the outline.

Listing 13. Adding the OrganizationPage to the ContactOutline.

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

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

4.4.7. 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 the look as the screenshot provided in Figure 000.

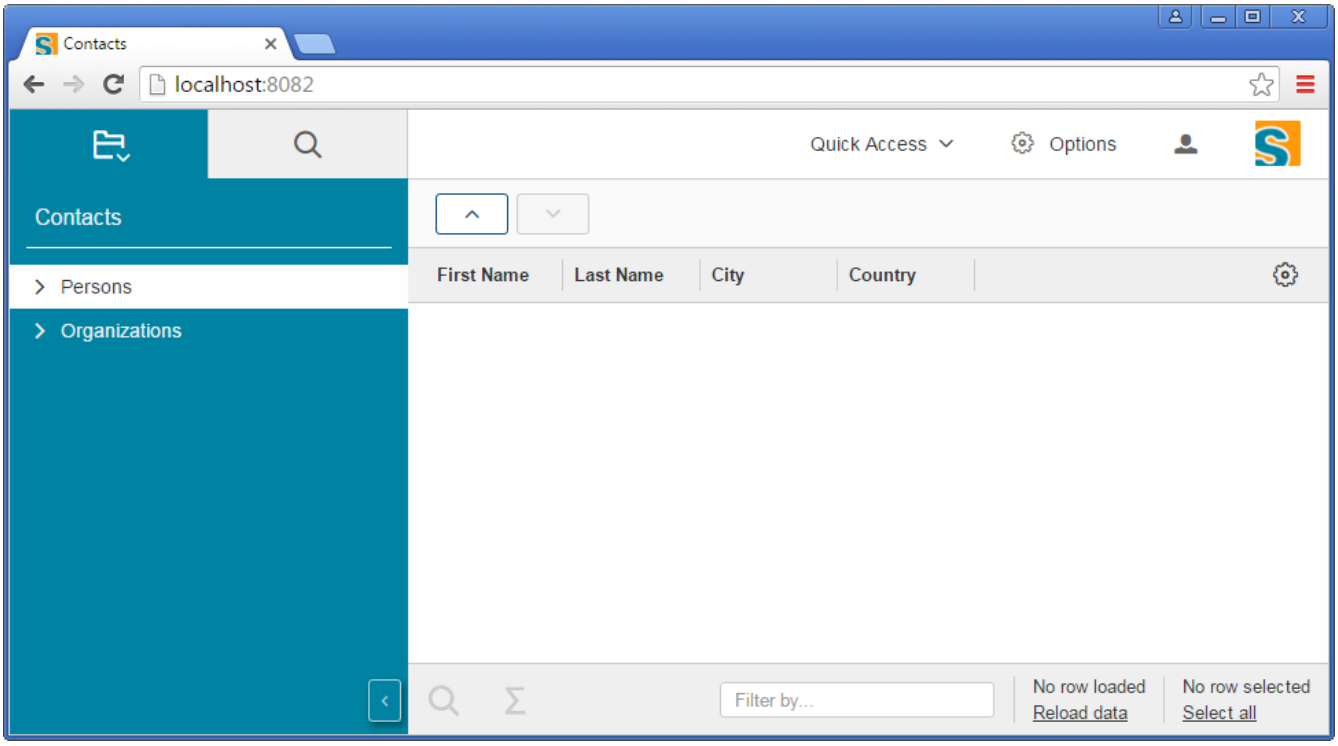


Figure 31. The "Contacts" application with the person and organization pages at the end of tutorial step 2.

When comparing the state of the "Contacts" tutorial application with the Scout demo application in Figure 000 the main difference is the missing person data. Adding access to a database is the focus of the next tutorial step.

4.5. Creating and Accessing the Database

This tutorial step shows how Scout applications can interact with databases via JDBC. As the "Contacts" application implements a clean layering, only the Scout backend server connects to the database. We can therefore completely 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 added infrastructure is then explained in the sections listed below.

- Scout Config Properties
- The SQL Service and SQL Statements
- The Database Setup Service

With the basic infrastructure in place we first review the already existing backend application and answer the question What is missing?. Section Fetching Organization and Person Data then describes how to 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?.

4.5.1. 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 `org.eclipse.scout.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 additional Maven dependencies

Listing 14. 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>6.0.0-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>
    <dependency>
      <groupId>org.apache.derby</groupId>
      <artifactId>derby</artifactId>
      <version>10.11.1.1</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.

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

- Expand folder `src/main/java` of the Maven server module in the Eclipse Package Explorer
- Select the existing package `org.eclipse.scout.contacts.server.sql` and hit `Ctrl+N`
- This opens the dialog to select a wizard. Enter "package" into the serach field
- Double click on the Java *Package* proposal to select the *New Java Package* wizard
- 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 A One Day Tutorial 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 000.

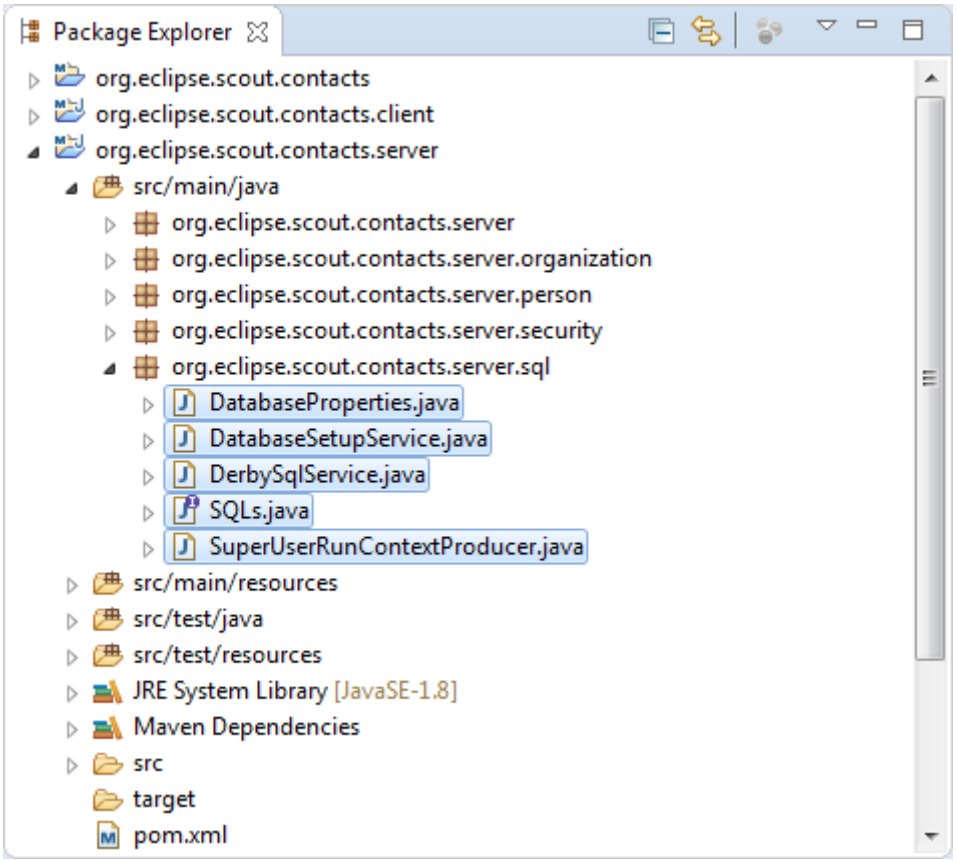


Figure 32. The copied database classes in the tutorial workspace.

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.

4.5.2. 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 becomes possible without any reprogramming. It is enough to exchange the parameter values in a properties file.

The Scout backend (and frontent) application initializes all defined config properties from matching values found in the file `config.properties` or the default values defined in the config property class. In the case of the "Contacts" application the config property files are located in the subfolder `src/main/resrouces` of the Maven modules that specify the frontend and the backend application. Example entries for the "Contacts" database setup are provided in Listing database properties. We can use these entries for the configuration of the "Contacts" backend server. Just add these lines to the file config.properties in Maven module `org.eclipse.scout.contacts.server.app.dev`.

Listing 15. Properties relevant for creating and accessing the database.

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

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

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` as shown in Listing config properties. This listing also shows how thematically related properties can be implemented as inner classes.

Listing 16. Typed properties for the "Contacts" application

```
public class DatabaseProperties {

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

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

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

  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

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

4.5.3. 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 Derby a Derby SQL service class can be created using the `AbstractDerbySqlService` provided by the Scout framework. The only project specific method is `getConfiguredJdbcMappingName` as implemented in Listing DerbySqlService.

Listing 17. 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 autocreate 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.mapping.name` 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`.

Having described the Derby SQL service that we will use in the "Contacts" application the text below discusses how the actual SQL statements are handled. For the "Contacts" application all statements used by the application are defined in class `SQLs`. 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 Eclispe IDE when using the statements
- The setup is easy to explain

The SQL statements related to the database structure are provided in Listing SQLs. The statements (or building blocks of statments) in class `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 18. 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"

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

4.5.4. 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 need to be integrated. In class `DatabaseSetupService` we will make use of the features listed below.

- 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 DatabaseSetupService. The actual creation of the "Contacts" database is performed by method `autoCreateDatabase` and before anything is created, the method checks if the auto creation is actually set to true.

For the database creation we first create a run context that is associated with the super user as no user is yet logged into the application. The context is then used to execute the runnable that creates the organization and person tables.

Listing 19. Class DatabaseSetupService to create the database tables for the "Contacts" application.

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

  @PostConstruct
  public void autoCreateDatabase() {
    if (CONFIG.getPropertyValue(DatabaseProperties.DatabaseAutoCreateProperty.class))
{
      try {
        RunContext context = BEANS.get(SuperUserRunContextProducer.class).produce();
        IRunnable runnable = new IRunnable() {

          @Override
          public void run() throws Exception {
            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(DatabaseProperties.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(DatabaseProperties.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.

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 Logback configuration. 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 20. 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.

4.5.5. What is missing?

This section reviews the backend infrastructure that has been created previously and identify 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 `OrganizationPage` with method `execLoadData` in the client module
- ¥ The service interface `IOrganizationService` and class `OrganizationPageData` in the shared module
- ¥ Class `OrganizationService` with the method stub `getTableData` in the server module

Listing 21. Accessing the "Contacts" backend server to fetch organization data.

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

On the client side the server roundtrip is implemented in method `execLoadData` as shown in Listing loading data from the server. This roundtrip between class `OrganizationPage` and `OrganizationService` works through the following steps.

1. `BEANS.get(IOrganizationService.class)` returns a reference to a client proxy service
2. Method `getTableData(filter)` is executed on the corresponding server service
3. This method returns the organization data in the form of an `OrganizationPageData` 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 getTableData.

Listing 22. Method getTableData to access the database and map the data into a pageData object.

```
public class OrganizationService implements IOrganizationService {

  @Override
  public OrganizationPageData getTableData(SearchFilter filter) {
    OrganizationPageData pageData = new OrganizationPageData();
    // fetch database data and store it into the pageData object
    return pageData;
  }
}
```

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

4.5.6. 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 `getTableData` in class `OrganizationService` we will use the two SQL snippet provided in Listing SELECT FROM ORGANIZATION.

Listing 23. 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_INTO = ""
      + "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 OrganizationService for the full listing of method `getTableData`. After adding the two additional lines, we update the imports of the classes with pressing `Ctrl+Shift+O`.

Listing 24. Method getTableData to access the database and map the data into a pageData object.

```
public class OrganizationService implements IOrganizationService {

  @Override
  public OrganizationPageData getTableData(SearchFilter filter) {
    OrganizationPageData pageData = new OrganizationPageData();
    // fetch database data and store it into the pageData object

    String sql = SQLs.ORGANIZATION_PAGE_SELECT + SQLs
.ORGANIZATION_PAGE_DATA_SELECT_INTO; 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_INTO` statement.

4.5.7. 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 000.

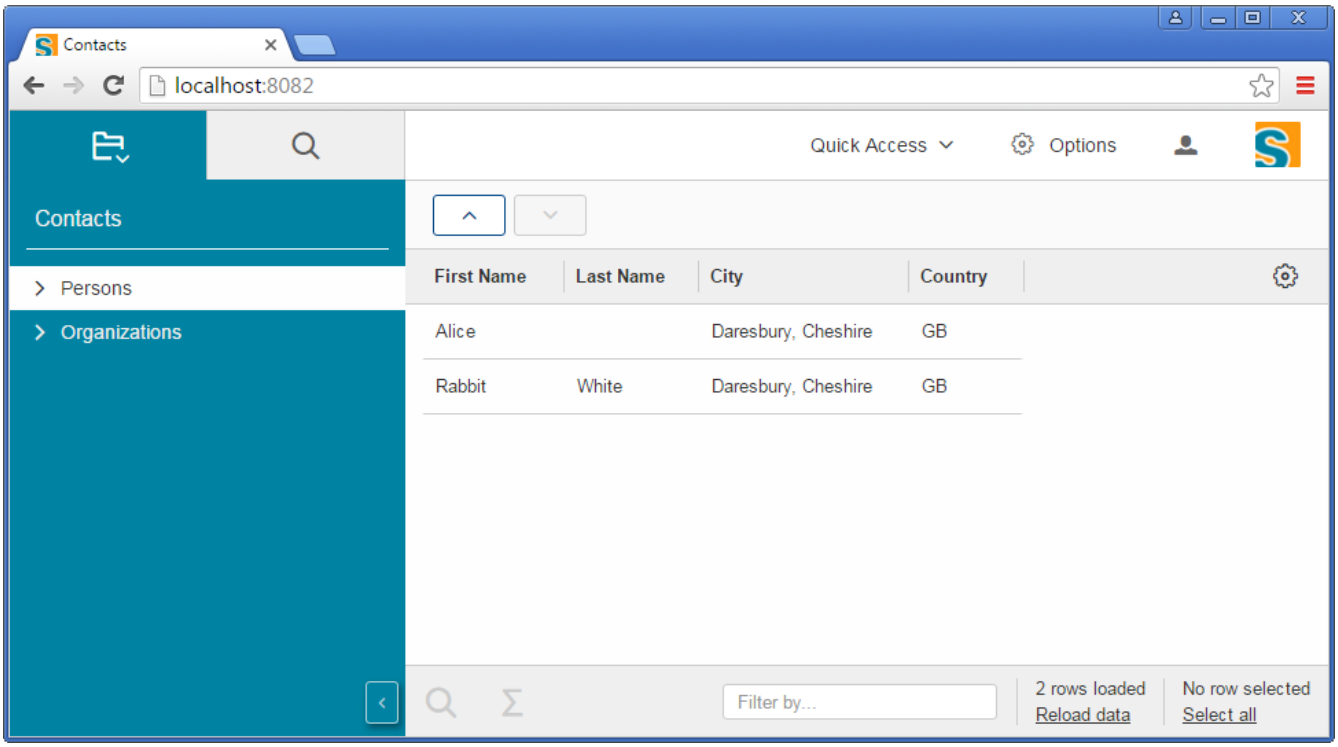


Figure 33. The "Contacts" application displaying person data at the end of tutorial step 3.

4.6. Adding Forms to Create/Edit Persons and Organizations

In this tutorial step we add the Scout forms that are used to create and edit persons and organizations in the user interface.

Before we start with the actual implementation section Designing the Person Form provides an introduction in the design of the person form and the general layouting concepts of the Scout framework. We then dive into the details of the implementation of the person form.

- Implementing the Form
- Adding a Gender Code Type
- Adding Form Fields
- Person Form Handler and Person Service
- Adding Form Field Validation

Before we implement the contact form in Adding the Company Form we modify the person form code to be able to reuse the picture field and the address box for the contacts form as described in Refactoring Form Fields. Finally, a summary of this tutorial step is provided in section What have we achieved?.

4.6.1. Designing the Person Form

We start with the sketch of the form layout as shown in Figure 000.

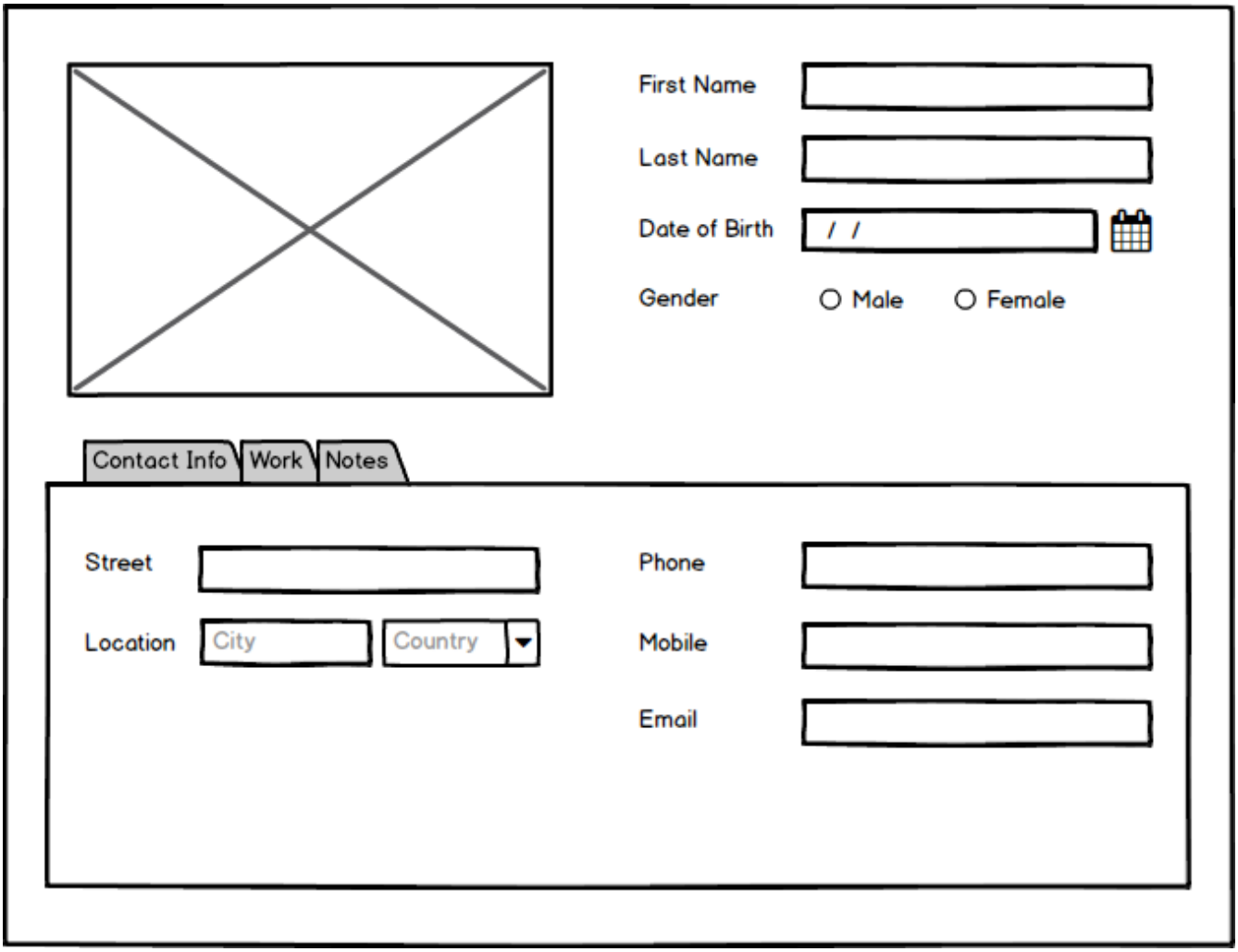


Figure 34. A sketch of the target layout for the person form.

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 "Details" tab provides contact details of the person and adding notes for the person in the form of free text is possible in the "Comments" tab.

Figure 000 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 are needed.

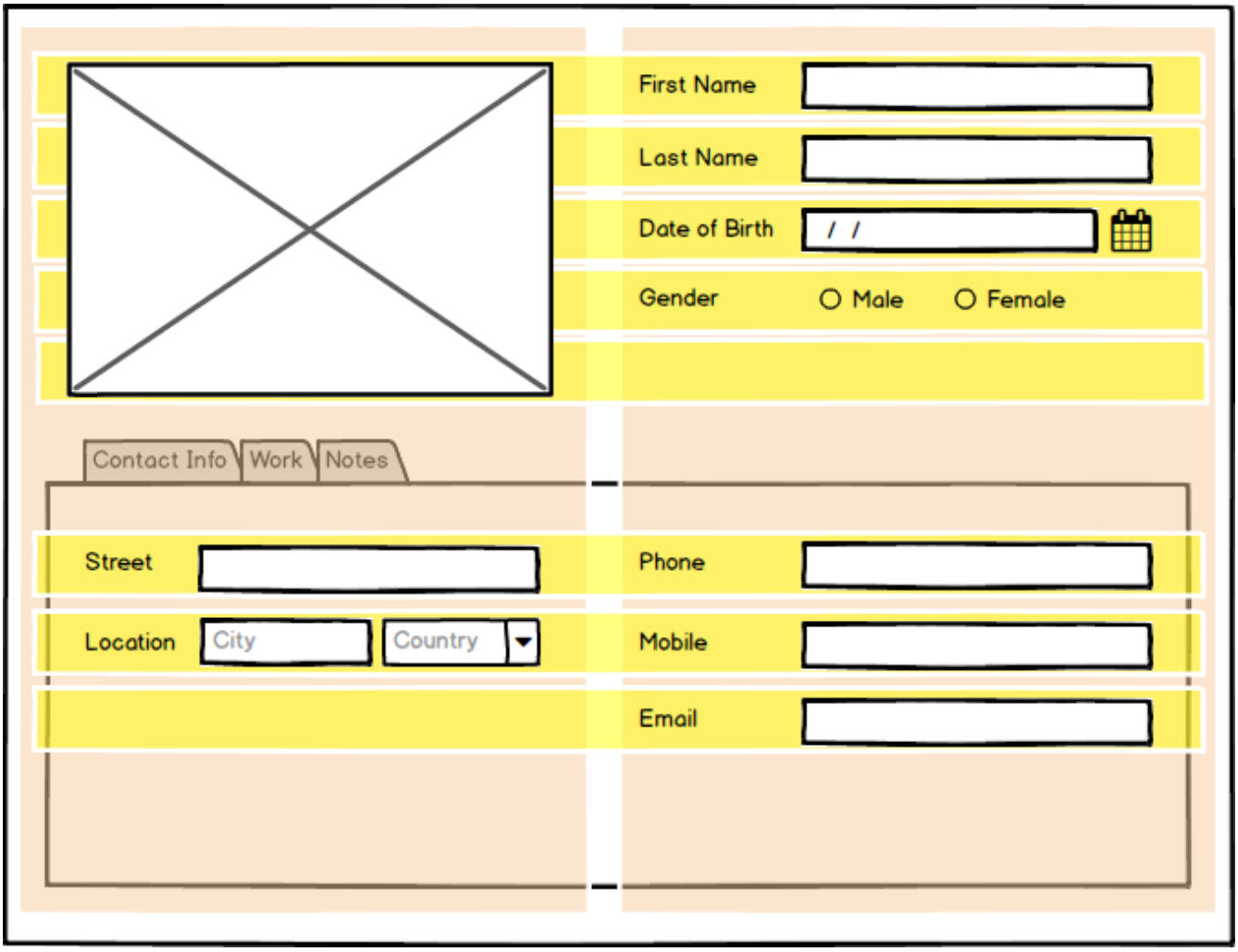


Figure 35. Logical columns and rows of the Scout form layout. Scout containers have two columns per default.

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 Scoutʼs 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 000

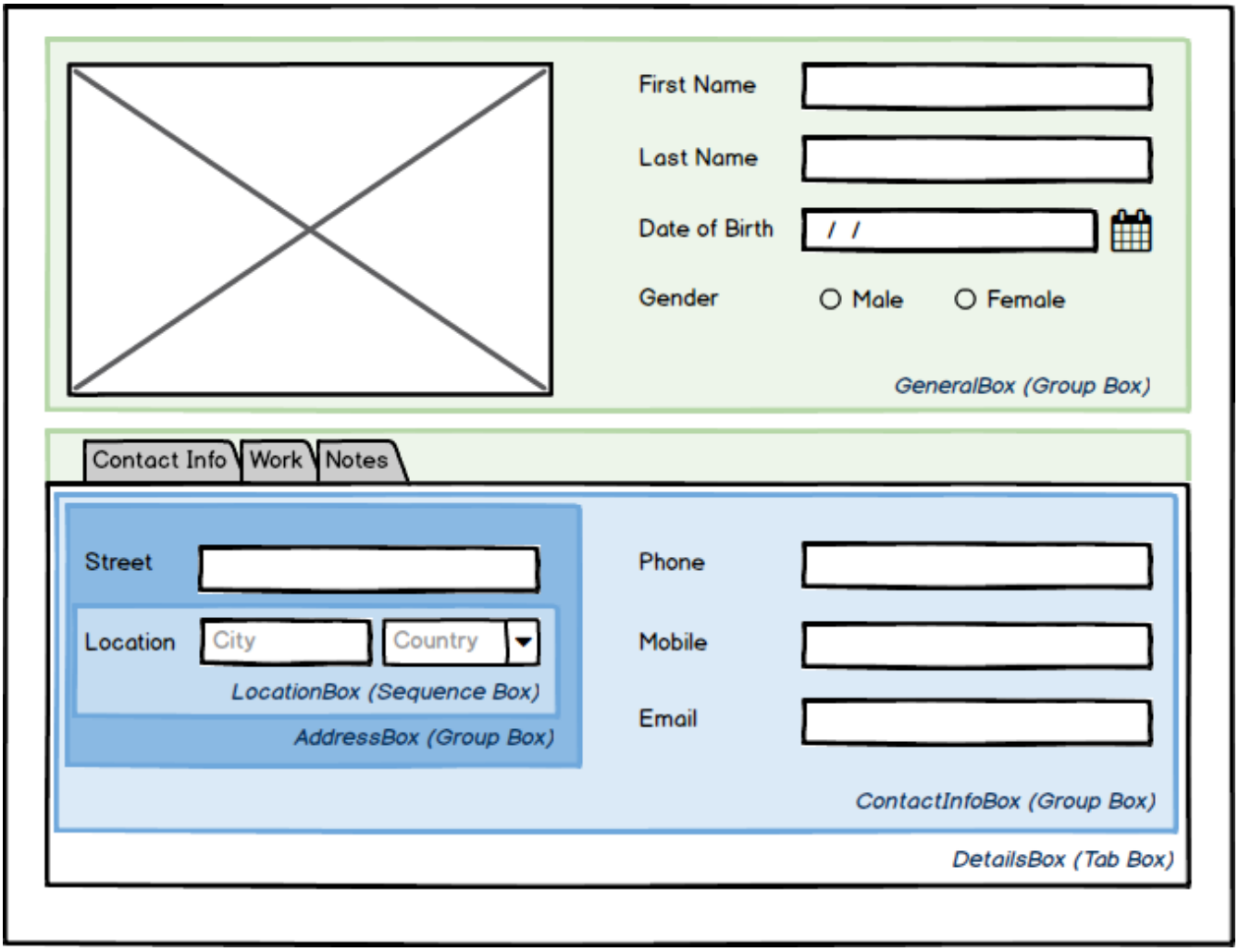


Figure 36. The hierarchical organization of the form including Scout group boxes, tab boxes and a sequence box.

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.

4.6.2. 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 "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 `org.eclipse.scout.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 000
6. Click **[Finish]** to start the creation of the form and its related components

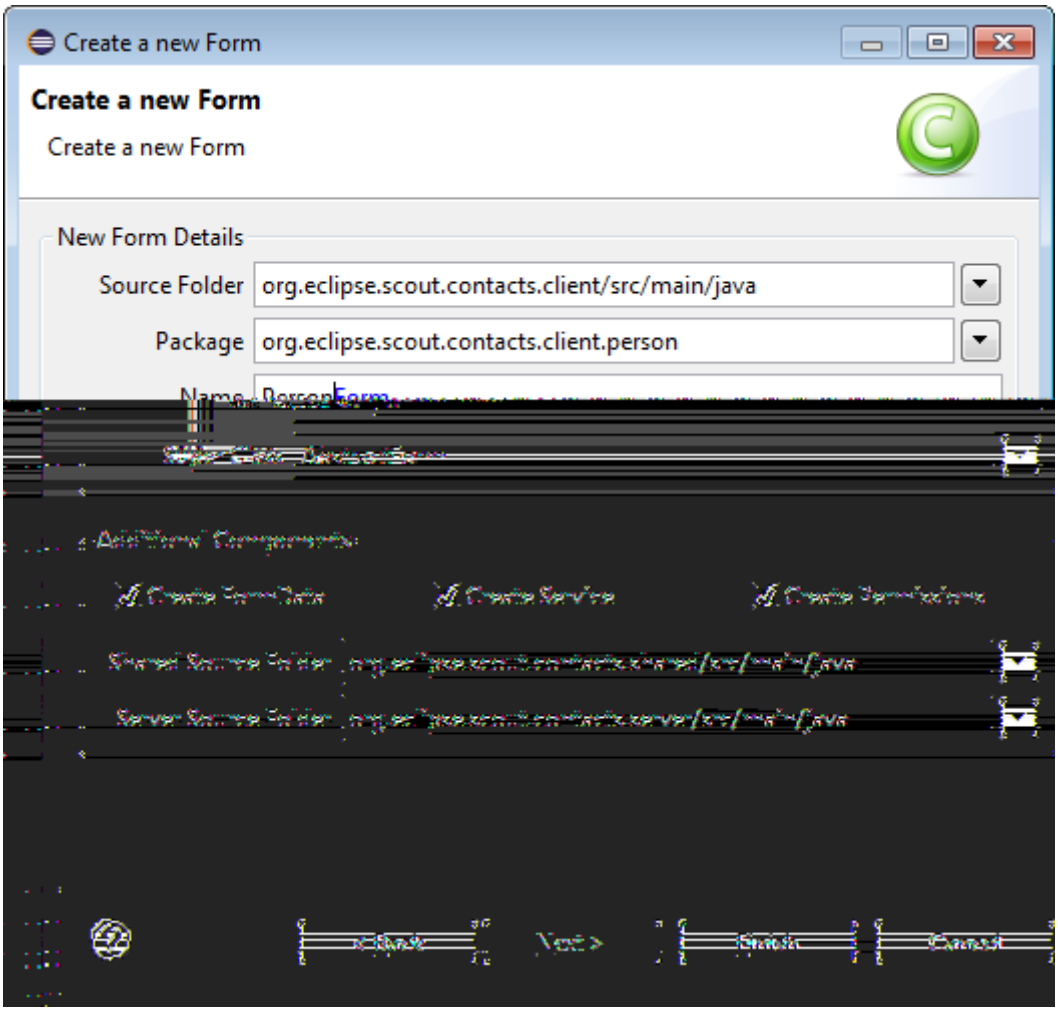


Figure 37. Use the New Scout Form to create the person form.

Now open the newly created class `PersonForm` in the Java editor and perform the changes listed below as shown in Listing PersonForm initial code.

- 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.

```
@FormData(value = PersonFormData.class, sdkCommand = FormData.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 PersonForm layout. First add classes `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.

Below the general box create class `DetailsBox` as a tab box by choosing the *Tab Box* proposal in the Scout content assist. Then, create the individual tab containers "Contact Info", "Work" and "Notes" as as inner classes of the details box according to Listing PersonForm layout.

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

```
public class PersonForm extends AbstractForm {

  @Order(10)
  public class MainBox extends AbstractGroupBox { ①

    @Order(10)
    public class GeneralBox extends AbstractGroupBox { ②
    }

    @Order(20)
    public class DetailsBox extends AbstractTabBox { ③

      @Order(10)
      public class ContactInfoBox extends AbstractGroupBox { ④

        @Order(10)
        public class AddressBox extends AbstractGroupBox {
        }
      }

      @Order(20)
      public class WorkBox extends AbstractGroupBox {
      }

      @Order(30)
      public class NotesBox extends AbstractGroupBox {
      }
    }

    @Order(100)
    public class OkButton extends AbstractOkButton {
    }

    @Order(110)
    public class CancelButton extends AbstractCancelButton {
    }
  }
}
```

① 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. The most common way to open a form is to add corresponding context menus to pages showing data of the

same entity. Following this pattern we add the context menus "New" and "Edit" to the person page according to Listing page context menus.

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

```
@PageData(PersonPageData.class)
public class PersonPage extends AbstractPageWithTable<PersonPage.Table> {

  public class Table extends AbstractTable {

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

    @Order(10)
    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());
        form.startModify(); ③
      }
    }

    @Order(20)
    public class NewMenu extends AbstractMenu {

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

      @Override
      protected Set<? extends IMenuType> getConfiguredMenuTypes() { ④
        return CollectionUtility.<IMenuType> hashSet(
            TableMenuType.EmptySpace, TableMenuType.SingleSelection);
      }

      @Override
      protected void execAction() {
        PersonForm form = new PersonForm();
        form.addFormListener(new PersonFormListener());
        form.startNew(); ⑤
      }
```

```
    }

    private class PersonFormListener implements FormListener {

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

① This action gets executed when the user presses `Enter` 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 The "Edit" menu is using the modify handler. This handler loads the data using the person's primary key.

④ Including `TableMenuType.EmptySpace` in the return value activates the "New" menu even when no row is selected.

⑤ The "New" menu opens the form using the new handler.

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 top level person menu.

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

```
public class Desktop extends AbstractDesktop {

  @Order(10)
  public class QuickAccessMenu extends AbstractMenu {

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

    @Order(10)
    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.

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.

4.6.3. 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 `org.eclipse.scout.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
6. Click button **[Finish]**
7. Set the `ID` constant to "Gender" in the created class `GenderCodeType`

The created class `GenderCodeType` will then look like Listing Gender code type except for the missing inner code classes.

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

```
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)
  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)
  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 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.

4.6.4. 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. Then 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. To also store the URL pointing to the picture we also add the invisible field `PictureUrlField` according to Listing picture field.

Listing 30. The picture field for the person form.

```
@Order(10)
public class PictureUrlField extends AbstractStringField {

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

@Order(20)
public class PictureField extends AbstractImageField {

  @Override ②
  protected Class<? extends IValueField> 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;
    }

    protected void updateImage(String url) {
      clearErrorStatus(); ④

      if (url == null) {
        setImage(null);
      }
      else {
        try {
          setImage(IOUtility.readFromUrl(new URL((String) url)));
          setAutoFit(true);
        }
        catch (Exception e) { ⑤
          String message = TEXTS.get("FailedToAccessImageFromUrl");
          addErrorStatus(new Status(message, IStatus.WARNING));
        }
      }
    }
  }
```

① Sets the field invisible. An invisible field does not need 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.

④ Clears any field error status.

⑤ Sets the field error status in case of an exception during the loading of the image.

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.

The remaining fields for the general box can then be added using the Scout content assist or by copying Listing picture field into the code below the picture field.

Listing 31. The other fields in the general box.

```
@Order(30)
public class FirstNameField extends AbstractStringField {

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

@Order(40)
public class LastNameField extends AbstractStringField {

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

@Order(50)
public class DateOfBirthField extends AbstractDateField {

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

@Order(60)
public class GenderGroup extends AbstractRadioButtonGroup<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 notes box.

Listing 32. The notes tab box with its multi line text field.

```
    @Order(30)
    public class NotesBox extends AbstractGroupBox {

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

      @Order(10)
      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 box. 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 can implement the location box as a sequence field according to Listing address box. To further optimize screen real estate we also switch to on-field labels for the city field and the country field.

Listing 33. The content of the address box.

```
    @Order(10)
    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)
        public class StreetField extends AbstractStringField {

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

        // use a sequence box for horizontal layout ③
        @Order(20)
        public class LocationBox extends AbstractSequenceBox {

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

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

          @Order(10)
          public class CityField extends AbstractStringField {

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

            @Override
```

```
      protected int getConfiguredLabelPosition() {
        return LABEL_POSITION_ON_FIELD; ⑤
      }
    }

    @Order(20)
    public class CountryField extends AbstractSmartField<String> {

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

      @Override
      protected int 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. To use the form to view and enter person data 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.

4.6.5. 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 form handler.

Listing 34. The new handler and modify handler for the person form.

```
public class PersonForm extends AbstractForm {


  public class ModifyHandler extends AbstractDirtyFormHandler {

    @Override
    protected void execLoad() {
      IPersonService service = BEANS.get(IPersonService.class); ①
      PersonFormData formData = new PersonFormData();
      exportFormData(formData); ②
      formData = service.load(formData); ③
      importFormData(formData); ④

      getForm().setSubTitle(calculateSubTitle()); ⑤
    }

    @Override
    protected void execStore() {
      IPersonService service = BEANS.get(IPersonService.class);
      PersonFormData formData = new PersonFormData();
      exportFormData(formData);
      service.store(formData); ⑥
    }
  }

  public class NewHandler extends AbstractDirtyFormHandler {

    @Override
    protected void execStore() {
      IPersonService service = BEANS.get(IPersonService.class);
      PersonFormData formData = new PersonFormData();
      exportFormData(formData);
      service.create(formData); ⑦
    }
  }

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

① Obtains a reference to the person service located on the Scout backend application.

② All form field values are transferred 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.

③ 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 form handler 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 PersonService methods.

Listing 35. 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 PersonCreatePermission())) {
      throw new VetoException(TEXTS.get("InsufficientPrivileges"));
    }

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

    SQL.insert(SQLs.PERSON_INSERT, formData); ①

    return store(formData); ②
  }

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

    SQL.selectInto(SQLs.PERSON_SELECT, formData); ③

    return formData;
  }

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

    SQL.update(SQLs.PERSON_UPDATE, formData); ④

    return formData;
  }
}
```

① The SQL insert statement adds adds a new person entry in the database. Only the primary key is used to create this entry.

② To save all other person attributes provided in the form data, the `store` method is reused.

③ The SQL select into transfers the person data from the database into the form data.

④ The SQL update statement transfers all person attributes provided in the form data to the person table.

In this section we have implemented the interaction of the frontend with the backend application including the persistance of person data in the database. 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.

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. An efficient approach to address this need is to make sure that the user only enters valid data into the application. In the next section we add a number of form field validations that implement this approach for the person form.

4.6.6. Adding Form Field Validation

Form field validation helps the user to enter valid data into the form fields. The Scout framework allows the validation on the level of individual fields, on the level of container fields or on the level of the whole form. The text below provides a number of examples to illustrate these possibilities.

We start with the validation on the level of a single field using the email address field defined in the "Contact Info" tab. The code provided in Listing email validation checks the size and the formatting of the email address.

Listing 36. The validation of the email field

```
@Order(40)
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 pattern.

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 example we add field validation to the address box to ensure the following 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. To enforce the other rules described above we need to take into account several field values at the same time. A possible implementation for these rules is provided in Listing address validation.

Listing 37. The validation of the fields in the address box

```
@Order(10)
public class AddressBox extends AbstractGroupBox {

  @Order(10)
  public class StreetField extends AbstractStringField {

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

  @Order(20)
  public class LocationBox extends AbstractSequenceBox {

    @Order(10)
    public class CityField extends AbstractStringField {

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

    @Order(20)
    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.

To verify the user input before saving the content of a complete form (and not on changing the content of individual fields) validation can also be implemented on the form level. In Listing form validation we use this mechanism to make sure that the user does not save a person form with at least providing a first name or a last name.

Listing 38. 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 In case both the first name and the last name fields are empty place the focus on the first name field.

③ Throw a new veto exception that includes a message to inform the user to enter a first name or a last name.

④ Returning `true` is indicating that validation has passed successfully on the form level.

4.6.7. Refactoring Form Fields

In this section we refactor the picture field and the address box and the email field as template fields that allow these fields to be reused in both the person form and the company form.

When we create the company in the next section of this tutorial we want to display the company's logo in the form. From a technical point of view this is the same as showing a picture of a person. To avoid a copy&paste approach we will refactor the picture field of the person form into a picture box that can be used by both the person and the company form. For the same reason we will also refactor the address box and the email field for reuse in different forms of the application.

The initial implementation of the image field consists of two separate fields, the `PictureField` and the `PictureUrlField`. To allow the user to edit the picture's URL an "Edit URL" menu and a small form is also added during the refactoring of this field.

TODO refactor picture field

Listing 39. The refactored picture field.

```
@FormData(value = AbstractUrlImageFieldData.class, ①
    sdkCommand = FormData.SdkCommand.CREATE,
    defaultSubtypeSdkCommand = FormData.DefaultSubtypeSdkCommand.CREATE)
public class AbstractUrlImageField extends AbstractImageField {

  private String url; ②

  @FormData ②
  public String getUrl() {
    return url;
  }

  @FormData ②
  public void setUrl(String url) {
    this.url = url;
    updateImage();
  }

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

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

  protected void updateImage() {
    clearErrorStatus();

    if (url == null) {
      setImage(null);
    }
    else {
      try {
        setImage(IOUtility.readFromUrl(new URL((String) url)));
        setAutoFit(true);
      }
      catch (Exception e) {
        addErrorStatus(new Status(TEXTS.get("FailedToAccessImageFromUrl"), IStatus
.WARNING));
      }
    }
  }
}
```

TODO Edit Url form

Listing 40. The form to edit the picture URL

```
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);
  }

  @Order(10)
  public class MainBox extends AbstractGroupBox {

    @Order(10)
    public class UrlBox extends AbstractGroupBox {

      @Order(10)
      public class UrlField extends AbstractStringField {

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

    @Order(20)
    public class OkButton extends AbstractOkButton {
    }

    @Order(30)
    public class CancelButton extends AbstractCancelButton {
    }
  }

  public class ModifyHandler extends AbstractFormHandler { ②
  }
}
```

TODO Edit menu

Listing 41. The "Edit URL" menu for the refactored picture field

```
public class AbstractUrlImageField extends AbstractImageField {


  @Order(10)
  public class EditURLMenu extends AbstractMenu {

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

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

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

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

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

TODO refactor address box

Listing 42. The "Edit URL" menu for the refactored picture field

```
@FormData(value = AbstractAddressBoxData.class, ①
    sdkCommand = FormData.SdkCommand.CREATE,
    defaultSubtypeSdkCommand = FormData.DefaultSubtypeSdkCommand.CREATE)
public abstract class AbstractAddressBox extends AbstractGroupBox {
  // no modifications to the content of the address box necessary
}
```

4.6.8. Adding the Company Form

Add the Company Form: Copy & Paste

4.6.9. What have we achieved?

In the third step of the "Contacts" tutorial we have added Scout forms to edit and create persons and organisations in the "Contacts" application.

TODO: put more details here

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 <<img-contacts_tutorial_result_step_4]].

[Êcontacts tutorial result step 4Ê] |

C:\Users\jgu\git\org.eclipse.scout.docs\docs\build\beginners_guide/../../imgs/contacts_tutorial_result_ste

p_4.png

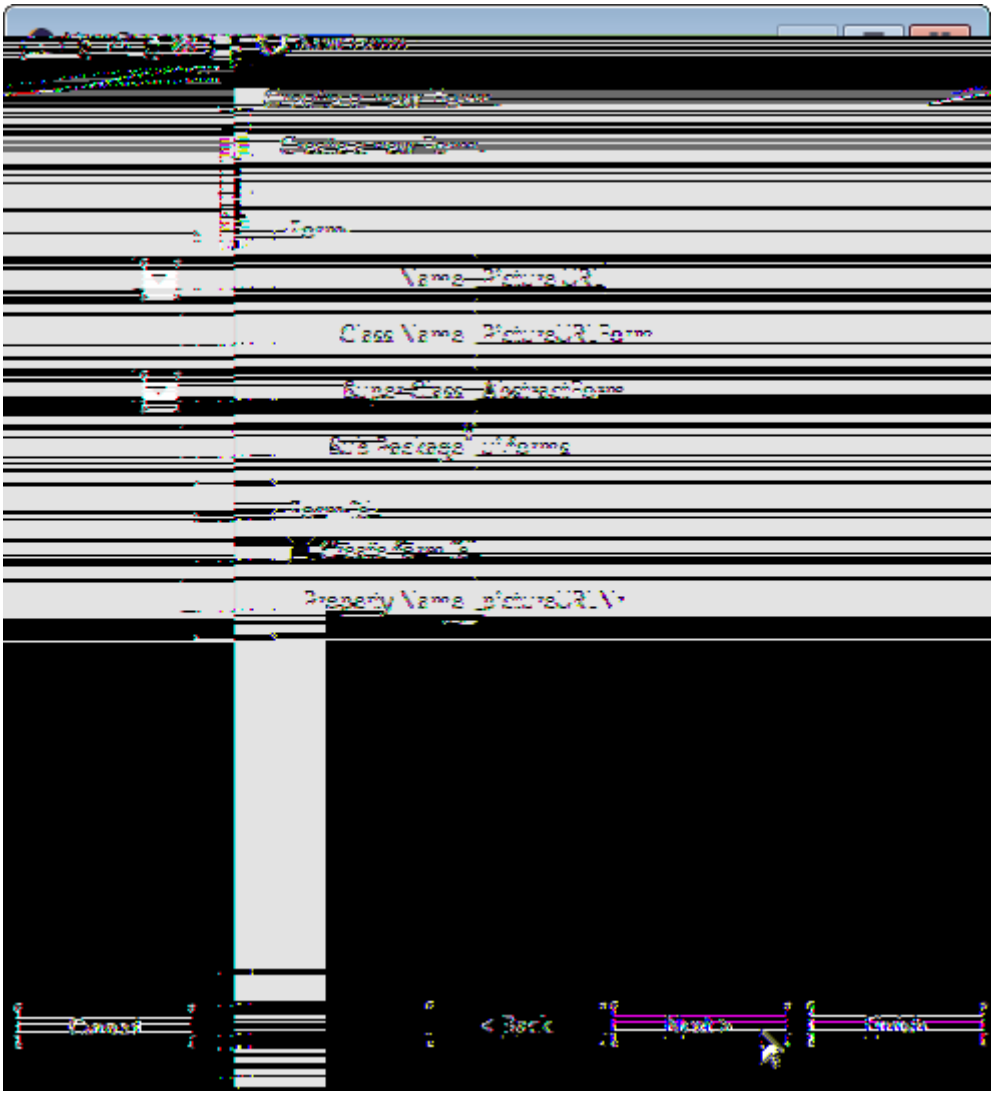
Figure 38. The "Contacts" application with the person form at the end of tutorial step 4.

4.7. Linking Oranizations and Persons

4.8. Next Steps and Outlook

Chapter 5. OLD STUFF

5.1. A simple Form to edit the Picture URL



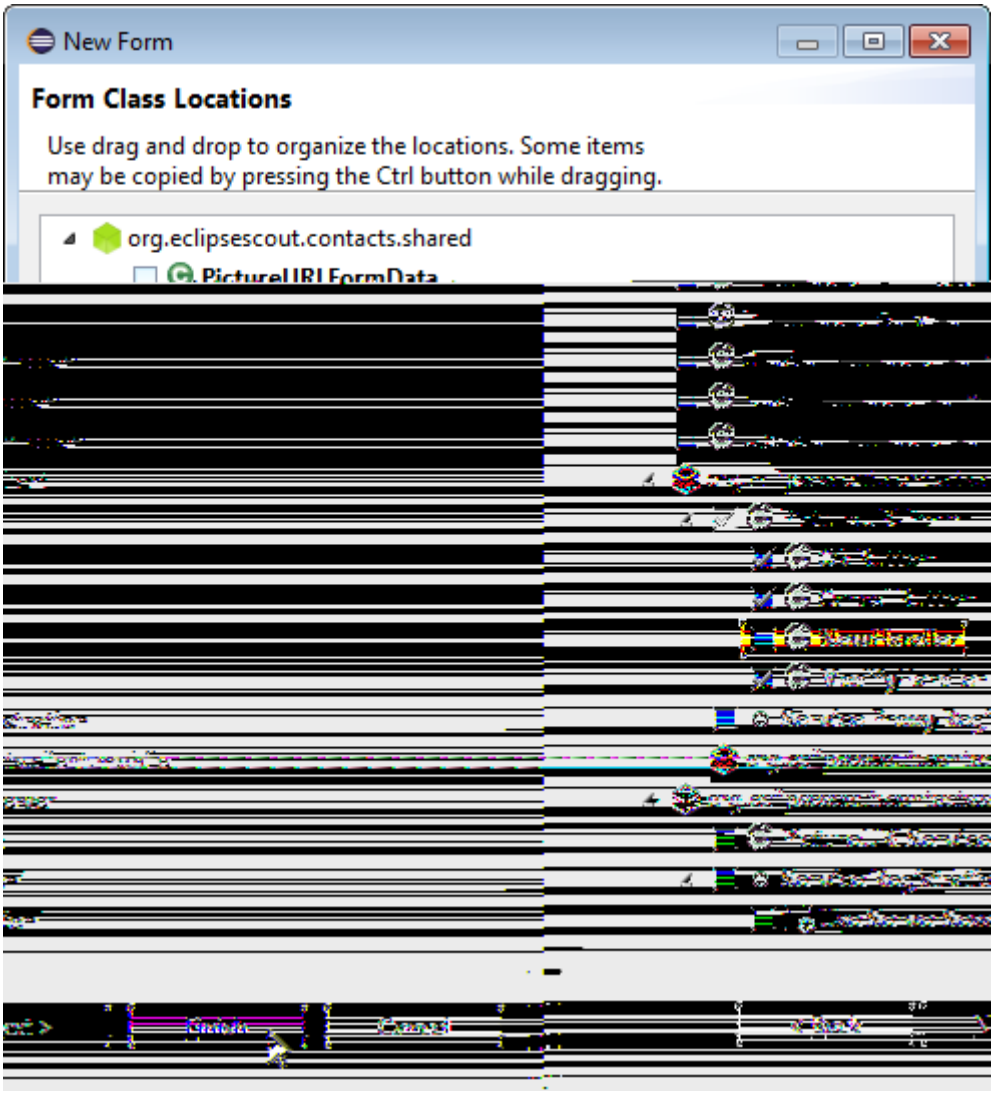


Figure 39. Add the URL editor form. This form does not need any connections to the server. Therefore, the related components such as services and the form data can be removed in the second wizard step shown on the right side.

To edit a personÕs picture link, we create a simple URL editor form as shown in Figure 000. As we only need this form to update the URL information of a personÕs picture field, we do not need any connectivity to the backend of the ÒMy ContactsÓ application. That is why almost all form and service artefacts are deselected in the second wizard step shown on the right side of Figure 000.

Listing 43. The UI structure of the PictureURLForm used to update the URL of the picture field in the person form.

```
TODO include was {codedir}/oneDayTutorial/org.eclipsescout.contacts.client/src/org
/eclipsescout/contacts/client/ui/forms/PictureURLForm.java[lines=24..25;99..113;122..1
22;126..127]
```

As this form only holds a single URL field, we omit the description of the creation of the URL editor formÕs content and provide the resulting Java code instead. In Listing PictureURLField just the formÕs MainBox code is shown.

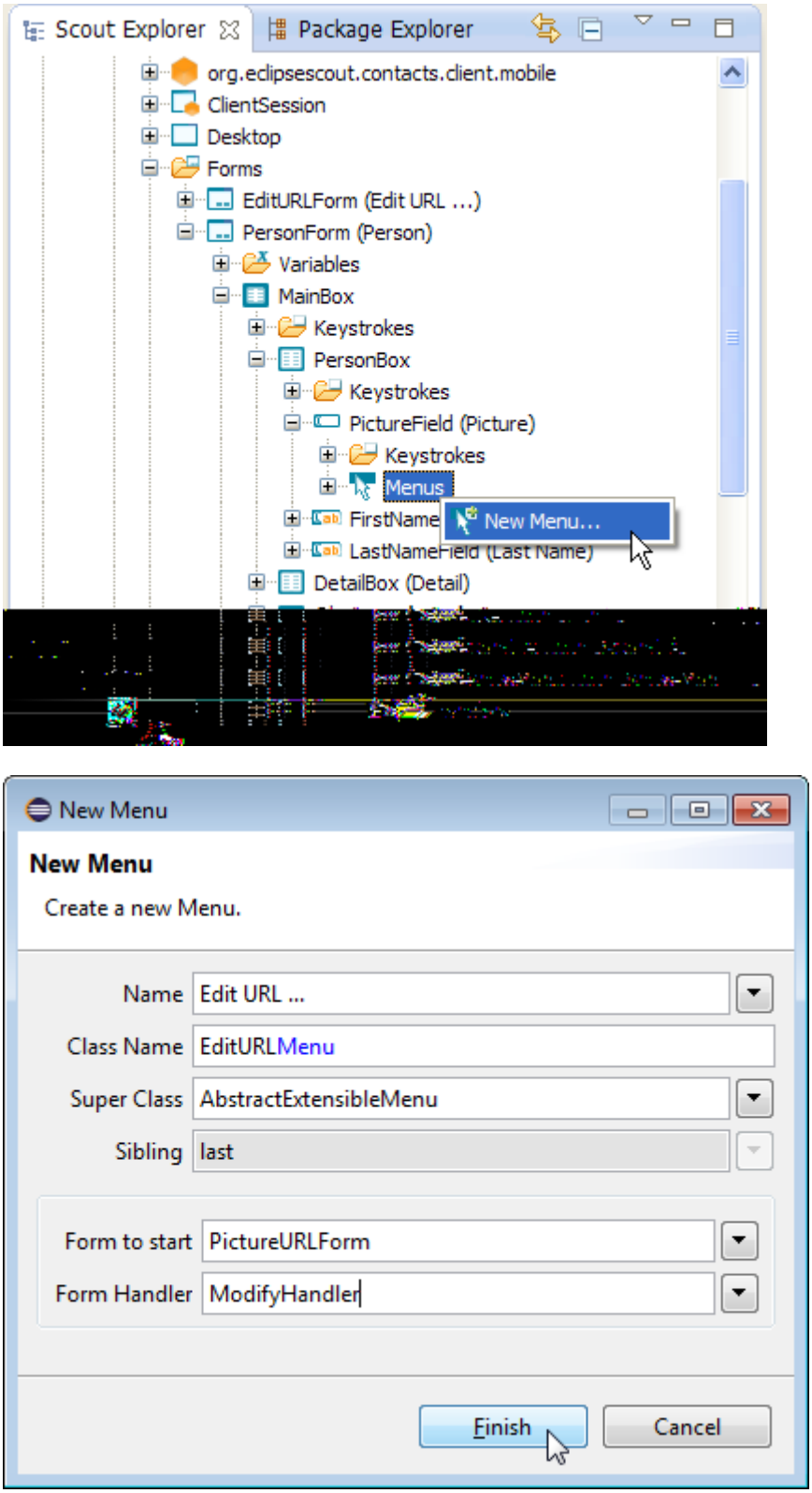


Figure 40. Add the URL edit menu to the picture field.

This form is then started via an ÒEdit URL ÉÓ contextmenu on the image field. The creation of this contextmenu is shown in Figure 000. See Listing EditURLMenu for the actual implementation of the execAction for this contextmenu.

Listing 44. The edit menu implemented in class EditURLMenu of the picture field. If the URL was changed the picture URL field of the person form is set accordingly in method execAction

```
TODO include was {codedir}/oneDayTutorial/org.eclipsescout.contacts.client/src/org
/eclipsescout/contacts/client/ui/forms/PersonForm.java[lines=268..285]
```

Once the edit URL form is started with form.startModify() the client waits in method form.waitFor until the form is closed by the user. If the user has changed any field content (the picture URL in our case) and closed the form with the **[OK]** button, the method form.isFormStored returns true, and we can copy the new URL from the editor form field into the picture URL field of the person form. Such a change will then trigger method execChangedMasterValue of the PictureField which in turn updates the image shown in the person form.

5.2. Adding the Company Smartfield

At the current stage of the ÒMy ContactsÓ application, we have no option to manage the relationship between people and companies. To manage this relation, we now add a company smart field to the person form. This smart field will then hold the current assignment of the person represented in the person form.

A Scout smart field can be viewed as user friendly dropdown field on steroids that also implements *search-as-you-type* to pick a specific entry. In the simplest case the smart field provides access to a small and locally provided list of key value pairs. But for the intended use in the ÒMy ContactsÓ application, we will need to access a list of elements provided by the server that will be compiled dynamically at runtime.

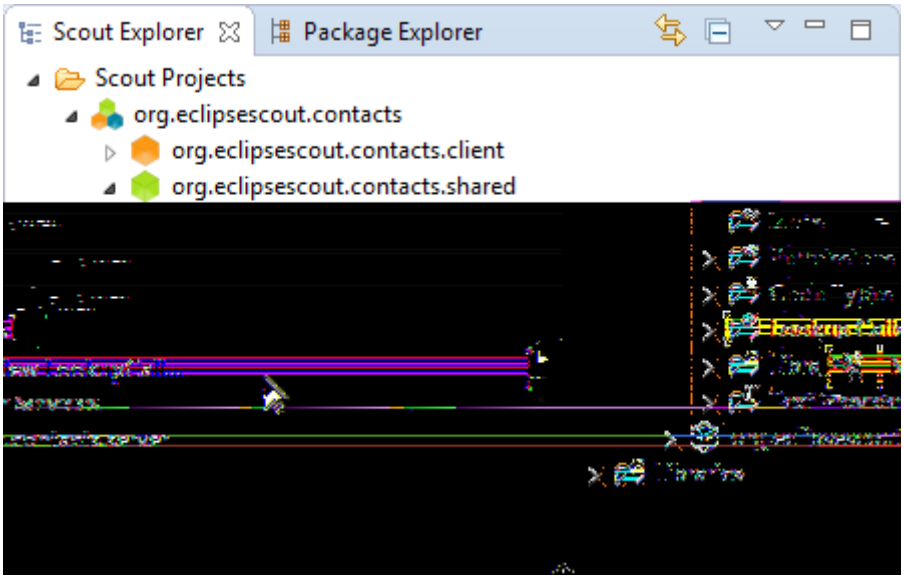


Figure 41. Add a lookup call to the applications shared node.

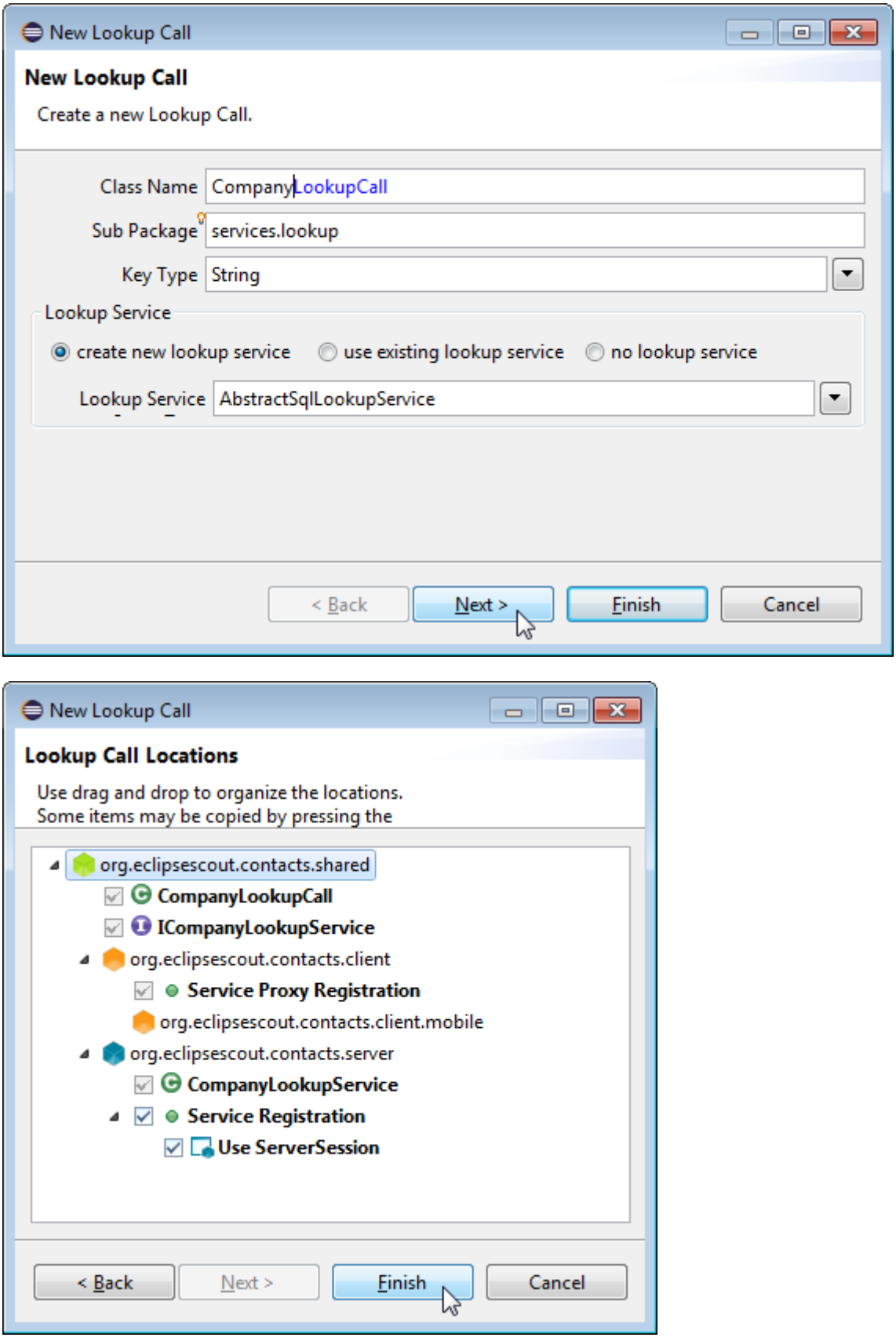


Figure 42. The two wizard steps to enter the details of the company lookup call.

To create the access to this list, we start with the creation of the company lookup call. As shown in Figure 000 the lookup call is added on the *Lookup Calls* folder under the green shared node of the ÒMy ContactsÓ application. This opens the *New Lookup Call* wizard as shown in Figure 000. In the first wizard step, enter ÒCompanyÓ into the *Class Name* field, Òservices.lookupÓ into the *Sub Package* field and ÒStringÓ into the *Key Type* field. Now verify that the wizard step looks the same as the

screenshot shown on the left hand side of Figure 000. Before the wizard is closed, click on the **[Next]** button to move to the second wizard step. As shown on the right hand side of Figure 000, the wizard will also create a corresponding CompanyLookupService on the application's server. We can now close this wizard with the **[Finish]** button and add the business logic to this company lookup service.

Listing 45. The company lookup call with its getConfiguredService method in the application's shared plugin.

```
TODO include was {codedir}/oneDayTutorial/org.eclipsescout.contacts.shared/src/org
/eclipsescout/contacts/shared/services/lookup/CompanyLookupCall.java[lines=12..21]
```

Listing 46. The company lookup service in the application's server plugin. The key and the text criteria are used to search for values by key or by the provided name substring.

```
TODO include was {codedir}/oneDayTutorial/org.eclipsescout.contacts.server/src/org
/eclipsescout/contacts/server/services/lookup/CompanyLookupService.java[lines=6..20]
```

The CompanyLookupCall just created by the Scout SDK wizard is provided in Listing CompanyLookupCall. As we can see, the only method implemented is getConfiguredService that points to the specific server service to be used. In the Scout Explorer, the new company lookup service can be found in the *Lookup Services* folder under the blue server node of the application. In this service, we need to implement method getConfiguredSqlSelect as shown in Listing CompanyLookupService. For Scout lookup services, specific *key*, *text* and *all* criteria blocks need to be provided. This criteria are included in the SELECT statement using the <key>, <text> and <all> tags as shown in the listing. The Scout runtime uses the <key>-block in cases where a specific key is already assigned to the smart field. The <text>-block is used as a query criteria to create the dynamic *search-as-you-type* hit list based on the (sub)string entered by the user so far. Finally, the <all>-block is used to define the hit list to be shown when the user does not enter any text into the smart field but clicks on the field's search icon instead. The bind variable :key and :text are provided by Scout and hold the value of the assigned key or the text entered into the smart field.

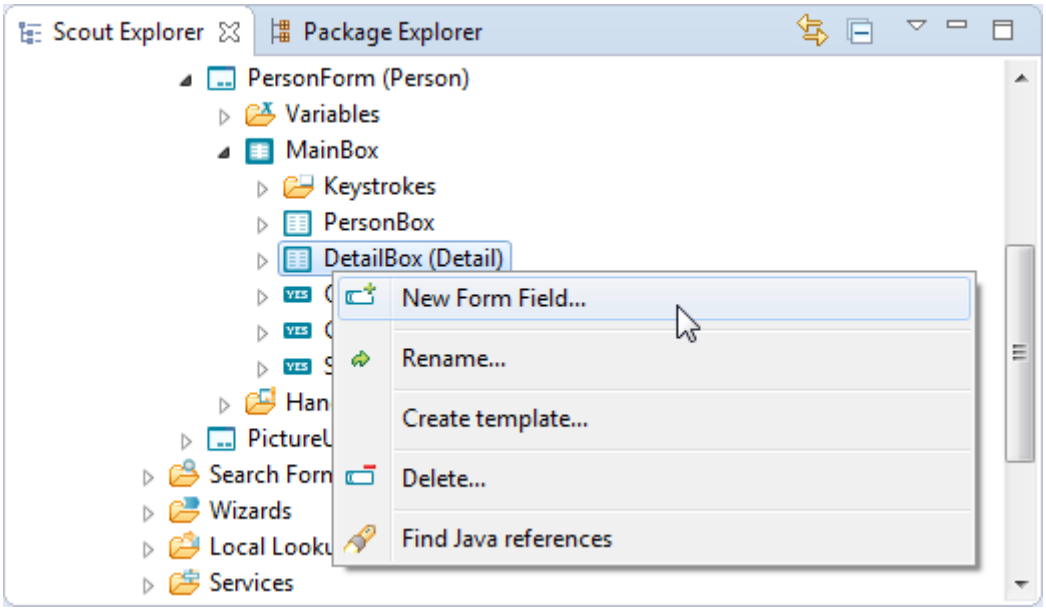
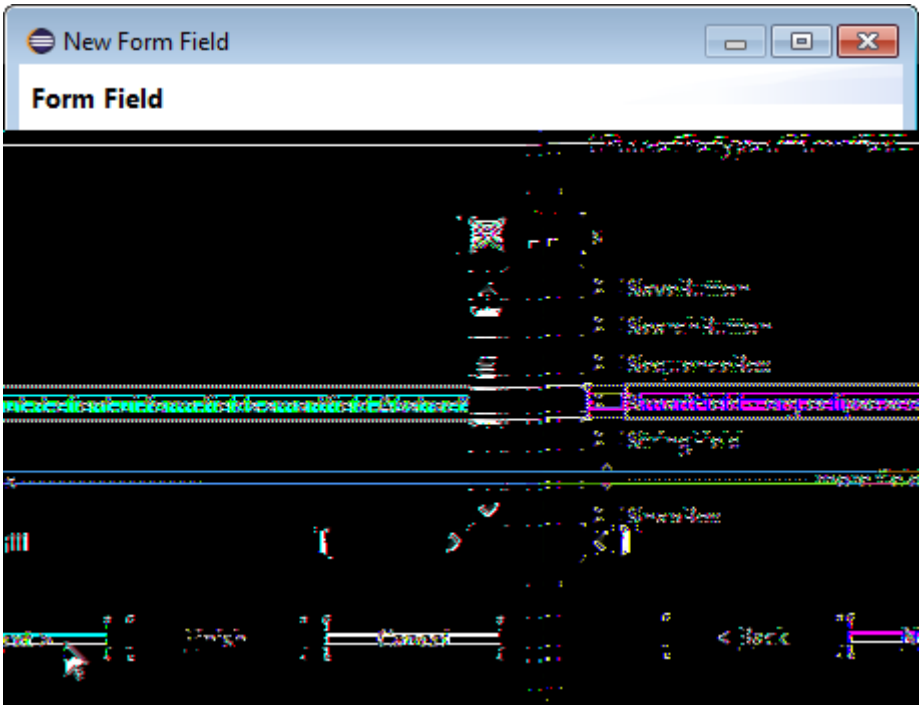


Figure 43. Add a smart field to the person form.



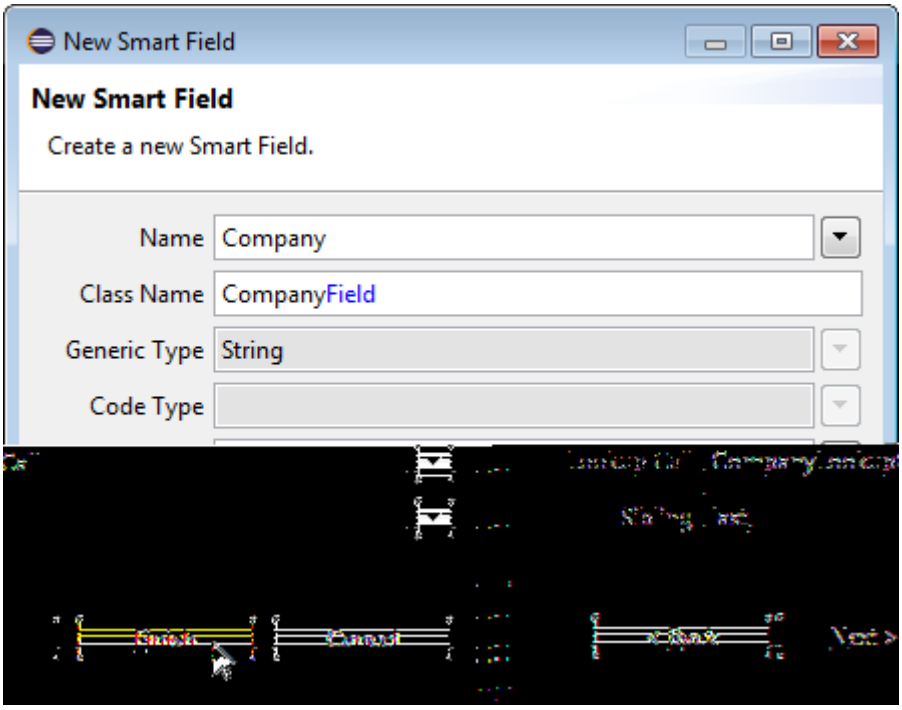


Figure 44. Create the company smart field for the person form. In the second wizard step shown on the right side, first remove the content in the Generic Type field and then select the company lookup call into the corresponding field.

We are now ready to add the company smart field to the person form. To start the *New Form Field* wizard we use the context menu on the DetailBox of the person form as shown in Figure 000. In the first wizard step, we chose the *SmartField* entry as the field type and click the **[Next]** button. Then, we enter ÒCompanyÓ into the *Name* field as shown on the right hand side of Figure 000. Make sure that you select the *String* entry in the *Generic Type* field as we are using string values to identify companies in the ÒMy ContactsÓ application. And in the *LookupCall* field, we can now select the *CompanyLookupCall* that we have just created before. Finally, the position of the new company smart field can be set in the *Sibling* field before the location field before the wizard is closed with the **[Finish]** button.

Listing 47. The smart field CompanyField of the person form and its wiring with the company lookup call.

```
TODO include was {codedir}/oneDayTutorial/org.eclipsescout.contacts.client/src/org
/eclipsescout/contacts/client/ui/forms/PersonForm.java[lines=324..337]
```

The implementation of the company smart field created by the Scout SDK wizard is provided in Listing CompanyField. A look at the implementation of the CompanyField class shows the wiring with the company lookup service.

5.3. Adding the Map Form

We now want to add the ÒMapÓ form shown in the front of [img-my_contacts_swt]. The purpose of this form is to show a map corresponding to the address entered into the *location* field of the person form using the Google Maps Image API. [18: The Google Maps Image API can be used to fetch static map images: https://developers.google.com/maps/documentation/imageapis/.]. This implies

that map images will only be shown in the map form if the entered address can be parsed by the Google Maps Image API.

To create the maps form we start the *New Form* wizard and enter the new translated text ÒMapÓ into the *Name* field and Òui.formsÓ into the *Sub Package* field of the first wizard step. Then, we click the **[Next]** button to configure the artefacts to be created by the wizard. For the map form we can use the configuration as shown on the right hand side of Figure 000 with the difference that we do not need the cancel button. Having deselected all artefacts except for the ok button and the modify handler, the wizard can be closed with the **[Finish]** button.

After the form creation wizard has been closed, we can add an ÒAddressÓ variable to the form by starting the *New Property Bean* wizard on the *Variables* node of the newly created map form. In the property bean wizard, enter ÒAddressÓ into the *Name* field and set the *Bean type* field to *String*.

As the next step, the map image field is added to the from. For this, start the *New Form Field* wizard directly on the formÕs *MainBox* node. In the first form field wizard step, select *ImageField* as the field type and click on the **[Next]** button. Now enter ÒMapÓ into the *Class Name* field and close the second wizard step with the **[Finish]** button. To set the properties of the new map field, select the *MapField* node below the main box node of the map form. In the MapField's Scout Object Properties untick the *Label Visible* property and add an execInitField method by clicking on the green plus icon next to this operation. The configuration of the map field can then be completed in section *Advanced Properties*. Here, we set the *Grid H* property to 6 and update the *Width in Pixel* property and the *Height in Pixel* property to a value of 400 each.

Listing 48. In the execInitField method of the map form the image content is fetched from the Google Maps API.

```
TODO include was {codedir}/oneDayTutorial/org.eclipsescout.contacts.client/src/org
/eclipsescout/contacts/client/ui/forms/MapForm.java[lines=115..133]
```

To add the Java code to display the map in the image field, click on the *execInitField* link in the Scout Object Properties of the map field. According to the implementation provided in Listing execInitField, an URL for the Maps Image API is first constructed. This url also contains the content of the map formÕs address variable and the configured dimension of the map field. The map picture returned by the Google API is then read using IOUtility.getContent and directly fed into the image fields `setImage` method.

The last step involving the implementation of the map form feature is its integration into the person form. As visible on the lower left part of the person form shown in [img-my_contacts_swt], a *Show Map ...* link is available. We now add such a link to the person form including the necessary wiring for opening the newly created map form. For this, navigate to the person form in the Scout SDK and click on its *MainBox* node below. Then, open the context menu on the *MainBox* node and start the *New Form Field* wizard. In the first wizard step, select the *LinkButton* from the available field types and click the **[Next]** button to load the second wizard step. Here, just enter the new translated text ÒShow Map ÉÓ into the *Name* field and close the wizard with the **[Finish]** button.

Listing 49. The implementation of the "Show Map ..." link button. %In method `execClickAction` a new map form is first created, the the address variable is set with the content of the person's location and the form is opened with `startModify`.

```
TODO include was {codedir}/oneDayTutorial/org.eclipsescout.contacts.client/src/org
/eclipsescout/contacts/client/ui/forms/PersonForm.java[lines=347..362]
```

To add the necessary wiring code to the link button double click the *ShowMapButton* node in the Scout Explorer and implement its execClickAction method. As shown in Listing ShowMapButton, we only need to create a new map form in the click action, set its address variable and open the form with form.startModify.

5.4. Creating the Company Form

Creating the company form and the necessary backend services is not described here. Instead, this task is left as an exercise to the reader and in the text below some minimal guidelines are provided.

To create the company form, start with the *New Form* wizard as in the case of the person form. This will then create all necessary artefacts including the forms, the server service, and the form data for the communication between the client and the server. And don0t forget to add a companyId variable to the company form. To decide on the fields that need to be on the company form you may check the setup of the database schema provided in [lst-mycontacts.server.services.installdb.company]. If in doubt about what to do, please refer to the procedure used to create the person form.

In case you get lost completely, you may download the 0My Contacts0 application from this books Github repository as described in the Scout wiki. [19: Download and installation of the 0My Contacts0 application: http://wiki.eclipse.org/Scout/Book/4.0#Download_and_Run_the_Scout_Sample_Applications.].

Appendix A: Licence and Copyright

This appendix first provides a summary of the Creative Commons (CC-BY) licence used for this book. The licence is followed by the complete list of the contributing individuals, and the full licence text.

A.1. Licence Summary

This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit https://creativecommons.org/licenses/by/3.0/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

A summary of the license is given below, followed by the full legal text.

You are free:

- **to Share** ---to copy, distribute and transmit the work
- **to Remix**---to adapt the work
- to make commercial use of the work

Under the following conditions:

Attribution ---You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

With the understanding that:

Waiver ---Any of the above conditions can be waived if you get permission from the copyright holder.

Public Domain ---Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.

Other Rights ---In no way are any of the following rights affected by the license:

- Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
- The author's moral rights;
- Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

Notice ---For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to https://creativecommons.org/licenses/by/3.0/.

A.2. Contributing Individuals

In the text below, all contributing individuals are listed in alphabetical order by name. For

contributions in the form of GitHub pull requests, the name should match the one provided in the corresponding public profile.

Bresson Jeremie, Fihlon Marcus, Nick Matthias, Schroeder Alex, Zimmermann Matthias

A.3. Full Licence Text

The full licence text is available online at http://creativecommons.org/licenses/by/3.0/legalcode

Appendix B: Scout Installation

B.1. Overview

This chapter walks you through the installation of Eclipse Scout. The installation description (as well as the rest of this book) is written and tested for Eclipse Scout 6.0 which is delivered as integral part of the Eclipse Neon release train, 2016. Detailed information regarding the scheduling of this release train is provided in the Eclipse wiki. [20: Luna release plan: http://wiki.eclipse.org/Neon/Simultaneous_Release_Plan].

We assume that you have not installed any software relevant for the content of this book. This is why the Scout installation chapter starts with the installation of the Java Development Kit (JDK). Consequently, you will have to skip some of the sections depending on your existing setup.

In the text below, installation routines are described separately for Windows, Mac, and Linux.

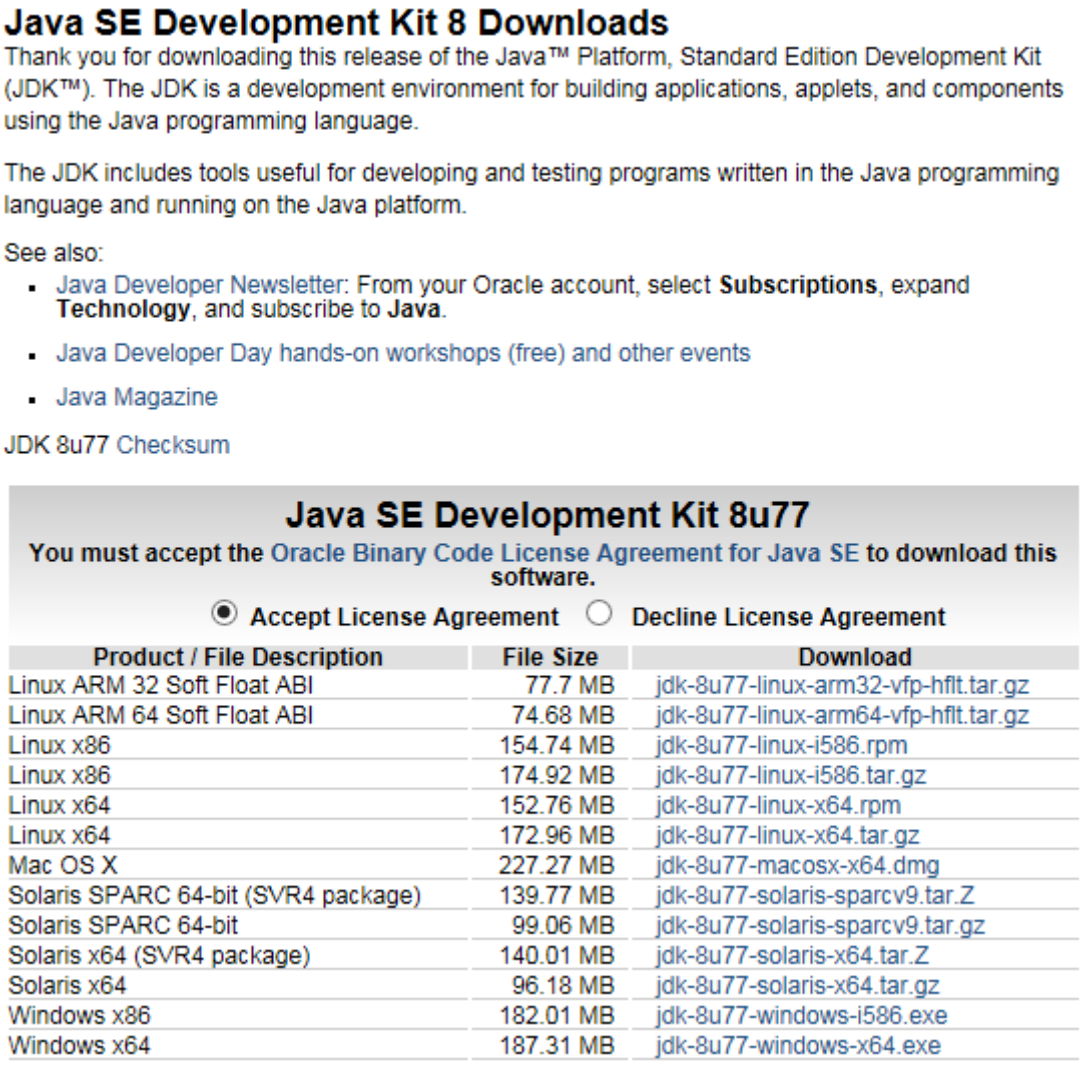
B.2. Download and Install a JDK

The first step to install Scout is to have an existing and working installation of a JDK version 8.

Currently, we recommend to install the Oracle JDK 8 together with Scout. Although, using OpenJDK with Scout should work too. To successfully install the JDK you need to have at least local admin rights. You also need to know your hardware architecture in order to download the correct JDK installer.

For Windows, the steps necessary to determine your hardware architecture are described on Microsoft's support site. [21: Windows 32/64-bit installation: http://support.microsoft.com/kb/827218]. For Linux several ways to determine if your os is running with 32 or with 64 bits can be found on the web. [22: Linux 32/64-bit installation example page: http://mylinuxbook.com/5-ways-to-check-if-linux-is-32-bit-or-64-bit/] For Mac this step is simple, as only a 64 bit package is provided on JDK the download page.

Once you know your hardware architecture, go to Oracle's official download site. [23: Official JDK 8 download: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html] and accept the licence agreement by clicking on the appropriate radio button. Then, select the *Windows x64* package if you are running 64-bit Windows as shown in Figure 000. If you are running 32-bit Windows, go for the *Windows x86* package. It is also recommended to download the *Java SE 8 Documentation*. The Java API documentation package is available from the official download site. [24: Java API documentation download: http://www.oracle.com/technetwork/java/javase/downloads/index.html], located further down under section *Additional Resources*.



Java SE Development Kit 8 Downloads

Thank you for downloading this release of the Java™ Platform, Standard Edition Development Kit (JDK™). The JDK is a development environment for building applications, applets, and components using the Java programming language.

The JDK includes tools useful for developing and testing programs written in the Java programming language and running on the Java platform.

See also:

- Java Developer Newsletter: From your Oracle account, select **Subscriptions**, expand **Technology**, and subscribe to **Java**.
- Java Developer Day hands-on workshops (free) and other events
- Java Magazine

JDK 8u77 Checksum

Java SE Development Kit 8u77

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

◉ Accept License Agreement ○ Decline License Agreement

Product / File Description	File Size	Download
Linux ARM 32 Soft Float ABI	77.7 MB	jdk-8u77-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Soft Float ABI	74.68 MB	jdk-8u77-linux-arm64-vfp-hflt.tar.gz
Linux x86	154.74 MB	jdk-8u77-linux-i586.rpm
Linux x86	174.92 MB	jdk-8u77-linux-i586.tar.gz
Linux x64	152.76 MB	jdk-8u77-linux-x64.rpm
Linux x64	172.96 MB	jdk-8u77-linux-x64.tar.gz
Mac OS X	227.27 MB	jdk-8u77-macosx-x64.dmg
Solaris SPARC 64-bit (SVR4 package)	139.77 MB	jdk-8u77-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	99.06 MB	jdk-8u77-solaris-sparcv9.tar.gz
Solaris x64 (SVR4 package)	140.01 MB	jdk-8u77-solaris-x64.tar.Z
Solaris x64	96.18 MB	jdk-8u77-solaris-x64.tar.gz
Windows x86	182.01 MB	jdk-8u77-windows-i586.exe
Windows x64	187.31 MB	jdk-8u77-windows-x64.exe

Figure 45. Installer download for Oracle JDK 8.

Once you have successfully downloaded the JDK installer, follow the Windows installation guide. [25: Install the JDK on Windows: http://docs.oracle.com/javase/7/docs/webnotes/install/windows/jdk-installation-windows.html#Run]. To verify the installation you might want to go through this Java ÒHello World!Ó tutorial. [26: Windows Java ÒHello World!Ó: http://docs.oracle.com/javase/tutorial/getStarted/cupojava/win32.html].

Installation instructions for Linux. [27: Install the JDK on Linux: http://docs.oracle.com/javase/7/docs/webnotes/install/linux/linux-jdk.html] and Mac. [28: Install the JDK on Mac: http://docs.oracle.com/javase/7/docs/webnotes/install/mac/mac-jdk.html.] are also available from Oracle.

B.3. Download and Install Scout

Before you can install Scout make sure that you have a working Java Development Kit (JDK) installation version 8. To download the Eclipse Scout package visit the official Eclipse download page as shown in Figure 000.

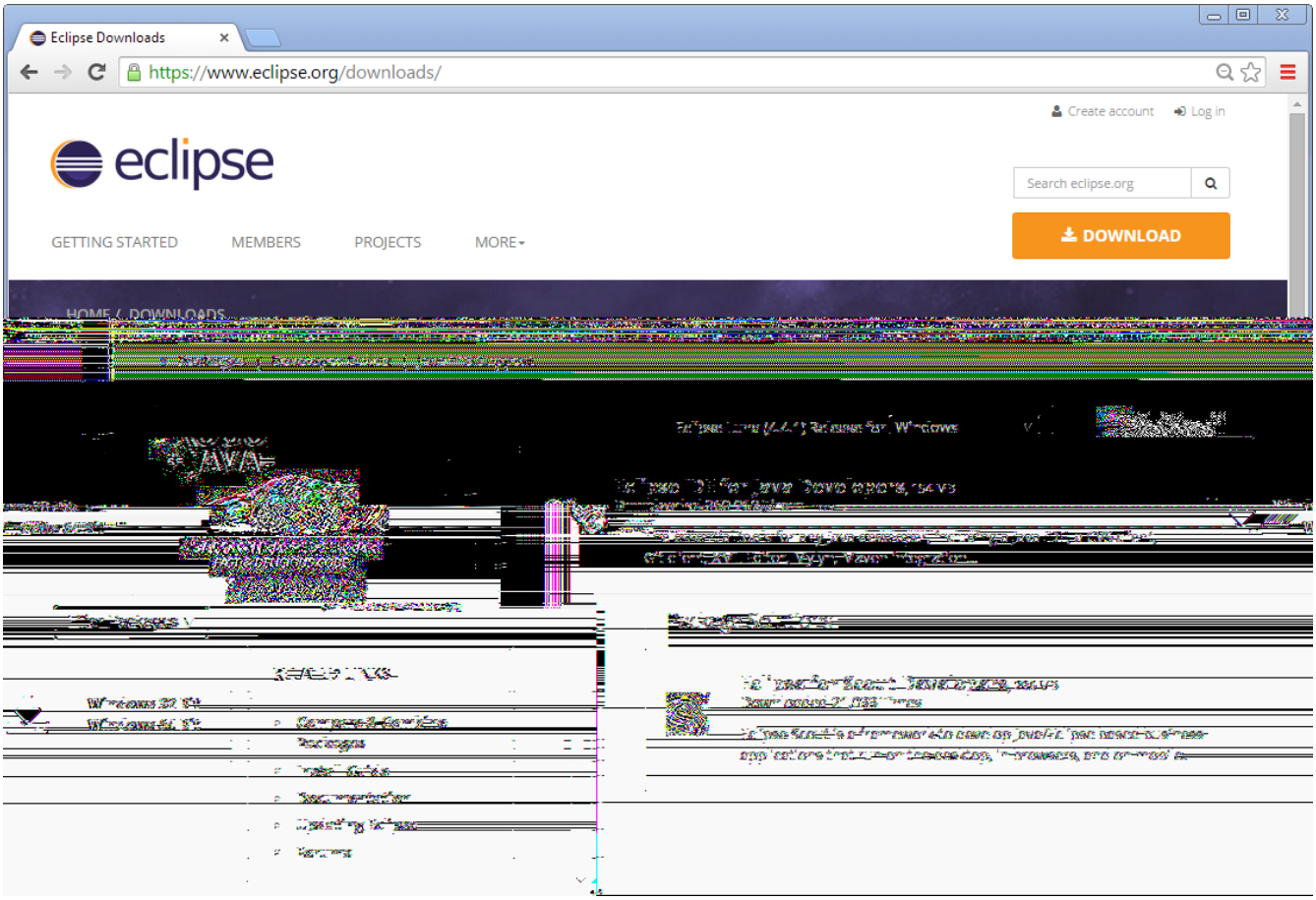


Figure 46. The Eclipse download page. The platform filter is set to Windows and the available Packages are filtered for Scout.

If the download page shows the wrong platform, manually select the correct platform in the dropdown list. As shown in Figure 000, the Scout package is available as a 32 bit and a 46 bit package. Make sure to pick the package that matches your JDK installation. You can check your installation on the command line as follows.

```
console-prompt>java -version
java version "1.8.0_77"
Java(TM) SE Runtime Environment (build 1.8.0_77-b28)
Java HotSpot(TM) 64-Bit Server VM (build 25.77-b28, mixed mode)
```

If the output explicitly mentions the 64 bit installation as shown above, you have a 64 bit installation. Otherwise, you have a 32 bit JDK installed. Now you can select the correct Scout package from the Eclipse download site. After the package selection, confirm the suggested download mirror as shown in Figure 000.

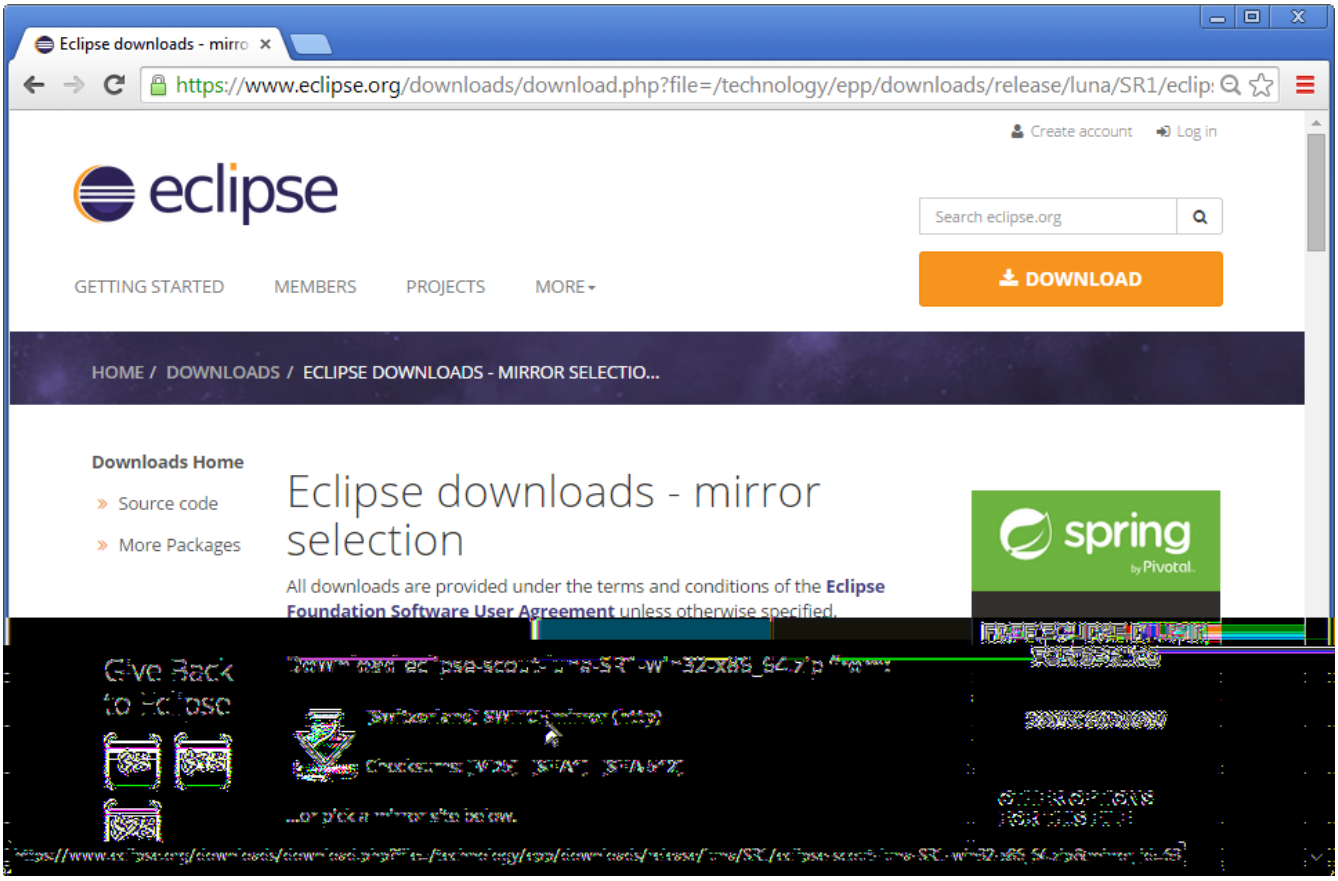


Figure 47. Downloading the Scout package from a mirror.

As the Scout package is a simple ZIP (or tar.gz) file, you may unpack its content to a folder of your choice. Inside the eclipse sub-folder, you will then find the Eclipse executable file, such as the `eclipse.exe` file on a Windows plattform. Starting the Eclipse executable brings up the workspace launcher as shown in Figure 000.

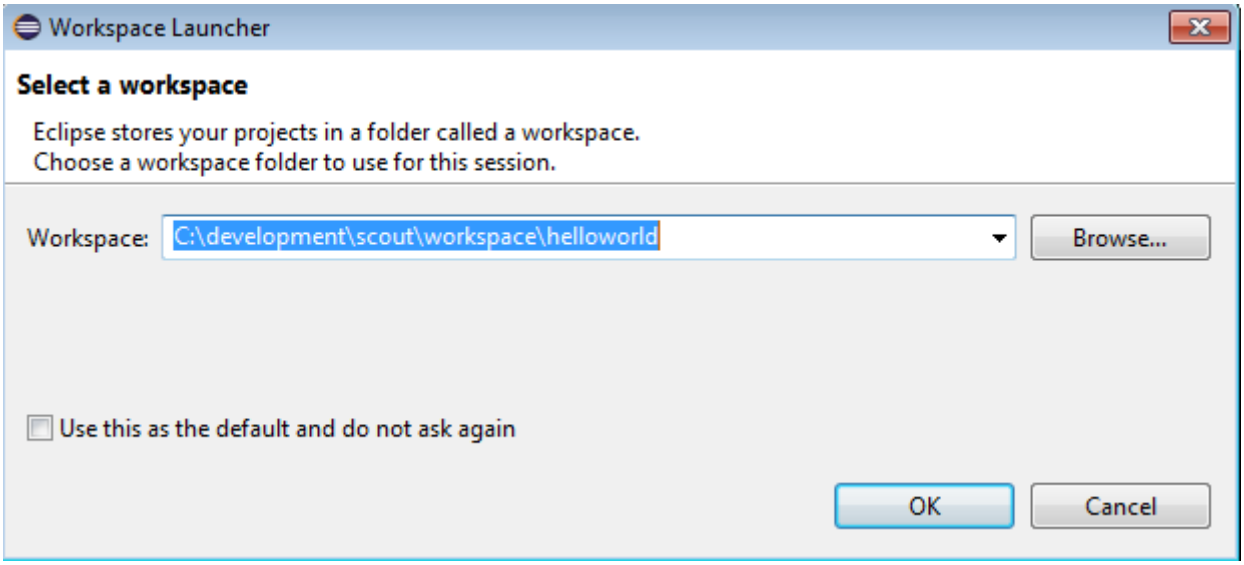


Figure 48. Starting the Eclipse Scout package and selecting an empty workspace.

Into the *Workspace* field you enter an empty target directory for your first Scout project. After clicking the **[Ok]** button, the Eclipse IDE creates any directories that do not yet exist and opens the specified workspace. When opening a new workspace for the first time, Eclipse then displays the welcome screen shown in Figure 000.

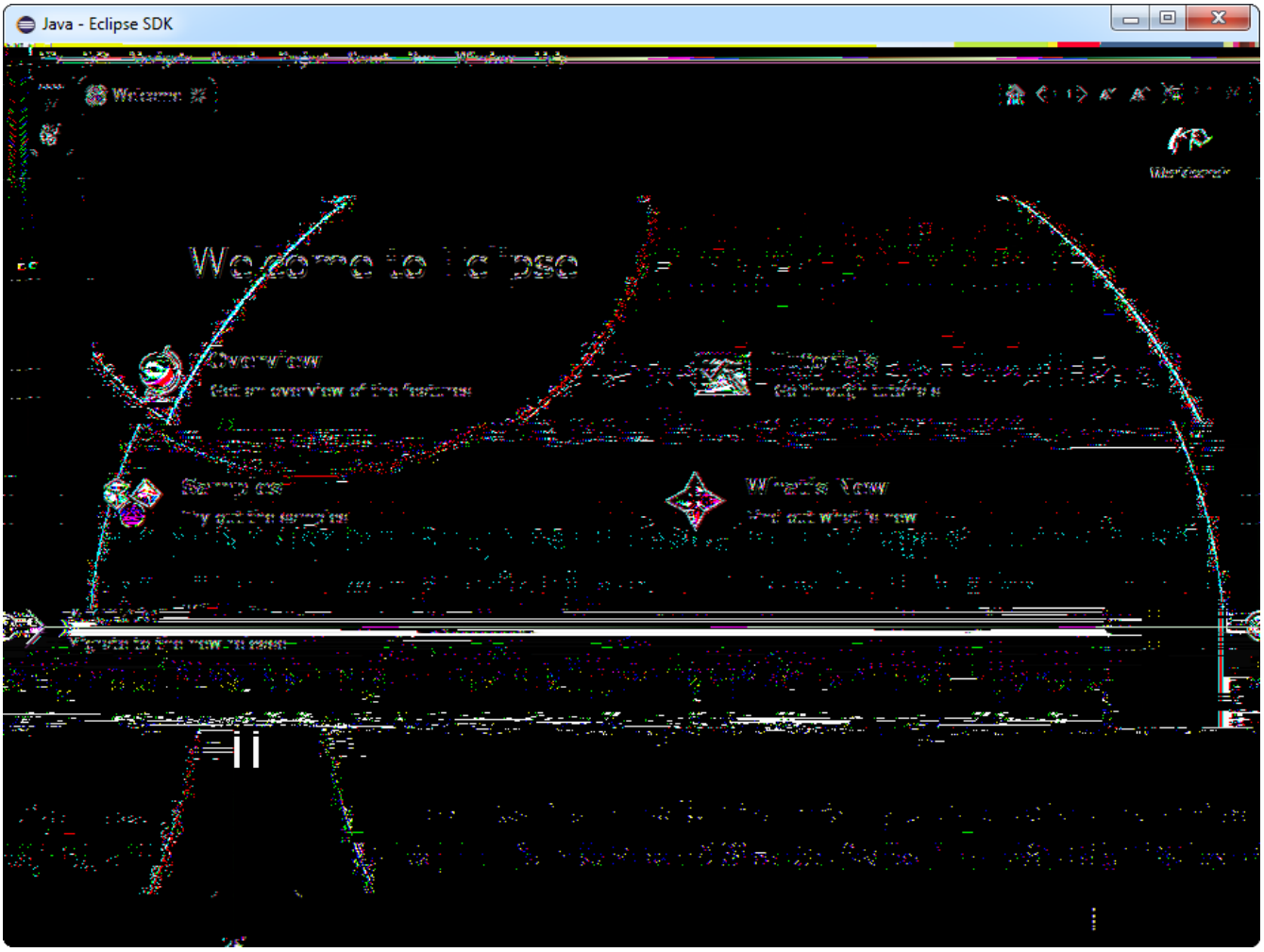


Figure 49. Eclipse Scout welcome screen.

To close the welcome page and open the Scout perspective in the Eclipse IDE click on the *Workbench* icon. As a result the empty Java perspective is displayed according to Figure 000.

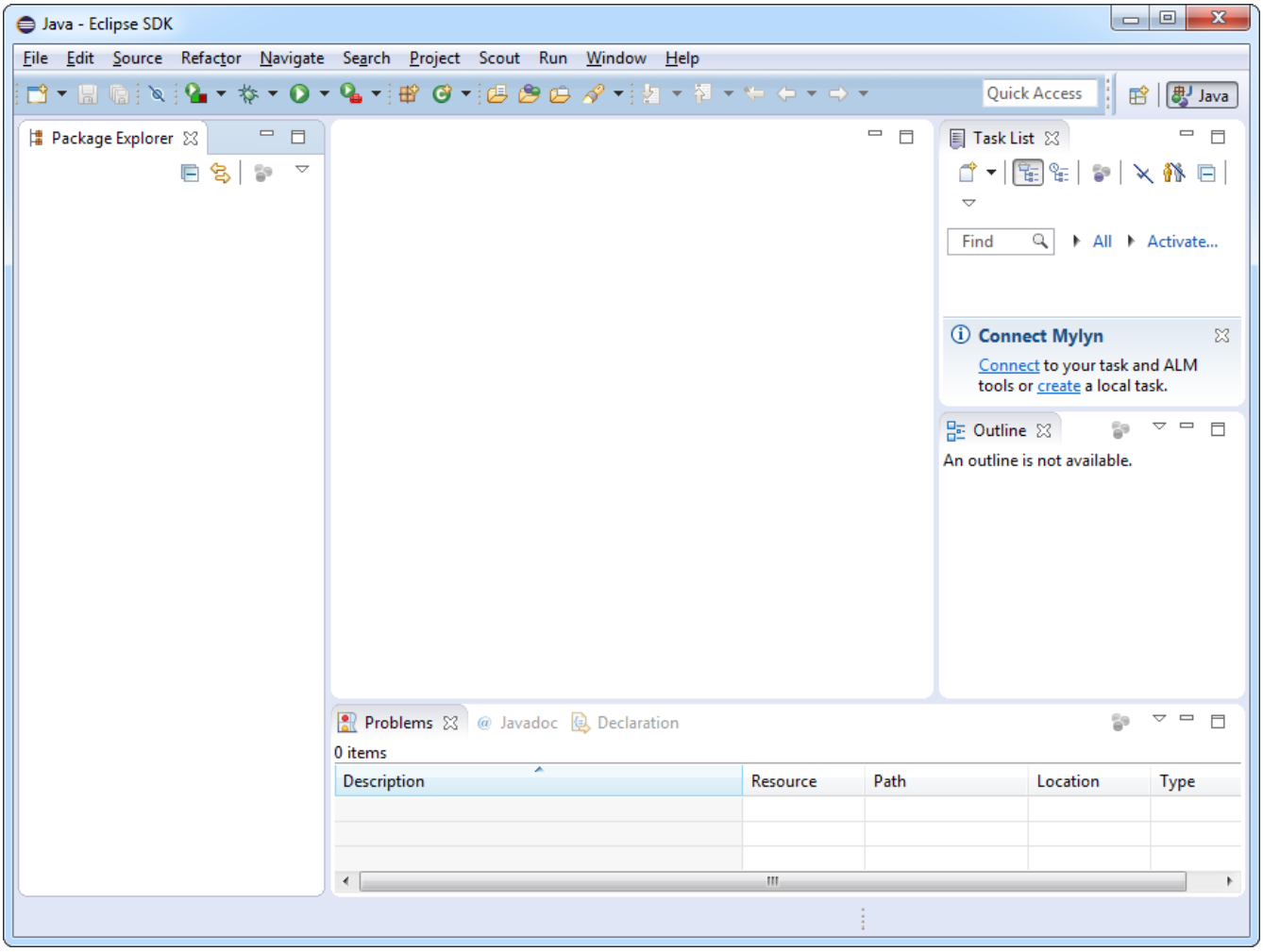


Figure 50. Eclipse Scout started in the Scout SDK perspective.

Congratulations, you just have successfully completed the Eclipse Scout installation!

If you have only installed a single JDK you will not need to change the default `eclipse.ini` file of your Eclipse installation. In case you have installed multiple JDKs coming with their individual Java Runtime Environments (JREs), you might want to explicitly specifiy which JRE to use. Open the file `eclipse.ini` in a editor of your choice and insert the following two lines at the top of the file:

```
-vm
C:\java\jdk1.8.0_77_x64\jre\bin\server\jvm.dll
```

where the second line specifies the exact path to the JRE to be used to start your Eclipse Scout installation.

If you have explicitly specified the JRE to be used you verify this in the running Eclipse installation. Fist, select the **Help | About Eclipse** menu to open the about dialog. Then, click on the **[Installation Details]** button and switch to the *Configuration* tab. In the long list of system properties you will find lines similar to the ones shown below.

```
*** Date: Donnerstag, 19. Juni 2014 10:37:17 Normalzeit

*** Platform Details:

*** System properties:
...
-vm
C:\java\jdk1.8.0_77_x64\jre\bin\server\jvm.dll
...
sun.java.command=... vm C:\java\jdk1.8.0_77_x64\jre\bin\server\jvm.dll -vmargs ...
```

You have now successfully completed the Eclipse Scout installation on your Windows environment. With this running Scout installation you may skip the following section on how to add Scout to an existing Eclipse installation.

B.4. Add Scout to your Eclipse Installation

This section describes the installation of Scout into an existing Eclipse installation. As the audience of this section is assumed to be familiar with Eclipse, we do not describe how you got your Eclipse installation in the first place. For the provided screenshots we start from the popular package *Eclipse IDE for Java EE Developers*.

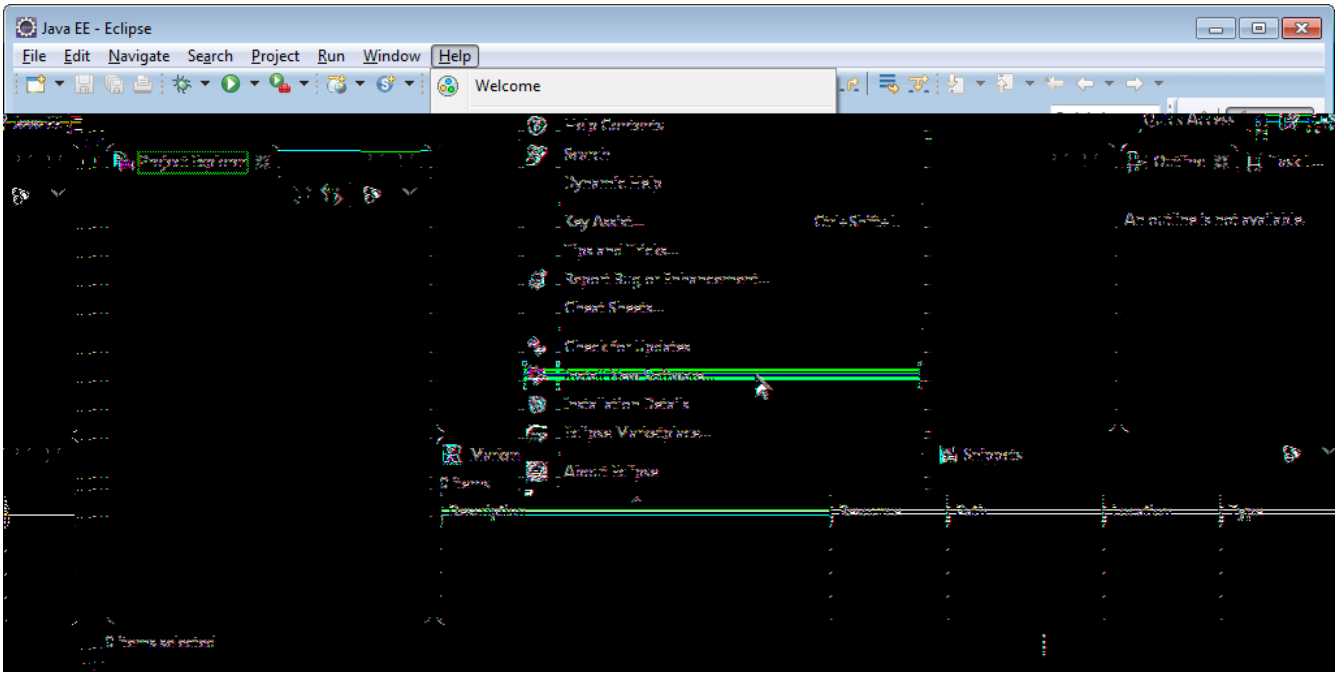


Figure 51. Eclipse menu to install additional software

To add Scout to your existing Eclipse installation, you need to start Eclipse. Then select the **Help | Install New Software...** menu as shown in Figure 000 to open the install dialog.

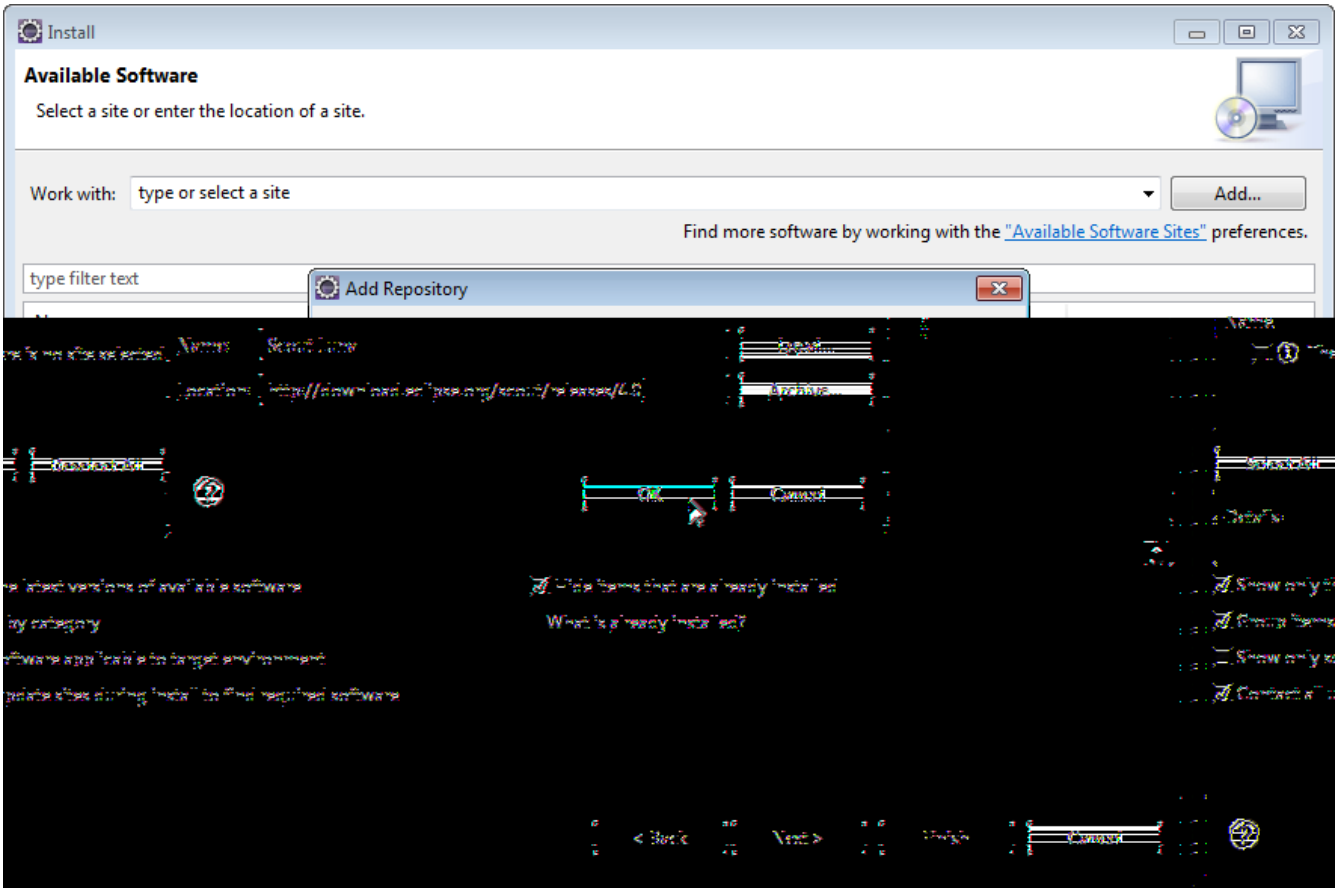


Figure 52. Add the current Scout repository

In the install dialog, click on the **[Add...]** button to enter the link to the Scout repository. This opens the popup dialog *Add Repository* As shown in Figure 000, you may use ÒScout LunaÓ for the *Name* field. For the *Location* field enter the Scout release repository as specified below. http://download.eclipse.org/scout/releases/4.0.

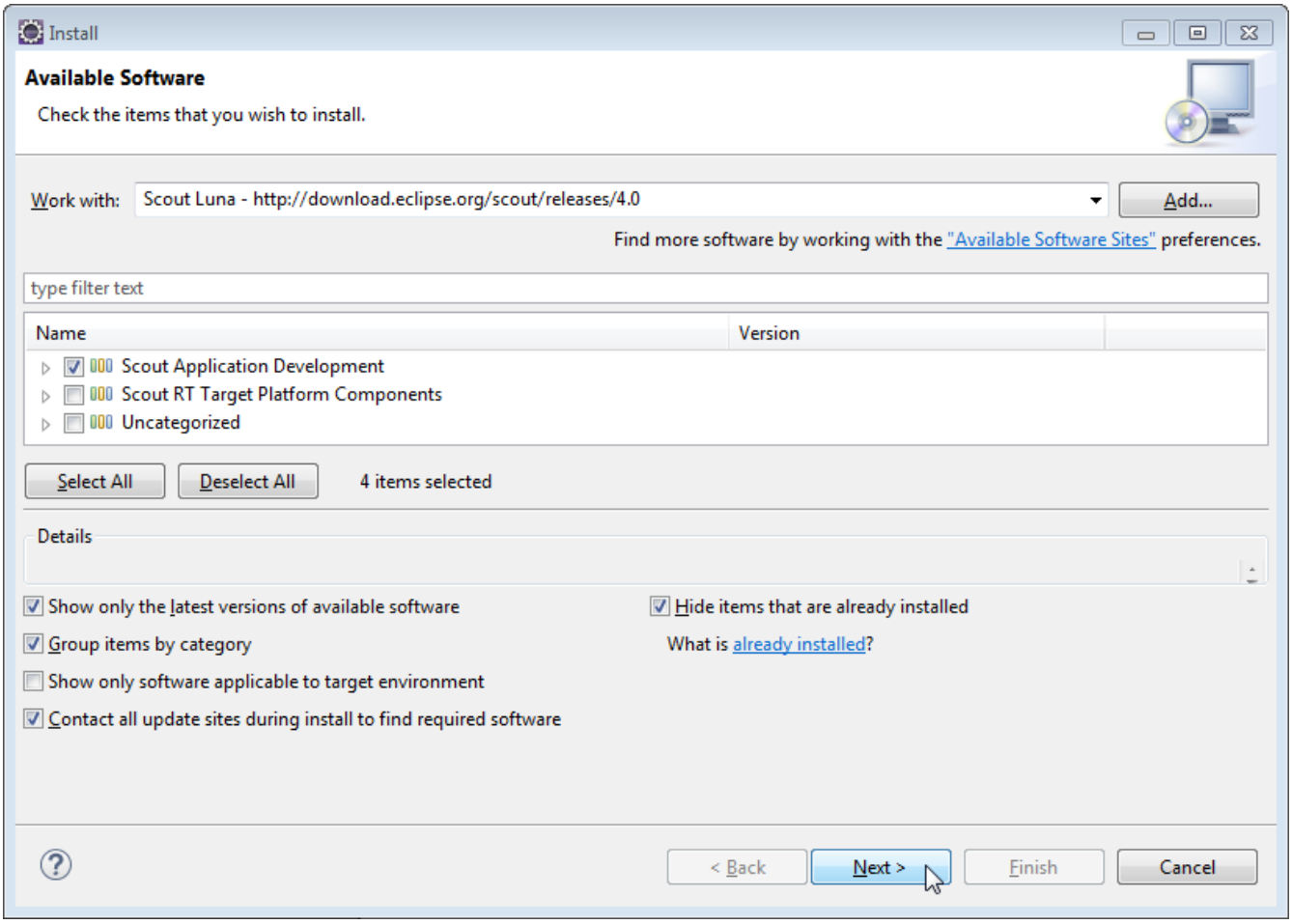


Figure 53. Select the Scout features to add to the Eclipse installation

After the Eclipse IDE has connected to the Scout repository, select the Scout feature *Scout Application Development* as shown in Figure 000. Then, move through the installation with the **[Next]** button. On the last installation step, accept the presented EPL terms by clicking on the appropriate radio button. To complete the installation, click the **[Finish]** button and accept the request for a restart of Eclipse. After the restart of the Eclipse IDE, you may add the Scout perspective using the **Window | Open Perspective | Other ...** menu and selecting the Scout perspective from the presented list. Clicking on the Scout perspective button should then result in a state very similar to Figure 000.

B.5. Verifying the Installation

After you can start your Eclipse Scout package you need to verify that Scout is working as intended. The simplest way to verify your Scout installation is to create a ÒHello WorldÓ Scout project and run the corresponding Scout application as described in Chapter ÒHello WorldÓ Tutorial.

Appendix C: Apache Tomcat Installation

Apache Tomcat is an open source web server that is a widely used implementation of the Java Servlet Specification. Specifically, Tomcat works very well to run Scout applications. In case you are interested in getting some general context around Tomcat you could start with the Wikipedia article. [29: Apache Tomcat Wikipedia: http://en.wikipedia.org/wiki/Apache_Tomcat.]. Then get introduced to its core component ÒTomcat CatalinaÓ. [30: Mulesoft's introduction to Tomcat Catalina: http://www.mulesoft.com/tomcat-catalina.] before you switch to the official Tomcat homepage. [31: Apache Tomcat Homepage: http://tomcat.apache.org/].

This section is not really a step by step download and installation guide. Rather, it points you to the proper places for downloading and installing Tomcat. We recommend to work with Tomcat version 8.0. Start your download from the official download site. [32: Tomcat 8 Downloads: http://tomcat.apache.org/download-80.cgi].

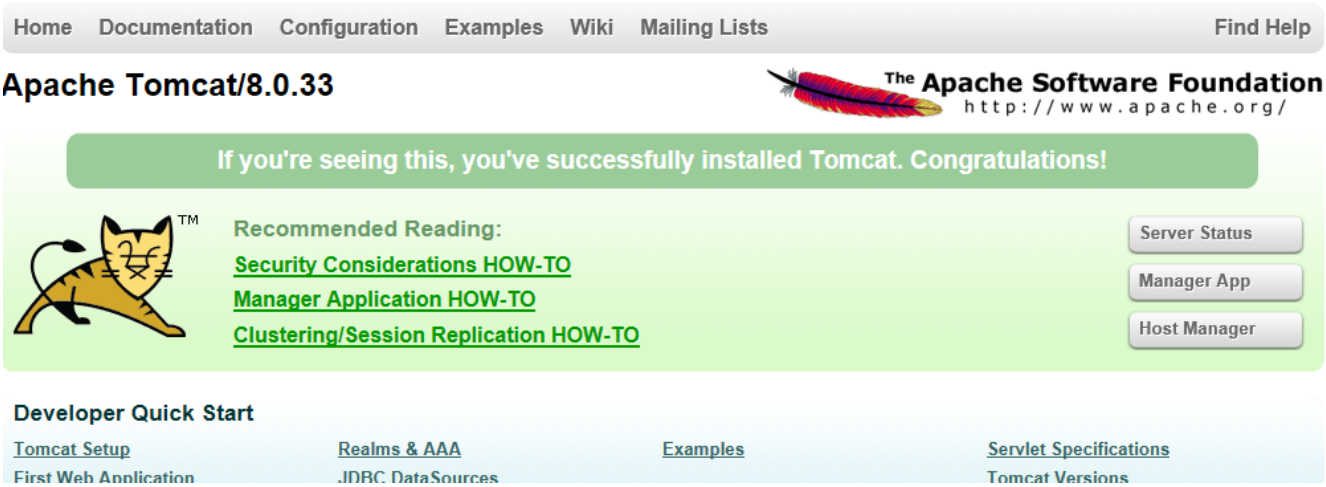


Figure 54. A successful Tomcat 7 installation.

Once you have downloaded and installed Tomcat 8 (see the sections below for plattform specific guidelines) you can start the corresponding service or deamon. To verify that Tomcat is actually running open a web browser of your choice and type http://localhost:8080 into the address bar. You should then see a confirmation of the successful installation according to Figure 000.

C.1. Platform Specific Instructions

According to the Tomcat setup installation for Windows. [33: Tomcat Windows setup: http://tomcat.apache.org/tomcat-8.0-doc/setup.html#Windows] download the package Ò32-bit/64-bit Windows Service InstallerÓ from the Tomcat 8 download site. Then, start the installer and accept the proposed default settings.

For installing Tomcat on OS X systems download the Ò tar.gzÓ package from the Tomcat 8 download site. Then, follow the installation guide. [34: Installing Tomcat on OS X: http://wolfpaulus.com/journal/mac/tomcat8] provided by Wolf Paulus.

For Linux systems download the Ò tar.gzÓ package from the Tomcat 8 download site. Then, follow the description of the Unix setup. [35: Tomcat Linux setup: http://tomcat.apache.org/tomcat-8.0-doc/setup.html#Unix_daemon] to run Tomcat as a deamon. If you use Ubuntu, you may want to

follow the tutorial. [36: Apache Tomcat Tutorial: http://www.vogella.com/articles/ApacheTomcat/article.html] for downloading and installing Tomcat provided by Lars Vogel.

C.2. Directories and Files

Tomcat's installation directory follows the same organisation on all platforms. Here, we will only introduce the most important aspects of the Tomcat installation for the purpose of this book.

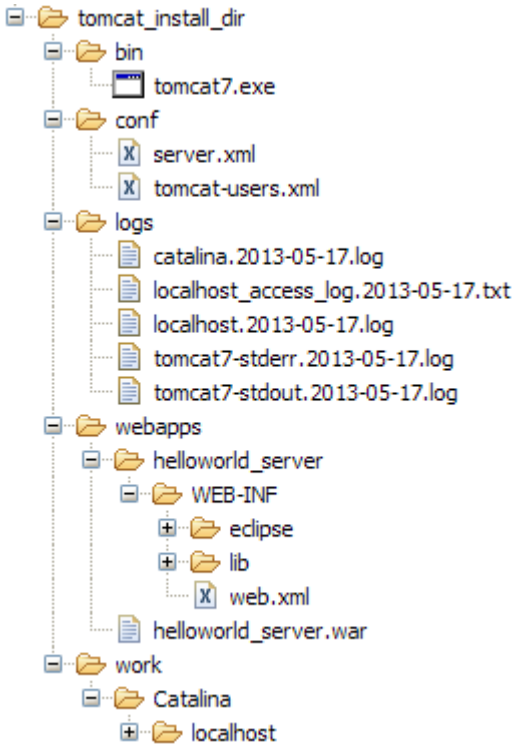


Figure 55. The organisation of a Tomcat installation including specific files of interest. As an example, the "Hello World" server application is contained in subdirectory `webapps`.

Note that some folders and many files of a Tomcat installation are not represented in Figure 000. We just want to provide a basic understanding of the most important parts to operate the web server in the context of this book. In the bin folder, the executable programs are contained, including scripts to start and stop the Tomcat instance.

The conf folder contains a set of XML and property configuration file. The file server.xml represents Tomcat's main configuration file. It is used to configure general web server aspects such as the port number of its connectors for the client server communication. For the default setup, port number 8080 is used for the communication between clients applications and the web server. The file tomcat-users.xml contains a database of users, passwords and associated roles.

Folder logs contains various logfiles of Tomcat itself as well as host and web application log files. XXX need to provide more on what is where (especially application logs and exact setup to generate log entries from scout apps).

The folder needed for deploying web applications into a Tomcat instance is called webapps. It can be used as the target for copying WAR files into the web server. The installation of the WAR file then extracts its content into the corresponding directory structure as shown in Figure 000 in the

case of the file helloworld_server.war.

Finally, folder work contains Tomcat's runtime "cache" for the deployed web applications. It is organized according to the hierarchy of the engine (Catalina), the host (localhost), and the web application (helloworld_server).

C.3. The Tomcat Manager Application

Tomcat comes with the pre installed "Manager App". This application is useful to manage web applications and perform tasks such as deploying a web application from a WAR file, or starting and stopping installed web applications. A comprehensive documentation for the "Manager App" can be found under the Tomcat homepage. [37: The Tomcat Manager Application: http://tomcat.apache.org/tomcat-8.0-doc/manager-howto.html.]. Here we only show how to start this application from the hompage of a running Tomcat installation.

To access this application you can switch to the "Manager App" with a click on the corresponding button on the right hand side. The button can be found on the right hand side of Figure 000. Before you are allowed to start this application, you need to provide username and password credentials of a user associated with Tomcats's manager-gui role.

```
<tomcat-users>
  <!--
  NOTE: By default, no user is included in the "manager-gui" role required
  to operate the "/manager/html" web application. If you wish to use it
  you must define such a user - the username and password are arbitrary.
  -->
  <user name="admin" password="s3cret" roles="manager-gui"/>
</tomcat-users>
```

To get at user names and passwords you can open file tomcat-users.xml located in Tomcat's conf directory. In this file the active users with their passwords and associated roles are stored. See Listing [lst-tomcat.users] for an example. From the content of this file, we see that user admin has password s3cret and also posesses the necessary role manager-gui to access the "Manager App". If file tomcat-users.xml does not contain any user with this role, you can simply add new user with this role to the existing users. Alternatively, you also can add the necessary role to an existing user. Just append a comma to the existing right(s) followed by the string manager-gui. Note that you will need to restart your Tomcat application after adapting the content of file tomcat-users.xml.

With working credentials you can now start the "Manager App" as described the "Hello World" tutorial in Section Deploy to Tomcat.

Do you want to improve this document? Please edit this page on GitHub.