Eclipse Scout

an Introduction

Matthias Zimmermann

Version 5.0.0-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.

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.

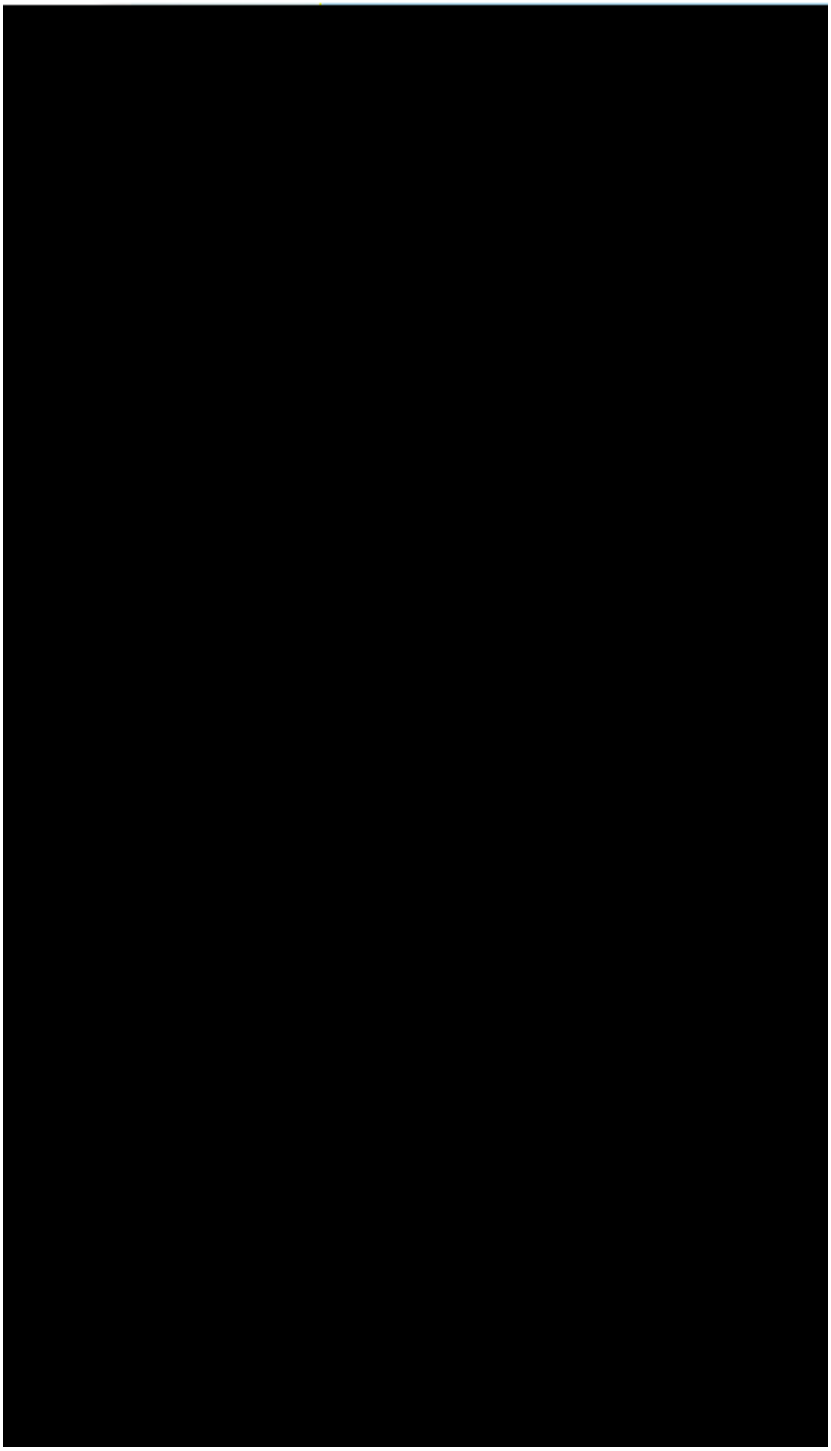


Figure 3. The same Scout enterprise application running on a mobile device.

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, Scoutʼs 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 Scoutʼs 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 Scoutʼs 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 Scoutʼs 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 applicationôs 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 any significant understanding of the Java Enterprise Edition (Java EE). [5: Java Enterprise Edition: http://en.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition] and the Eclipse Platform. [6: Eclipse Platform: http://wiki.eclipse.org/Platform].

Of course, having prior experience in client server programming with Java is helpful. And having used the Eclipse IDE for Java development before --- please do not mistake the IDE with the Eclipse platform. [7: By reading through the book you will learn that there is much more to the Eclipse platform than just the IDE] is certainly of benefit.

The ÒbadÓ 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). [8: 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. [9: Level I Exam: docs.oracle.com/javase/tutorial/extra/certification/javase-7-programmer1.html] and most of the material required for Level II. [10: 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. [11: 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. [12: 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 know tons of both Java and Eclipse

This means that you are one of these software wizards that get easily bored. You prefer to get a quick impression before deciding to dig deeper and hate going through lengthy descriptions. In that case let us assume that you are prepared to spend two hours to grasp the scope of Eclipse Scout and get an impression of its strengths and limitations. The list below suggests a sequence of sections to digest including a brief motivation for each one.

¥ Chapter ÒHello WorldÓ Tutorial `"Hello World"` Tutorial. Download and installation of the Scout

package should take less than 30 minutes, going through the `"Hello World"` takes another 15 minutes.

- Section Walking through the Initial Application ÒWalking through the Initial ApplicationÓ Read about some key elements used in every Scout client application including integration of server services and data binding.
- Chapter [cha-tooling] ÒScout ToolingÓ. Browse through the tooling chapter to get an impression on the tooling provided with Scout. Make sure you understand that the Scout SDK is supporting the developer without restricting the developer.
- Chapter [cha-large_example] ÒThe My Contacts ApplicationÓ. Check out the slightly larger demo application. In case you are not yet running out of time, download the demo app as described in the Scout wiki. [13: Download and installation of the ÒMy ContactsÓ application: http://wiki.eclipse.org/Scout/Book/#Download_and_Run_the_Scout_Sample_Applications.].

1.3.3. 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 [sec-my_contacts_guide] 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. [14: To contact the Scout team, use the feedback provided on the Scout homepage: https://eclipse.org/scout.].

2. ÒHello WorldÓ 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. [15: 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. Except for a single line of code in the server part of the ÒHello WorldÓ application, we will only be using the tooling provided by the Scout SDK.

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 ÒHello WorldÓ Background. This tutorial is also available in the Scout wiki. [16: 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. This workspace directory will then hold all the project code for the ÒHello WorldÓ application. Once the Eclipse IDE is running it will show the Scout perspective with the Scout Explorer view and an empty Scout Object Properties view. To create a new Scout project select the New Scout ProjectÉ context menu as shown in Figure New Scout Project Menu.

Figure 6. Create a new Scout project using the Scout SDK perspective.

In the *New Scout Project* wizard enter a name for your Scout project. As we are creating a ÒHello WorldÓ application, use org.eclipsescout.helloworld for the *Project Name* field according to Figure New Scout Project Wizard. Then, click the [!Finish!] button to let the Scout SDK create the initial project code for you.

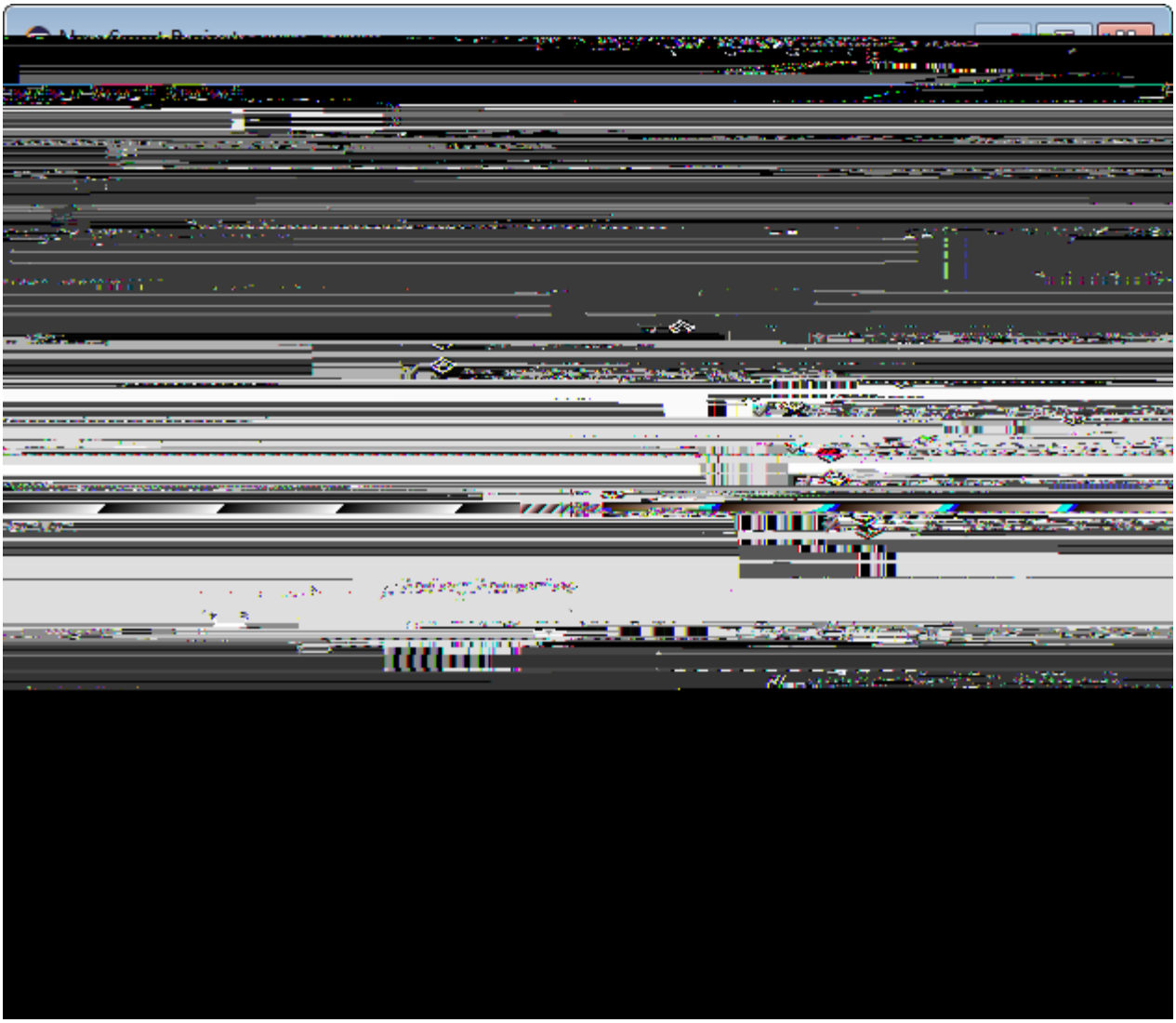


Figure 7. The new Scout project wizard.

Once the initial project code is built, the Scout SDK displays the application model in the *Scout Explorer* as shown in Figure Representation of the Hello World Application. This model is visually presented as a tree structure covering both the client and the server part of the application. The Scout Explorer view on the left hand side displays the top level elements of the complete Scout application. Under the orange node the Scout client components are listed. Components that are needed in both the Scout client and the Scout server are collected under the green node. And the Scout server components are listed below the blue node in the Scout Explorer view.

Figure 8. The Scout SDK showing the tree representation of our ÒHello WorldÓ application in the Scout Explorer. The Scout Object Properties contain the product launchers for the server and the available clients.

2.3. Run the Initial Application

After the initial project creation step we are ready to start the server and the clients of the still empty Scout application. For this, we switch to the Scout Explorer and select the root node *org.eclipse.scout.helloworld*. Selecting the applicationÕs *org.eclipse.scout.helloworld* node in the Scout Explorer displays the product launchers in the *Scout Object Properties*. As we can see in Figure 000, we have product launchers for four different development products.

Server	The Scout server application
RAP	The RAP server application for web and mobile clients
Swing	The Scout Swing desktop client application

SWT	The Scout SWT desktop client application

Figure 9. Starting the web client in the Scout SDK using the provided RAP product launcher. Make sure to start the server before starting any client product.

Each product launcher box provides a link to the corresponding Eclipse product file. [17: Product files define the set of all components that are necessary to build the complete application.], the configuration file. [18: The configuration file config.ini provides parameters that are read at startup of the corresponding program.], as well as three launcher icons to start and stop the corresponding application. The green *Circle* icon starts the product in normal mode. The *Bug* icon just below, starts a product in debug mode. To terminate a running product, the red *Square* icon is provided. Alternatively, you can also stop products by clicking on the same red icon in the console view. This is shown on the right hand side of Figure 000. Client products may also be stopped by closing the clientÕs main window or using the provided File | Exit menu.

Before any of the client products is started, we need to start the server product using the green circle or the bug launcher icon. During startup of the Scout server you should see console output similar to the one shown on the right hand side of Figure 000. Once the server is running, you may start the web client as shown in Figure 000, the Swing client, or the SWT client in the same way. And with a running RAP product, the Scout web client can be opened in a web browser. Just click on the provided *Automatic Device Dispatch* link or open a a browser and manually type the address `http://localhost:8082/web` into the browserÕs navigation bar.

Figure 10. Running the three client applications. Each client displays an empty desktop form. From left to right: The web client, the Swing client, and the SWT client

Having started the Scout server and all client products, the client applications should become visible as shown in Figure 000.

2.4. The User Interface Part

The project creation step has created a Scout client that displays an empty desktop form. We will now add widgets to the clientÕs desktop form that will later display the ÒHello World!Ó message.

To add any widgets to the desktop form, navigate to the *DesktopForm* in the Scout Explorer. For this, click on the orange client node in the Scout Explorer view. Then, expand the *Forms* folder by clicking on the small triangle icon, and further expand the *DesktopForm*. As a result, the *MainBox* element becomes visible below the desktop form as shown in Figure New Form Field Menu. With a click of the right mouse button over the *MainBox*, the available context menus are displayed. To start the form field wizard select the New Form Field É menu.

Figure 11. Using the New Form Field É *menu to start the form field wizard provided by the Scout SDK.*

In the first step of the form field wizard shown on in Figure Add the DesktopBox field choose GroupBox as the form field type and click on the [!Next!] button. In the second wizard step, enter ÔDesktopÕ into the *Class Name* field before you close the wizard with the [!Finish!] button. The Scout SDK will then add the necessary Java code for the DesktopBox in the background.

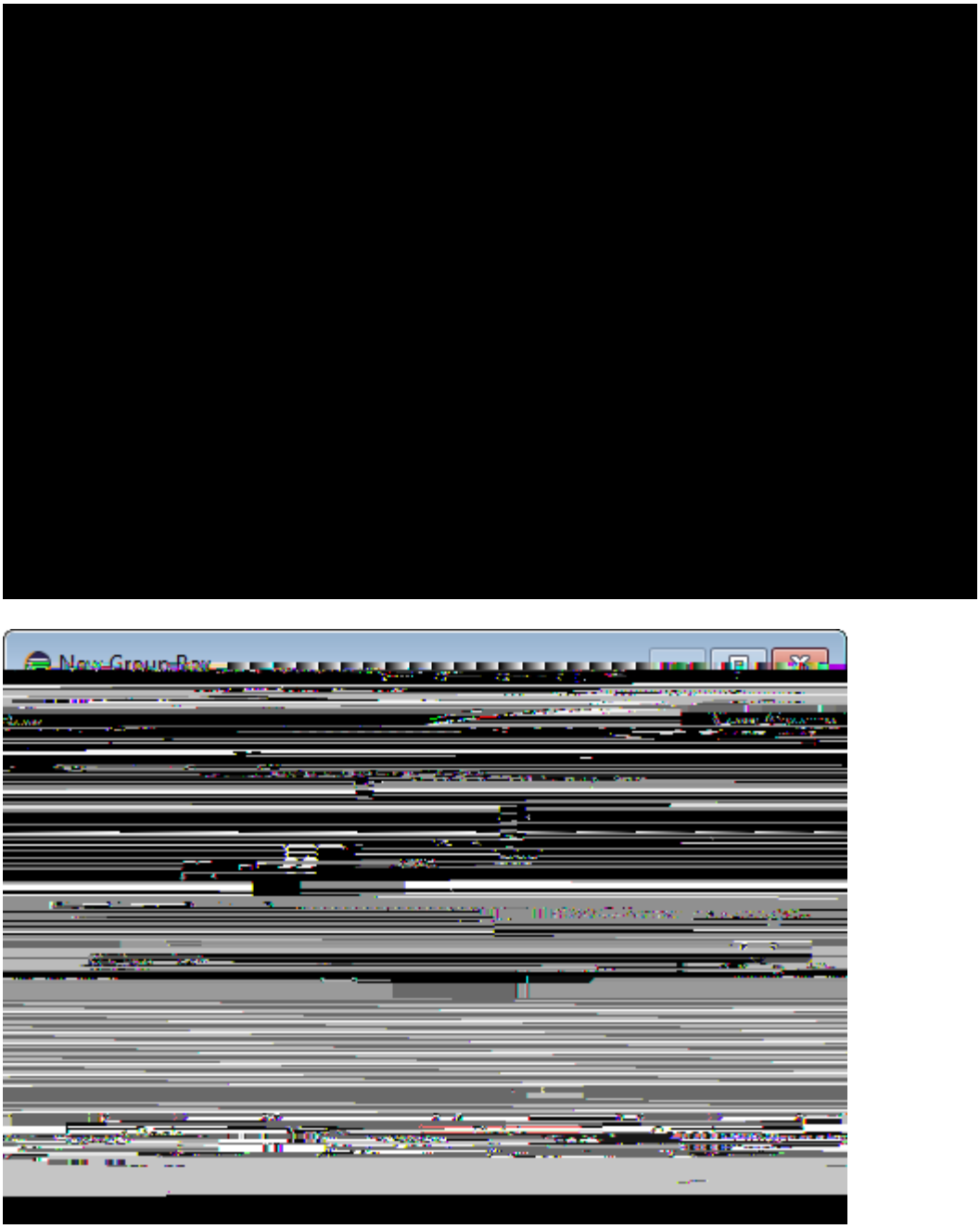


Figure 12. Adding the DesktopBox field with the Scout SDK form field wizard.

We can now add the text field widget to the group box just created. To do this, expand the *MainBox* in the Scout Explorer view to access the newly created *DesktopBox* element. On the *DesktopBox* use the New Form Field É menu again. In the first wizard step, select *StringField* as the form field type according to Figure Add a StringField. To select the *StringField* type you can either scroll down the list of available types or enter Òst Ó into the field above the field type list. In the second wizard step, enter ÔMessageÕ into the *Label* field. As we do not yet have the text ÔMessageÕ available in our ÒHello WorldÓ application the wizard prompts the user with the proposal `New Translated Text É`.

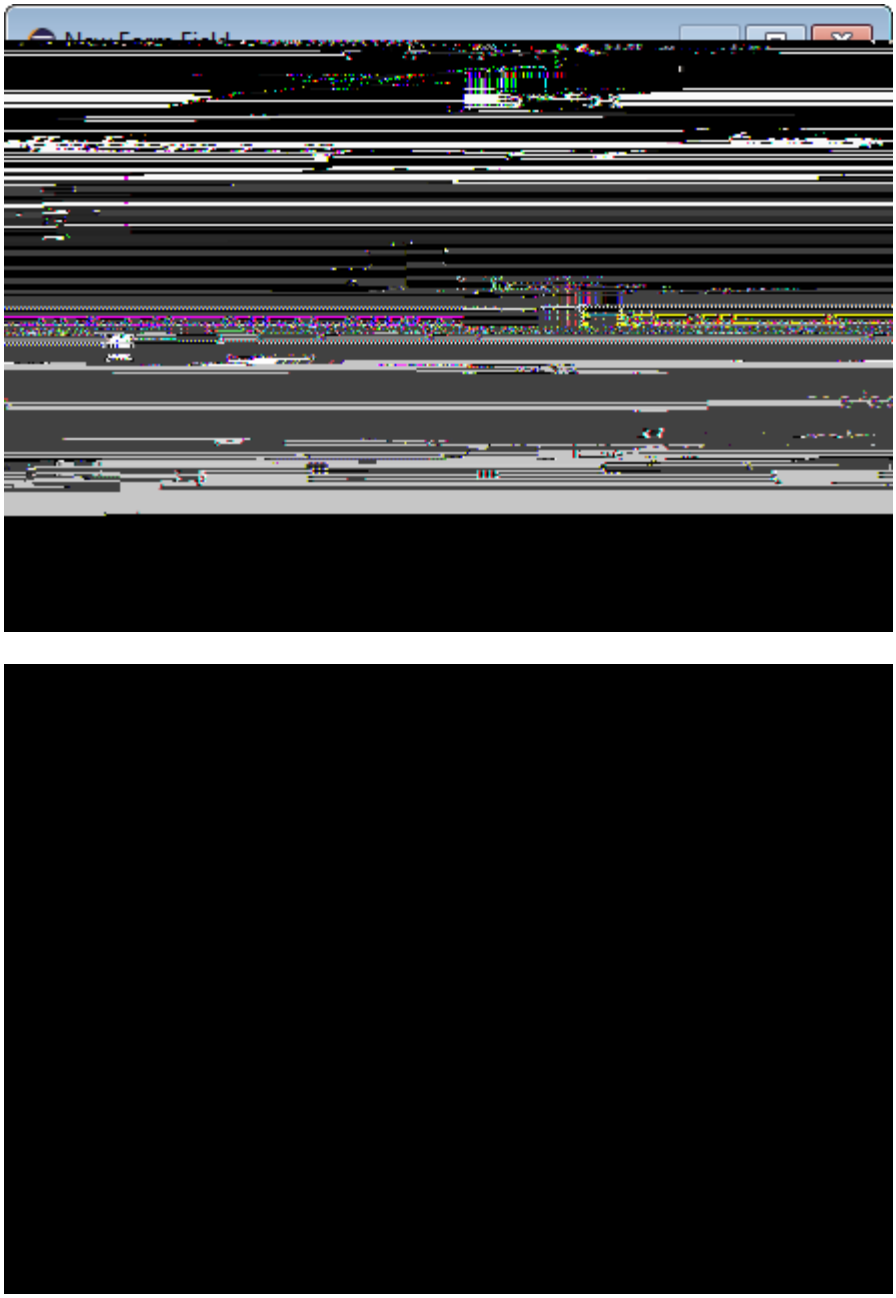


Figure 13. Adding a StringField and providing a new translation entry.

With a double click on this option a new text entry can be added to the application as shown in Figure Add a new translation entry. Once an initial translation for the message label is provided, close the translation dialog with the [!Ok!] button. Finally, close the form field wizard using the [!Finish!] button.

Figure 14. Adding a new translation entry.

By expanding the *DesktopBox* element in the Scout Explorer, the new message field becomes visible. Now, double click on the message field element to load the corresponding Java code into an editor and displays the message field's properties in the Scout Object Properties as shown in Figure showing MessageField. This is a good moment to compare your status with this screenshot. Make sure that both the Java code and the project structure in the Scout Explorer look as shown in Figure showing MessageField.

Figure 15. Scout SDK showing the MessageField

Having verified your status of the ÒHello WorldÓ application start the start the server and a client of the application as described in the previous section. The client applications will then display your message widget. However, the text widget is still empty, as we did not yet load any initial content into this field. This is the topic of the next section.

2.5. The Server Part

The responsibility of the Scout server in our ÒHello WorldÓ application is to provide an initial text content for the message field in the clientÕs user interface. We implement this behaviour in the load method of the serverÕs DesktopService. An empty stub for the load method of the DesktopService service has already been created during the initial project creation step.

To navigate to the implementation of the desktop service in the Scout SDK, we first expand the blue top-level *server* node in the Scout Explorer. Below the server node, we then expand the *Services* folder which shows the *DesktopService* element. Expanding this *DesktopService* node, the load method becomes visible as shown in Figure showing Server node.

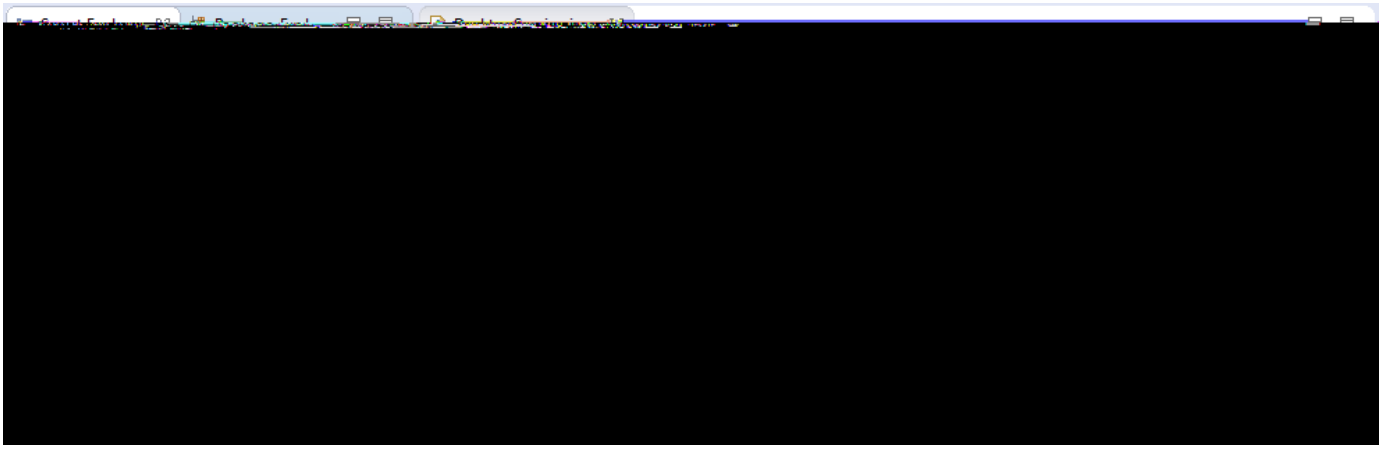


Figure 16. The Scout Explorer showing the blue server node expanded with the Services folder. In this folder the load method of DesktopService is selected and its initial implementation is shown in the editor on the right side.

The DesktopService represents the server service corresponding to the DesktopForm on the client side. This initial setup represents Scoutỏs default where client forms and server services typically come in pairs. Whenever the clientỏs user interface displays a form to the user, the client connects to the server and calls the load method of the corresponding server service. We just need to add our business logic to the load method of the serverỏs DesktopService.

According to the signature of the load method, a formData object is passed into this method that is then handed back in the return statement. To complete the implementation of the load method it is sufficient to assign the text 'hello world!' to the message field part of the form data. The complete implementation of the load method is provided in Listing load() implementation.

Listing 1. load() implementation in the DesktopService.

```
Ê @Override
Ê public DesktopFormData load(DesktopFormData formData) throws ProcessingException {
Ê   formData.getMessage().setValue("Hello World!"); !
Ê   return formData;
Ê }
```

! assign a value to the value holder part of the FormData corresponding to the message field

With this last element we have completed the Scout ÒHello WorldÓ application.

2.6. Add the Rayo Look and Feel

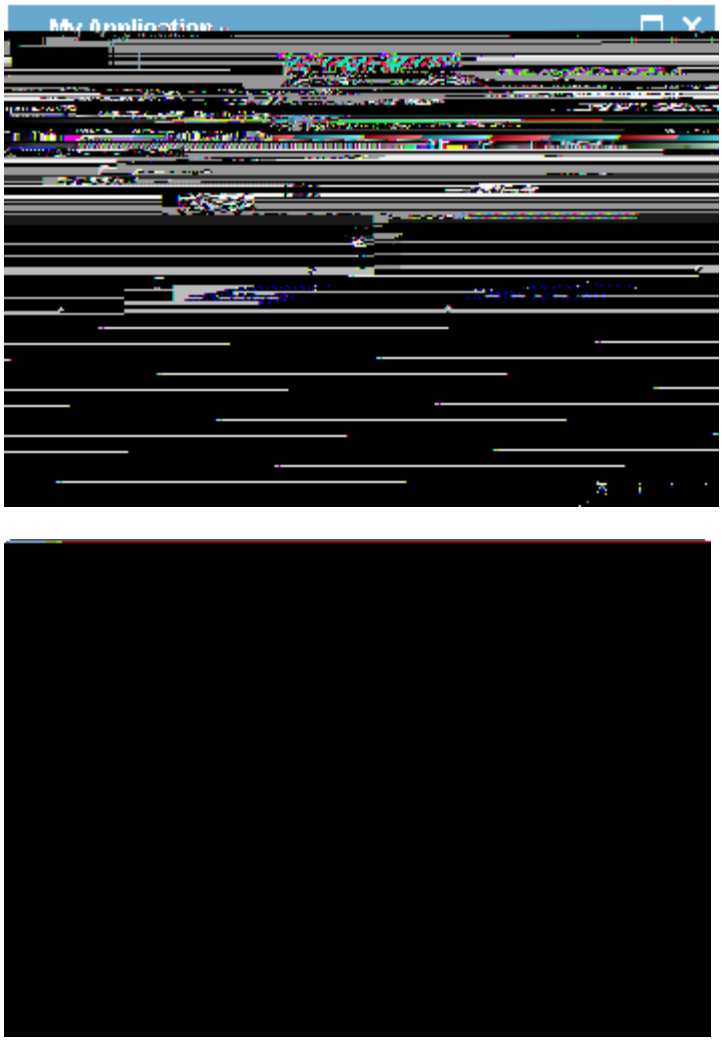


Figure 17. The ÒHello WorldÓ client application with the Rayo look and feel. The desktop client is shown on the left and the web client on the right hand side.

For Eclipse Scout applications a slick look and feel called Rayo is available in the Eclipse Marketplace. [19: Eclipse Marketplace: http://marketplace.eclipse.org/]. And in this (optional) part of the ÒHello WorldÓ tutorial we will add Rayo to our ÒHello WorldÓ Swing client application. As a result, we will get a Scout desktop application that looks the same as the corresponding Scout web client as shown in Figure 000.

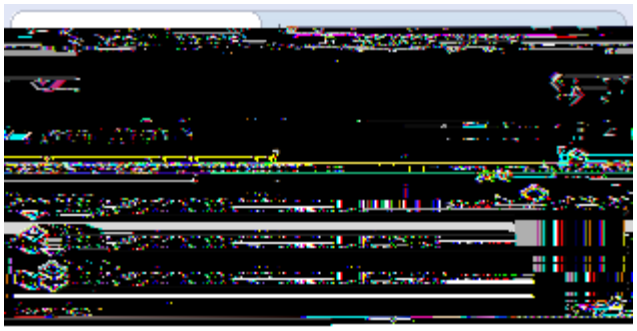


Figure 18. Adding the Rayo Swing look and feel. The Rayo checkbox to activate the look and feel is highlighted on the left hand side. The dialog on the right hand side shows the changes in the Swing plugin and the target file that will be made by the Scout SDK.

To add Rayo in the Scout SDK to our ÒHello WorldÓ project, switch to the Scout Explorer and select the top-level *org.eclipse.scout.helloworld* node. Then, according to Figure 000, select the checkbox *Rayo Swing Look and Feel for Eclipse Scout* under the *Technologies* section of the Scout Object Properties. This brings up a dialog showing the proposed changes to applicationÕs target file and the Swing plugin of the ÒHello WorldÓ application. These changes need to be confirmed with the [!OK!] button. The first time the user adds the Rayo feature in the Scout SDK, Eclipse needs to download the package from the Eclipse Marketplace. This download and subsequent installation of Rayo will make you to go through the following steps.

1. Accept Licence: GPL with Classpath Exception
2. Accept unsigned content

After the successful download and installation of the Rayo package, start the Swing client using the procedure described in Section Run the Initial Application. When we also start the web client of the ÒHello WorldÓ application using the RAP product launcher, we can compare the result side by side.

2.7. Exporting the Application

We are now ready to move the finished ÒHello WorldÓ application from our development environment to a productive setup. The simplest option to move our application into the Õwild Õ is to use the *Export Scout Project* wizard provided by the Scout SDK. Using the default settings, the export wizard produces two WAR files. [20: Web application ARchive (WAR): http://en.wikipedia.org/wiki/WAR_file_format_%28Sun%29] that contain the complete Scout server and the desktop and mobile client applications.

To deploy the application to a web server the WAR files generated by the wizard are the only artefacts needed. The first WAR file contains the Scout server including a zipped desktop client for downloading. In the second WAR file, the RAP server application that provides both the web client and the client for mobile devices.

Figure 19. Starting the Export Scout Project wizard in the Scout SDK with the context menu. In the first wizard step, the target directory for the WAR files and the artefacts to export are specified.

Figure 20. The first dialog of the Export Scout Project wizard. Here, the target directory for the WAR files that will be generated by the wizard is specified.

To start the export wizard, we start the Scout SDK with the ÒHello WorldÓ Scout project. In the Scout Explorer we then select the corresponding Export Scout ProjectÉ context menu on the ÒHello WorldÓ top level application node as shown in Figure 000. In the first wizard dialog shown in Figure 000, the

target directory for the WAR files needs to be specified. You may choose any directory as the target directory. [21: Make sure to remember the location of this directory. We will need the directory location again when we deploy these WAR files to the Tomcat web server.]. After clicking [!Next!] button the second wizard step proposes the server product file that specifies the artefacts to be exported including the file name for the WAR file for the ÒHello WorldÓ server application. Typically, the proposed default values are fine. Move to the third dialog with [!Next!] button.

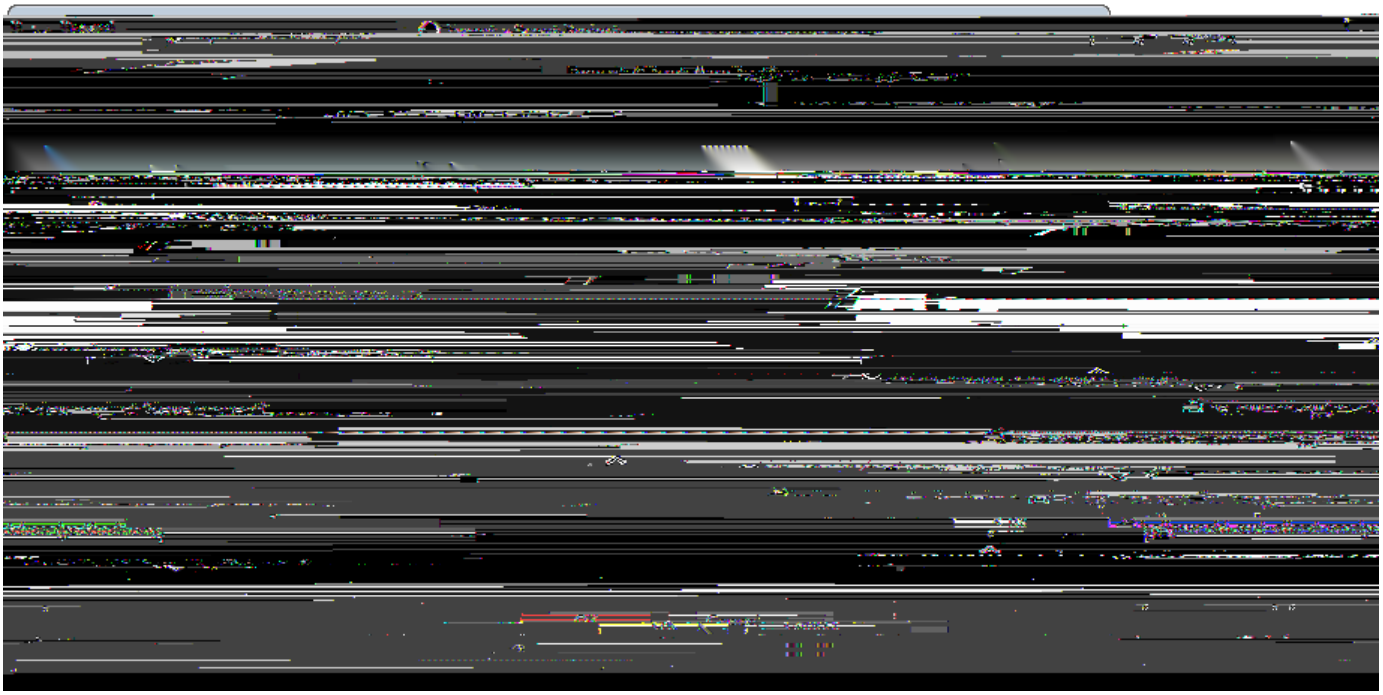


Figure 21. The third dialog of the Export Scout Project wizard defines the client application to be included in the `helloworld_server.war` *file. In the last step of the export wizard the RAP sever is exported to the specified file name (right).*

In the third dialog of the *Export Scout Project* wizard the desktop client to be included in the WAR file needs to be specified. The default selection is set to the SWT client application. For the ÒHello WorldÓ example, we want to include the Swing client application with the Rayo Look and Feel. For this, we need to change the selected product to *helloworld-swing-client.product (production)* according to Figure 000. With [!Next!] button we move to the last wizard step.

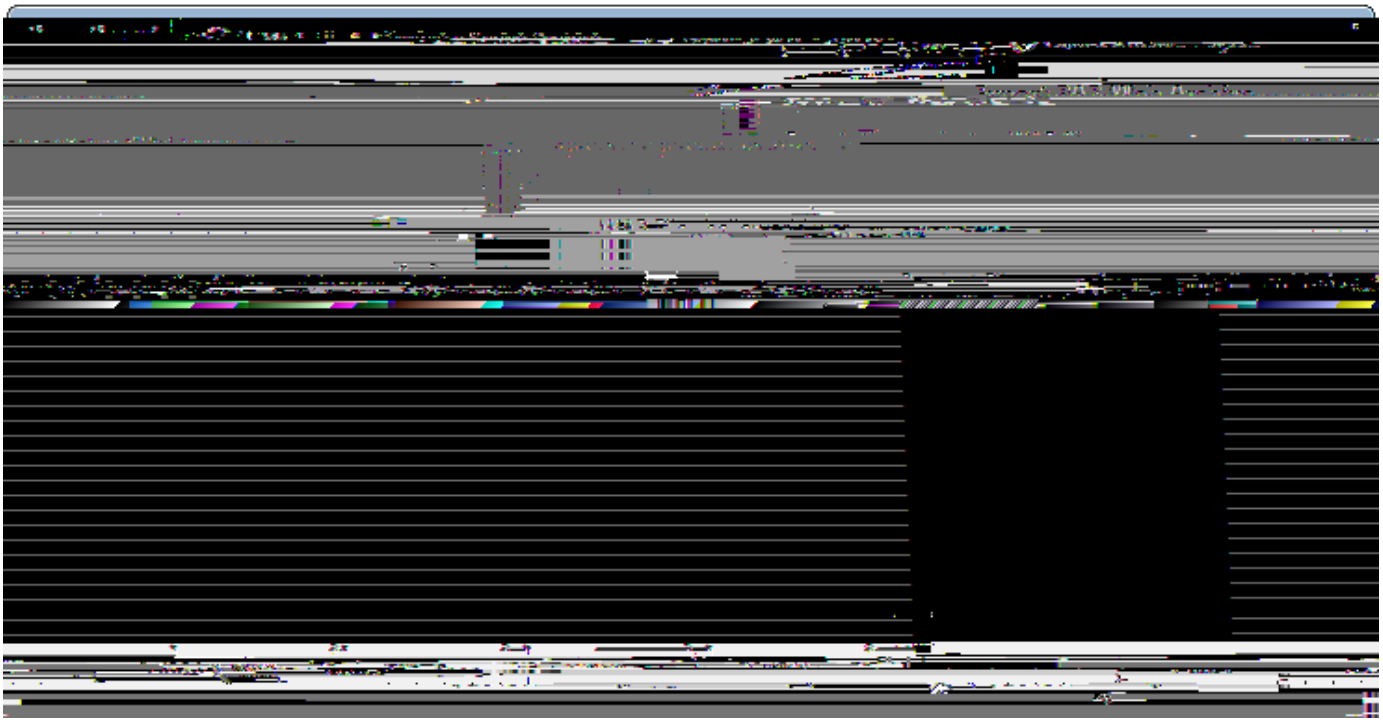


Figure 22. The last dialog of the Export Scout Project wizard defines the export of the RAP server. Normally, the proposed field values do not need any adjustments.

In the last wizard dialog shown in Figure 000, the RAP server product and the corresponding WAR file name are specified. Normally, the proposed field values are fine and we can close the wizard with [!Finish!] button. After this last step, the Scout SDK is assembling the necessary artefacts and building the two ÒHello WorldÓ WAR files. These two WAR files are the only items needed for deploying the ÒHello WorldÓ application to a web server

2.8. Deploying 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 Figure 000.

Figure 23. The Tomcat shown after a successful installation. After clicking on the ÒManager AppÓ button (highlighted in red) the login box is shown in front. A successful login shows the ÒTomcat Web Application ManagerÓ.

Figure 24. The ÒTomcat Web Application ManagerÓ. The WAR files to be deployed can then be selected using button ÒChoose FileÓ highlighted in red.

Once the web browser displays the successful running of your Tomcat instance, switch to its ÒManager AppÓ by clicking on the button highlighted in Figure 000. After entering user name and password the browser will display the ÒTomcat Web Application ManagerÓ as shown in Figure 000. If you donÕt know the correct username or password you may look it up in the file tomcat-users.xml as described in Appendix Directories and Files.

After logging successfully into TomcatÕs manager application, you can select the WAR file(s) to be deployed using button ÒChoose FileÓ according to the right hand side of Figure 000. After picking your helloworld_server.war and helloworld.war file and closing the file chooser, click on button ÒDeployÓ (located below button ÒChoose FileÓ) to deploy the application to the Tomcat web server. This will copy

the selected WAR file into Tomcats webapps directory and unpack its content into a subdirectory with the same name. Deploying the file helloworld.war will extract its contents into a subdirectory named helloworld. And the file helloworld_server.war will be extracted into subdirectory helloworld_server. You can now connect to the deployed application using the browser of your choice and enter the following address.

```
Ê http://localhost:8080/helloworld_server/
```

Figure 25. The ÒHello WorldÓ home page, providing a link to download the desktop client.

You will then see the home page of the server of your ÒHello WorldÓ application shown in Figure 000. From here you can download the zipped client application that can be saved in a directory of your choice. After unpacking the zip file, you may start the executable file named helloworld. This will start the ÒHello WorldÓ client application as shown on the left hand side of Figure 000. To start the ÒHello WorldÓ web application, open a browser and enter the following address.

```
Ê http://localhost:8080/helloworld/
```

Figure 26. The ÒHello WorldÓ client application running on the desktop, in the browser and on a mobile device.

Depending on the device your browser is running on you will be redirected to `helloworld/web` on a desktop or laptop computer, to `helloworld/mobile` on a mobile device or to `helloworld/mobile` if you are connecting from a tablet device. Figure 000 shows screenshots for a desktop client, the web application and the same application in a mobile browser. As demonstrated in these screenshots `helloworld/web` and `helloworld/mobile` lead to a different presentation of the same UI optimized to the target form factors of desktop browsers, tablets, and mobile phones.

3. ÒHello WorldÓ Background

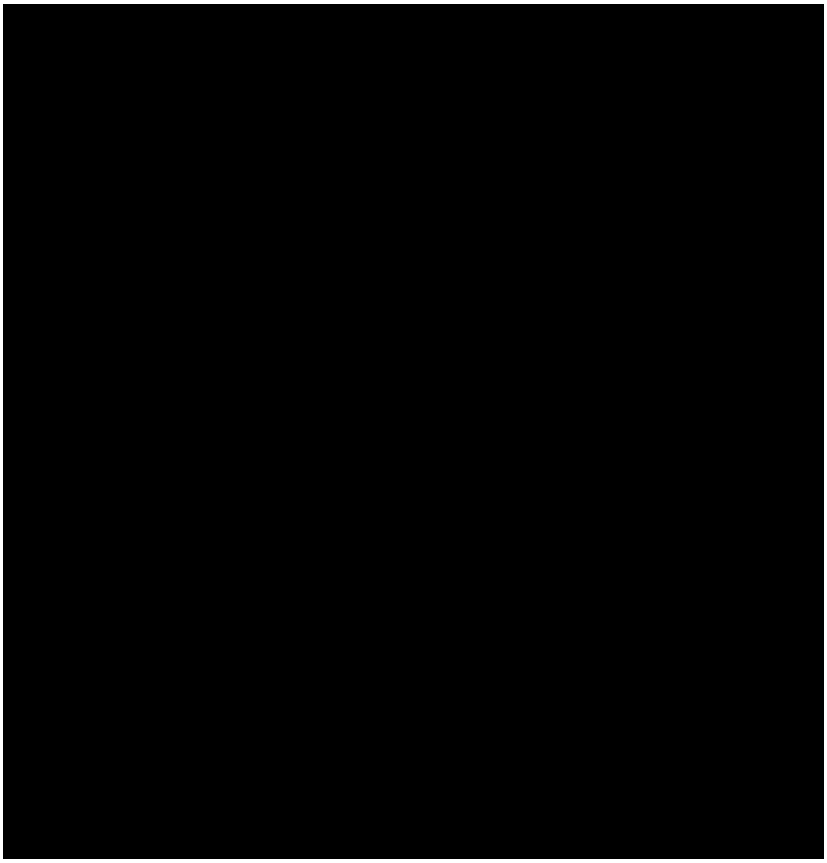
The previous ÒHello WorldÓ tutorial has been designed to cover the creation of a complete client server application in a minimal amount of time. In this chapter, we will take a deeper look at the ÒHello

World0́ and provide background information along the way. The goal is to explain many of the used concepts in the context of a concrete Scout application to allow for a well rounded first impression of the Eclipse Scout framework and the tooling provided by the Scout SDK.

The structure of this chapter is closely related to the 0̀Hello World0́ tutorial. As you will notice, the order of the material presented here exactly follows the previous tutorial and identical section titles are used where applicable. In addition to Chapter 0̀Hello World0́ Tutorial, we include Section Walking through the Initial Application to discuss the initial application generated by the Scout SDK.

3.1. Create a new Project

The first thing you need for the creation of a new Scout project is to select a new workspace. For Eclipse, a workspace is a directory where Eclipse can store a set of projects in a single place. As Scout projects typically consist of several Eclipse plugin projects the default (and recommended) setting is to use a single workspace for a single Scout project.



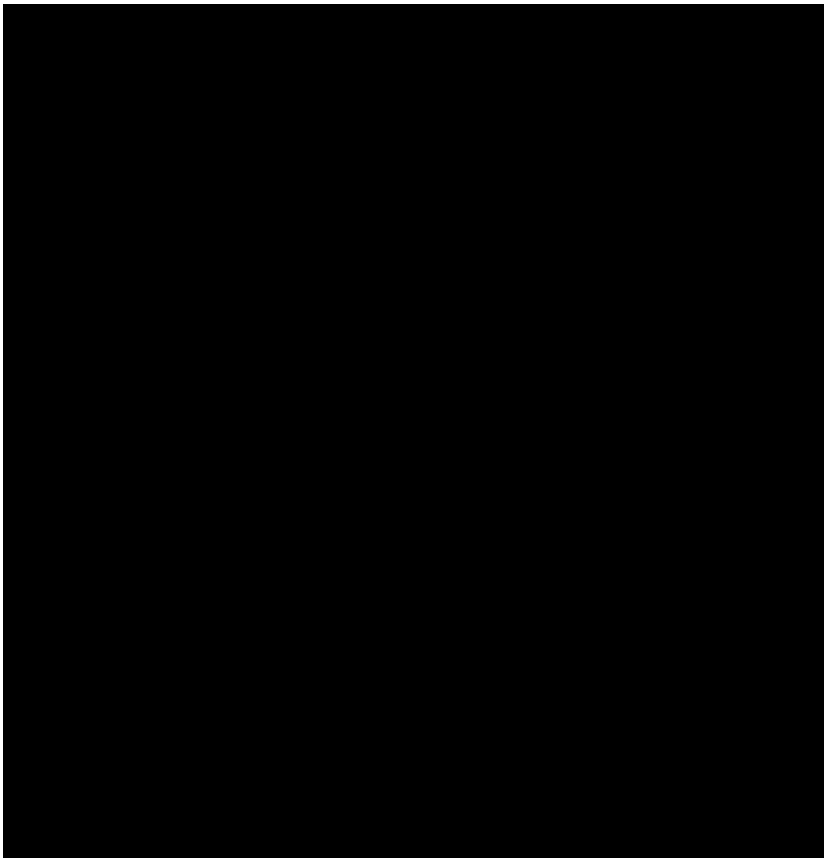


Figure 27. The Eclipse plugin projects of the ÒHello WorldÓ application shown by the Package Explorer in the Scout SDK on the left hand side. The corresponding view in the Scout Explorer is provided on the right hand side.

In the case of the ÒHello WorldÓ application, the workspace contains seven plugin projects as shown on the left side of Figure 000. In the expanded source folder of the client plugin *org.eclipse.scout.helloworld.client* the organisation of the Java packages is revealed. The Scout Explorer provided on the right side of Figure 000 shows three colored top level nodes below the main project *org.eclipse.scout.helloworld*.

In the Scout Explorer, the main project node expands to the orange client node *org.eclipse.scout.helloworld.client*, the green shared node *org.eclipse.scout.helloworld.client* and the blue server node *org.eclipse.scout.helloworld.server*. The client node first presents the white user interface (UI) nodes *org.eclipse.scout.helloworld.client.ui.** indicating the supported UI technologies. Next, the client mobile node *org.eclipse.scout.helloworld.client.mobile* is shown. It is responsible for adapting the layout of the user interface suitably for mobile and tablet devices. Finally, after the *ClientSession* node and the *Desktop* node, component specific folders allow for a simple navigation to the various client parts.

Comparing the Package Explorer with the Scout Explorer a couple of aspects are notable. First, the number and names of the Eclipse plugin projects is identical in both the Package Explorer and the Scout Explorer view. However, the Scout Explorer recognizes the Scout project structure and explicitly renders the relation between the different Eclipse plugins. In addition, individual node colors are used to indicate the role of each plugin project. Second, the focus of the Scout Explorer lies on the business functionality of the complete client server application. Artefacts only necessary to the underlying

Eclipse platform are not even accessible. Third, on the individual elements rendered in the Scout Explorer, the Scout SDK provides menus to start wizards useful to the selected context. In the case of the ÒHello WorldÓ tutorial we could create the complete application (except for a single line of Java code) using these wizards .

When we revisit the *New Scout Project* wizard in Figure New Scout Project Wizard, it now becomes trivial to explain how the *Project Name* field org.eclipse.scout.helloworld was used as the common prefix for plugin project names and Java package names. Based on the project name, the last part helloworld was used for the *Project Alias* field. As we have seen in Section Exporting the Application, this project alias is used by the Scout SDK to build the base names of the WAR files in the export step. In turn, after deploying the WAR files as described in Section Deploying to Tomcat, the RAP server application becomes available under the URL http://localhost:8080/helloworld. Should you have a catchy naming for you application in mind, com.mycompany.mycatchyname is therefore a good choice for the *Project Name* field.

3.2. Walking through the Initial Application

In this section, we will walk you through the central Scout application model components of the ÒHello WorldÓ example. As each of these components is represented by a Java class in the Scout framework, we can explain the basic concept using the available ÒHello WorldÓ source code. Below, we will introduce the following Scout components.

- Desktop
- Form
- Form handler
- Service
- MainBox
- Form data
- Form field

Please note that most of the Java code was initially generated by Scout SDK. In many cases this code can be used Òas isÓ and does not need to be changed. Depending on your requirements, it might very well be that you want to adapt the provided code to fit your specific needs. However, a basic understanding of the most important Scout components should help you to better understand the structure and working of Scout applications.

3.2.1. Desktop

The desktop is the central container of all visible elements of the Scout client application. It inherits from Scout class AbstractDesktop and represents the empty application frame with attached elements, such as the applications menu tree. In the ÒHello WorldÓ application, it is the Desktop that is first opened when the user starts the client application.

To find the desktop class in the Scout Explorer, we first navigate to the orange *client* node and double click the *Desktop* node just below. This will open the associated Java file `Desktop.java` in the editor view of the Scout SDK. Of interest is the overwritten callback method execOpened shown in Listing Desktop.

Listing 2. The configuration of the serverÔs resource servlet in the plugin.xml configuration file. The remaining content of the file has been omitted.

```
Ê @Override
Ê protected void execOpened() throws ProcessingException {
Ê   //If it is a mobile or tablet device, the DesktopExtension in the mobile plugin takes
care of starting the correct forms.
Ê   if (!UserAgentUtility.isDesktopDevice()) {
Ê     return;
Ê   }
Ê   DesktopForm desktopForm = new DesktopForm();
Ê   desktopForm.setIconId(Icons.EclipseScout);
Ê   desktopForm.startView();
Ê }
```

Method execOpened is called by the Scout framework after the desktop frame becomes visible. The only thing that happens here is the creation of a desktopForm object, that gets assigned an icon before it is started via method startView. This desktop form object holds the *Message* field text widget that is displayed to the user. [22: In the Scout application model we can only add UI fields to Scout form elements, not directly to the desktop.]. More information regarding form elements are provided in the next section.

3.2.2. Form

Scout forms are UI containers that hold form field widgets. A Scout form always inherits from Scout class AbstractForm and can either be displayed as a dialog in its own window or shown as a view inside of another UI container. In the ÒHello WorldÓ application a DesktopForm object is created and displayed as a view inside of the desktop element.

To find the desktop form class in the Scout Explorer, expand the orange *client* node. [23: To expand elements (nodes, folders, etc.) in the Scout Explorer, use a double click on the element or a single click on the plus icon in front of the element.]. Below this node, you will find the *Forms* folder. Expand this folder to show the *DesktopForm* as shown in Figure 000. In the Scout Object Property window in the screenshot, we can also see the *Display Hint* property. Its value is set to ÔViewÕ to display the desktop form as a view and not as a dialog in its own frame.

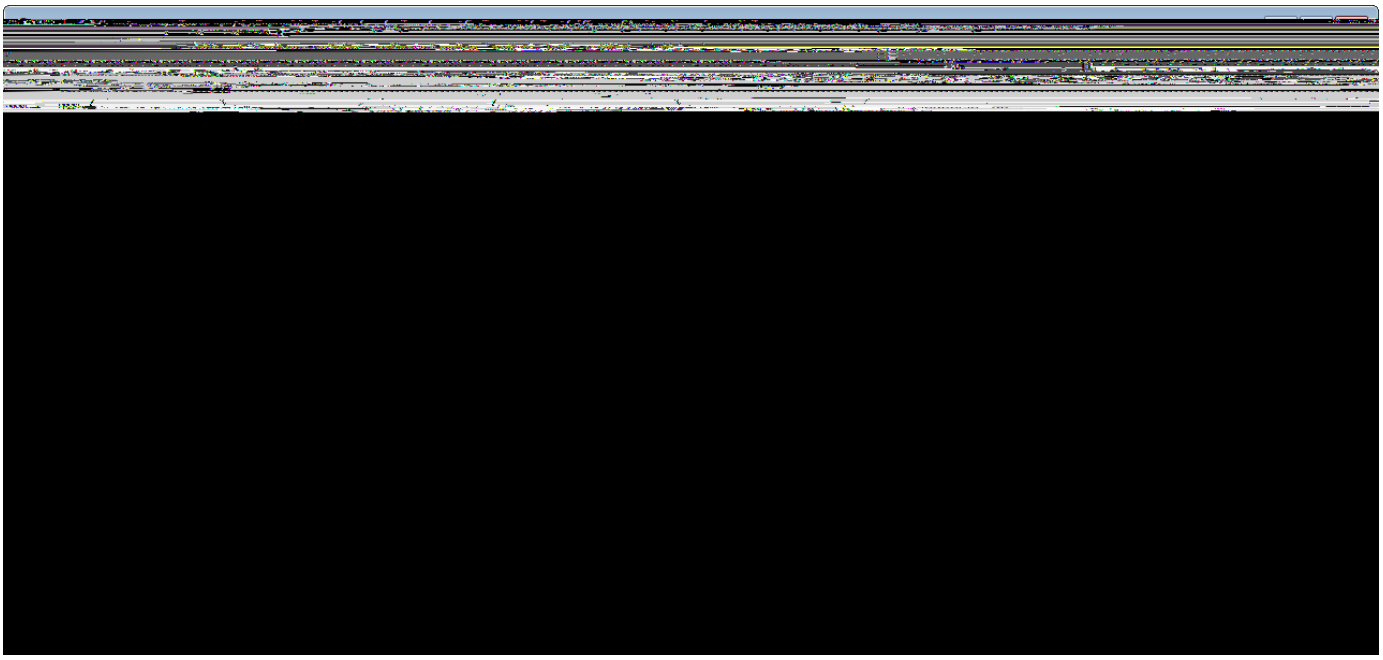


Figure 28. Scout SDK showing the DesktopFormÎs ViewHandler in the Scout Explorer and the properties of the DesktopForm in the Scout Object Properties.

Expand the *DesktopForm* to show its children: *Variables*, *MainBox* and *Handlers*. The *Variables* sub folder contains variables. They are invisible to the application user. The ÒHello WorldÓ application is so simple, it does not need variables. The sub folder *MainBox* contains form fields. These are the visible user interface elements. The main box of our DesktopForm holds the DesktopBox containing the MessageField added with the *New Form Field* wizard. Finally, the *Handlers* sub folder contains all available form handlers. The view handler shown in Figure 000 has been added in the initial project creation step.

3.2.3. Form Handler

Form handlers are used to manage the formÎs life cycle. Scout form handlers inherit from AbstractFormHandler and allow the implementation of desired behaviour before a form is opened, or after it is closed. This is achieved by overwriting callback methods defined in AbstractFormHandler. The necessary wiring is provided by the Scout framework, either by the initial project creation step or when using one of the provided Scout SDK wizards.

Listing 3. Class DesktopForm with its view handler and startView method. Other inner classes and methods are omitted here.

```
public class DesktopForm extends AbstractForm {

Ê public class ViewHandler extends AbstractFormHandler {

Ê   @Override
Ê   protected void execLoad() throws ProcessingException {
Ê     IDesktopService service = SERVICES.getService(IDesktopService.class);
Ê     DesktopFormData formData = new DesktopFormData();
Ê     exportFormData(formData);
Ê     formData = service.load(formData);
Ê     importFormData(formData);

Ê   }
Ê }

Ê public void startView() throws ProcessingException {
Ê   startInternal(new ViewHandler());
Ê }
}
```

In the ÒHello WorldÓ application, it is the overwritten execLoad method in the ViewHandler that defines what will happen before the desktop form is shown to the user. The corresponding source code is provided in Listing ViewHandler in DesktopForm. It is this execLoad method where most of the behaviour relevant to the ÒHello WorldÓ application is implemented. Roughly, this implementation is performing the following steps.

1. Get a reference to the forms server service running on the server.
2. Create a data transfer object (DTO). [24: Data Transfer Object (DTO): http://en.wikipedia.org/wiki/Data_transfer_object.]
3. Pass the empty DTO to the load service method (ask the server for some data).
4. Update the DTO with the content provided by the service load method.
5. Copy the updated information from the DTO into the desired form field.

To open the ViewHandler class in the Java editor of the Scout SDK, double click on the *ViewHandler* in the Scout Explorer. Your Scout SDK should then be in a state similar to Figure 000. In the lower part of Listing ViewHandler in DesktopForm we can see the wiring between the desktop form and the view handler in method startView. Further up, we find method execLoad of the view handler class.

Before we discuss this methodÕs implementation, let us examine when and how execLoad is actually called. As we have seen in the Desktop class (see Listing Desktop), the formÕs method startView is executed after the desktop form is created. Inside method startView (see Listing ViewHandler in

DesktopForm), the desktop form is started/opened using startInternal. In method startInternal a view handler is then created and passed as a parameter. This eventually leads to the call of our execLoad custom implementation.

We are now ready to dive into the implementation of method execLoad of the desktop formÕs view handler. First, a reference to a form service identified by IDesktopService is obtained using SERVICES.getService. Then, a form data object (the DTO) is created and all current form field values are exported into the form data via method exportFormData. Strictly speaking, the exportFormData is not necessary for the use case of the ÒHello WorldÓ application. But, as this is generated code, there is no benefit when we manually delete the exportFormData command. Next, using the load service method highlighted in Listing ViewHandler in DesktopForm, new form field values are obtained from the server and assigned to the form data object. Finally, these new values are imported from the form data into the form via the importFormData method. Once the desktop form is ready, showing it to the user is handled by the framework.

To add some background to the implementation of the execLoad above, the next section introduces services and form data objects.

3.2.4. Form Services and Form Data Objects

Form services and form data objects are used in the Scout framework to exchange information between the Scout client and server applications. When needed, a service implemented on the server side can register a corresponding proxy service on the client. This proxy service is invoked by the client as if it were implemented locally. In fact, when we get a service reference using SERVICES.getService, we do not need to know if this service runs locally on the client or remotely on the server.

In the ÒHello WorldÓ example application, the clientÕs desktop form has an associated desktop service running on the server. This correspondence between forms and form services is also reflected in the *Links* section of the Scout Object Properties of the desktop form. As shown in Figure 000, links are provided not only for the desktop form, but for its desktop form data, the corresponding desktop form service as well as for the service interface IDesktopService. On the client, this interface is used to identify and register the proxy service for the desktop service.

To transfer data between the client and the server, the ÒHello WorldÓ application uses a DesktopFormData object as a DTO. This form data object holds all form variables and values for all the form fields contained in the form. Taking advantage of this correspondence, the Scout framework provides the convenience methods exportFormData and importFormData. As a result, the developer does not need to deal with any mapping code between the form data object and the form fields.

The actual implementation of the desktop form service in class DesktopService is implemented on the server side. As the class DesktopService represents an ordinary Scout service it inherits from AbstractService. It also implements its corresponding IDesktopService interface used for registering both the actual service as well as the proxy service.

3.3. Run the Initial Application

3.3.1. The Launcher Boxes

To run a Scout application the Scout SDK provides launcher boxes in the Scout Object Properties as described in Section Run the Initial Application. These object properties are associated to the top level project node in the Scout Explorer. Using the *Edit* icon provided in the product launcher section of the Scout Object Properties, the list of launcher boxes can be specified as shown in Figure 000.

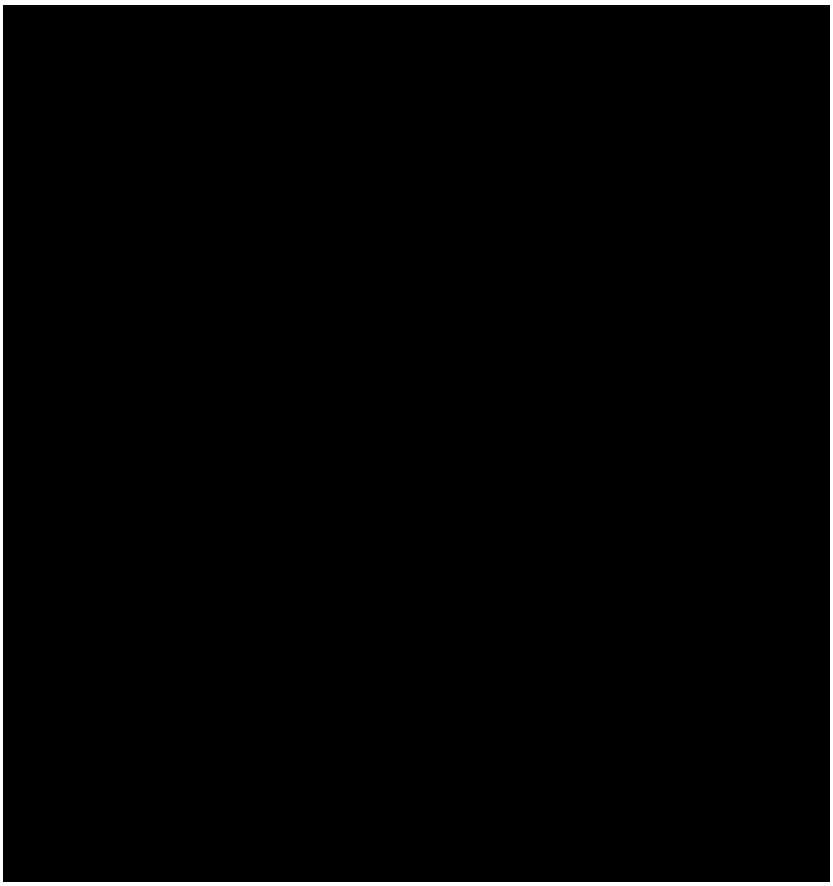
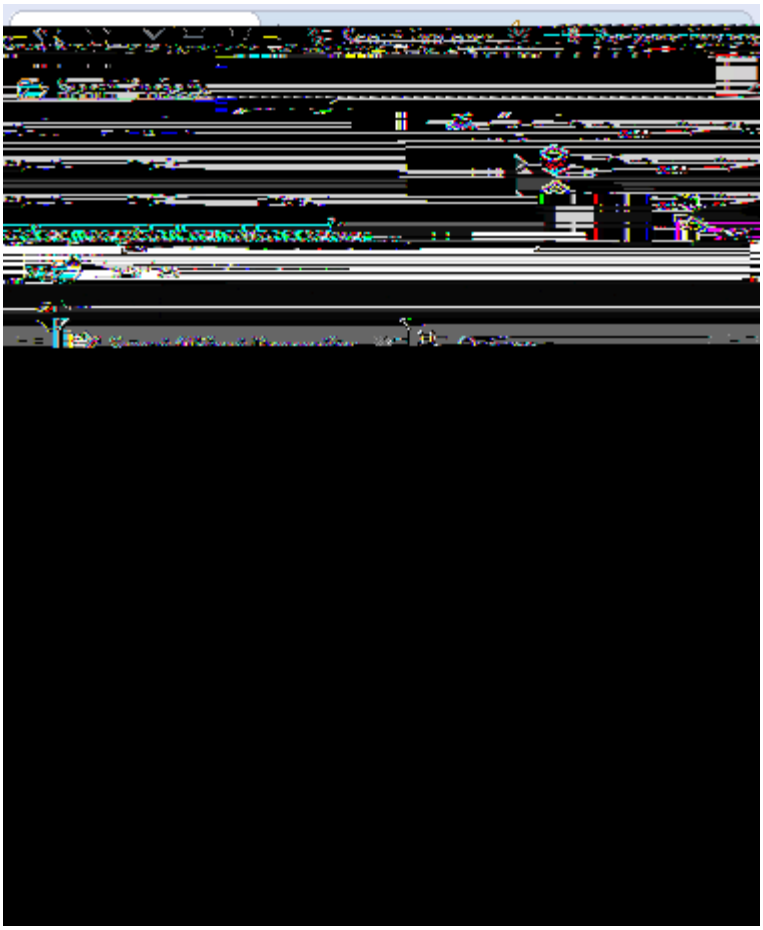


Figure 29. Using the Edit ContentÉ icon shown on the left hand side, the product selection dialog shown on the right side is opened. Using this product selection dialog, the list of launcher boxes can be specified.

3.3.2. Eclipse Product Files

The available products shown on the right side of Figure 000 represent the Eclipse product files created in the initial project creation step. Product files. [25: Read the following article for an introduction to Eclipse product files: http://www.vogella.com/articles/EclipseProductDeployment/article.html.] are used in Eclipse to specify the configuration and content of an executable application. In the case of the ÒHello WorldÓ project, four executable applications --- with two Eclipse product files for each application --- have been defined by the Scout SDK. The four applications, one for the server application and one for each client technology, have already been discussed in Section Run the Initial Application.



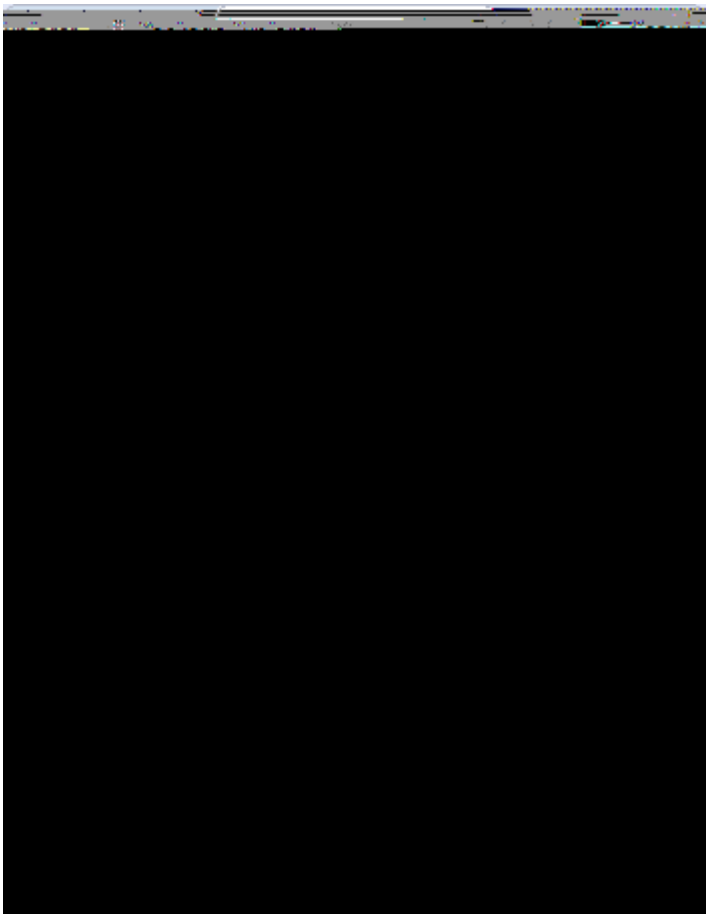


Figure 30. The production and development launcher boxes associated with the ÒHello WorldÓ server application are shown on the left side. In the Package Explorer shown on the right side, the production and development products are located under the products folder in the server plugin project.

We assume that Scout applications will be run in at least two different environments. Once from within the Eclipse IDE in the development environment, and once by the actual end users outside the Eclipse IDE. This second environment is named production environment. Depending on the complexity of deployment processes there might be some more environments to consider, such as testing and integration environments. This is the reason that the Scout SDK initially creates two product files that are associated with the development and the production environment.

Even in the case of the simple ÒHello WorldÓ example, the Scout application is started in two target environments. The development environment defines the product in the context of the Scout SDK. To export and run the Scout application outside of the Scout SDK, the production product files are used to define the application when it is to be started on a Tomcat web server. Figure 000 illustrates this situation for the ÒHello WorldÓ server application. On the left side, the blue server node is selected in the Scout Explorer. This opens the two server launcher boxes for the production and the development environment. On the right side of Figure 000, the corresponding plugin project *org.eclipse.scout.helloworld.server* is expanded to show the file based organisation of the two product definitions.

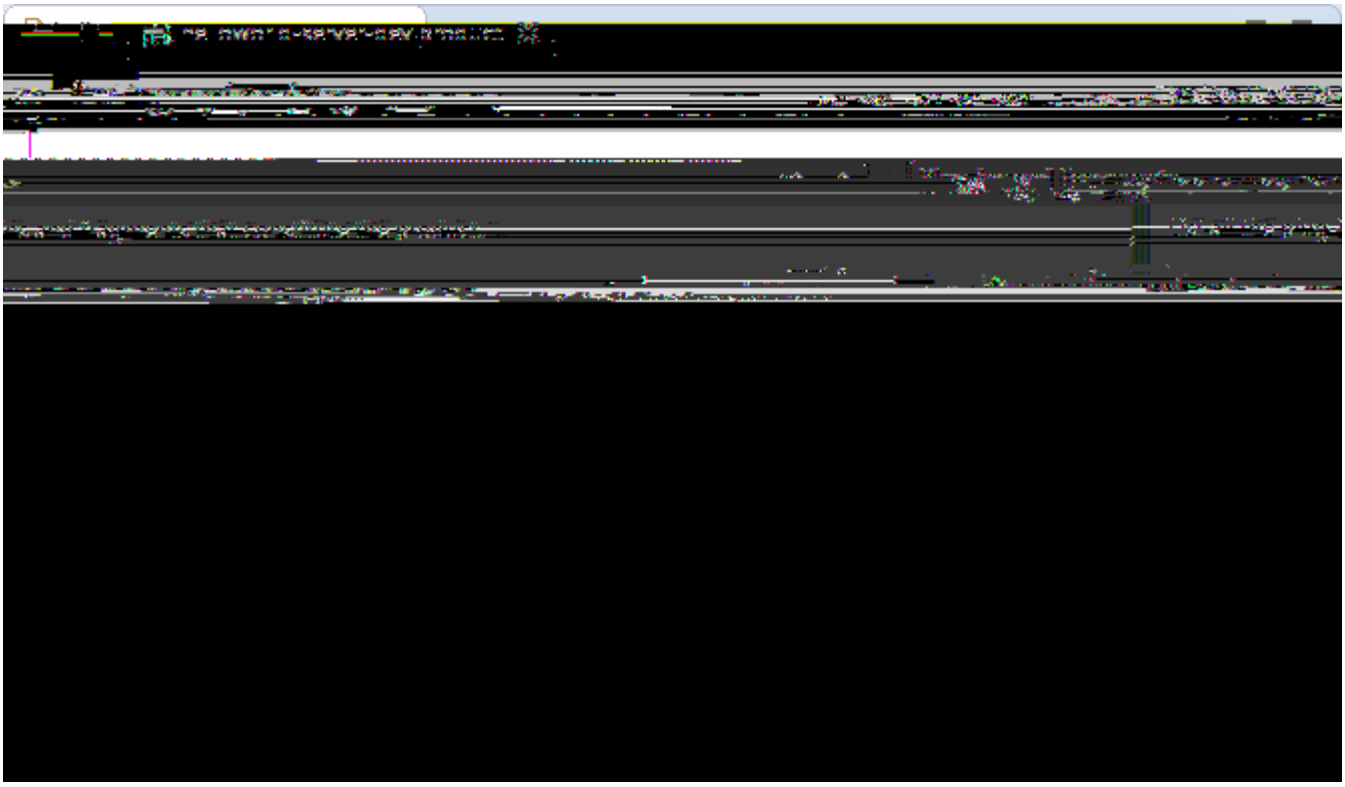


Figure 31. The Eclipse product file editor showing file helloworld-server-dev.product of the ÒHello WorldÓ application. In the Dependencies tab shown above, the list of Eclipse plugins that are required for the server application are shown.

For the case of the ÒHello WorldÓ example we did not need to edit or change the product files generated by the Scout SDK. However, if your requirements are not met by the provided product files, you may use the Eclipse product file editor. A screenshot of this editor is shown in Figure 000 with the tab *Dependencies* opened. In the tab Dependencies, the complete list of necessary plugins is provided. Example plugins visible in Figure 000 include the ÒHello WorldÓ server and shared plugins, Scout framework plugins, and Jetty plugins. The Jetty. [26: Jetty is web server with a small footprint: http://www.eclipse.org/jetty/.] plugins are only needed to run the ÒHello WorldÓ server application inside the Scout SDK. Consequently, Jetty plugins are not listed as a dependency in the Scout serverÕs production product file.

3.3.3. Eclipse Configuration Files

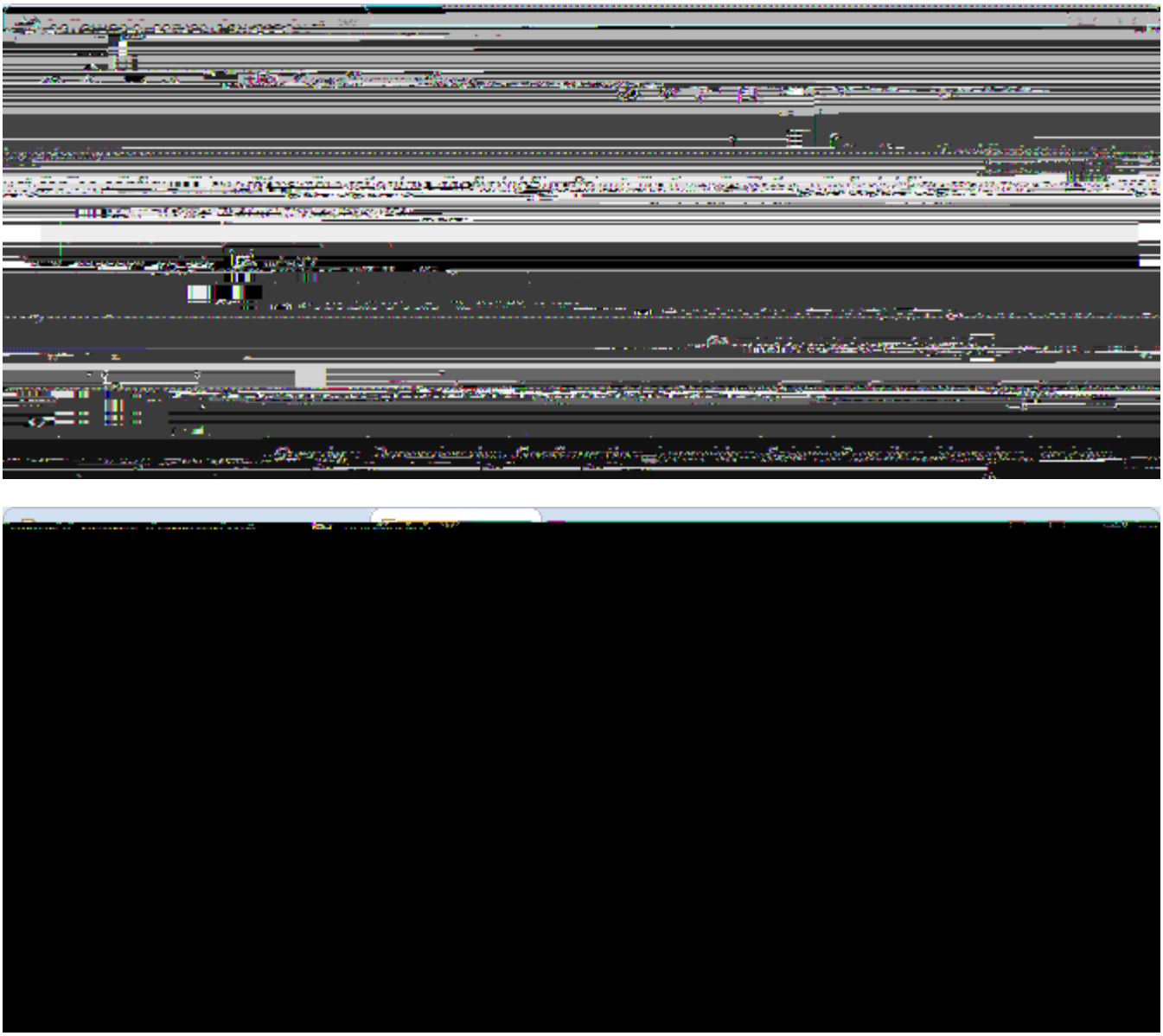


Figure 32. Above, the definition of the products config.ini in tab Configuration of the product file editor. Below, the content of the configuration file of the ÒHello WorldÓ server application is provided in a normal text editor.

Switching to tab *Configuration* in the product file editor, shows the selected radio button *Use an existing config.ini file* and the link to the configuration file provided in the *File* field as shown in the upper part of Figure 000. Below, a part of the serverÕs config.ini file is shown. Both the entry in the product file pointing to the configuration file, and the content of the config.ini file has been generated by the Scout SDK during the initial project creation step. As shown in the lower part of Figure 000, Eclipse configuration files have the format of a standard property file. The provided key value pairs are read at startup time if the config.ini file can be found in folder configuration by the Eclipse runtime.

3.3.4. Scout Desktop Client Applications

Having introduced Eclipse product files and configuration files based on the ÒHello WorldÓ server

application, we will now look at the different client applications in turn. With Swing applications. [27: Swing is the primary Java UI technology: http://en.wikipedia.org/wiki/Swing_%28Java%29.] and SWT applications. [28: Standard Widget Toolkit (SWT): http://en.wikipedia.org/wiki/Standard_Widget_Toolkit.], two alternative UI technologies are currently available to build Scout desktop client applications. More recently, JavaFX. [29: JavaFX is the most recent Java UI technology: http://en.wikipedia.org/wiki/JavaFX.] is promoted as a successor to Swing and it is therefore likely, that Scout will provide JavaFX client applications in the future.

When we compare the product files for the Swing and the SWT client applications, it is apparent that both client applications share a large number of plugins. Most importantly, the complete UI model and the business logic is identical for both client applications. In other words, the value created by the Scout developer is contained in the two plugins *org.eclipse.scout.helloworld.client* and *org.eclipse.scout.helloworld.shared*. To create an executable client application, we only need to combine these two plugins with a set of plugins specific to the desired UI technology.

After starting the ÒHello WorldÓ Swing client or the corresponding SWT client application, the client application first reads the startup parameters from its config.ini file. Among other things, this client configuration file contains the parameter server.url to specify the URL to the ÒHello WorldÓ server. After the startup of the ÒHello WorldÓ client application, it can then connect to the ÒHello WorldÓ server application using this address.

3.3.5. Scout Web, Tablet and Mobile Clients

For Scout web, tablet and mobile clients, the Eclipse RAP framework. [30: Remote Application Platform (RAP): http://www.eclipse.org/rap/.] is used. The RAP framework provides an API that is almost identical to the one provided by SWT and allows to use Java for server-side Ajax. [31: Asynchronous JavaScript and XML (AJAX): http://en.wikipedia.org/wiki/Ajax_%28programming%29.]. This setup implies that Scout tablet and mobile clients are not native clients but browser based. [32: To provide native clients with Scout, the simplest (commercial) option is most likely Tabris: http://developer.eclipsesource.com/tabris/].

Comparing the product file of the SWT client applications with the RAP application, we observe that the RAP development product does not include any SWT plugins, but a set of RAP and Jetty plugins. In addition, the RAP product also contains the Scout mobile client plugins *org.eclipse.scout.rt.client.mobile* and *org.eclipse.scout.helloworld.client.mobile*. These two plugins are responsible for transforming the UI model defined in the ÒHello WorldÓ client plugin to the different form factors of tablet computers and mobile phones.

If you start the ÒHello WorldÓ RAP application in your Scout SDK, you are launching a second server application in a Jetty instance on a different port than the ÒHello WorldÓ server application. As in the case of the desktop client applications, the RAP or Ajax server application knows how to connect to the ÒHello WorldÓ server application after reading the parameter server.url from its config.ini file.

3.4. The User Interface Part

Using the UI of the ÒHello WorldÓ application we explain in this section how the Scout UI form model is represented in Java. We also describe how this representation is exploited by the Scout SDK to automatically manage the form data objects used for data transfer between Scout client and Scout server applications. Finally, will have a brief look at internationalization. [33: Internationalization and localization, also called NLS support: http://en.wikipedia.org/wiki/Internationalization_and_localization.] support of Scout for texts.

Listing 4. The DesktopForm with its inner class MainBox containing the desktop box and message field

```
@FormData(value = DesktopFormData.class, sdkCommand = FormData.SdkCommand.CREATE)
public class DesktopForm extends AbstractForm {

Ê @Order(10.0)
Ê public class MainBox extends AbstractGroupBox {

Ê   @Order(10.0)
Ê   public class DesktopBox extends AbstractGroupBox {

Ê     @Order(10.0)
Ê     public class MessageField extends AbstractStringField {

Ê       @Override
Ê       protected String getConfiguredLabel() {
Ê         return TEXTS.get("Message");
Ê       }
Ê     }
Ê   }
Ê }
}
```

As discussed in Section Form Scout forms consist of variables, the main box and a number of form handlers. The main box represents the visible part of ScoutÔs form model. It may holds any number of form fields. Using container fields such as group boxes, it is possible to define complex structures such as hierarchical UI models containing multiple levels. In the Scout framework the forms structure is represented in the form of inner classes that are located inside of the MainBox class. And the *New Form Field* wizard of the Scout SDK fully supports this pattern. Listing MainBox provides the concrete example using the the desktop form of the ÒHello WorldÓ tutorial.

Using inner Java classes to model a formÔs content is a central aspect of the UI part of the Scout application model. It allows the Scout SDK to easily parse the formÔs Java code on the fly and directly reflect changes to the form model in the Scout Explorer and the Scout Property View. However, this is not the only benefit for the Scout SDK. As form data objects hold all form variables and the values of all form fields contained in the form, the Scout SDK can keep the form data classes in sync with the forms

of the application. It is important to note that this mechanism only depends on the Java code of the form field class. In consequence, the Scout SDK can update form field classes in the background even when form fields are manually coded into the formÕs Java class. This includes adding all the necessary getter and setter methods to access the values of all the fields defined on a form. As a result, Scout developers donÕt need to manually update form data objects when the UI model of a form is changed. The Scout SDK takes care of this time consuming and error prone task.

Listing 5. The HelloworldTextProviderService class. Its getter method provides the path and the base name for the text property files

```
public class HelloworldTextProviderService extends AbstractDynamicNlsTextProviderService
{
Ê @Override
Ê protected String getDynamicNlsBaseName() {
Ê   return "resources.texts.Texts";
Ê }
}
```

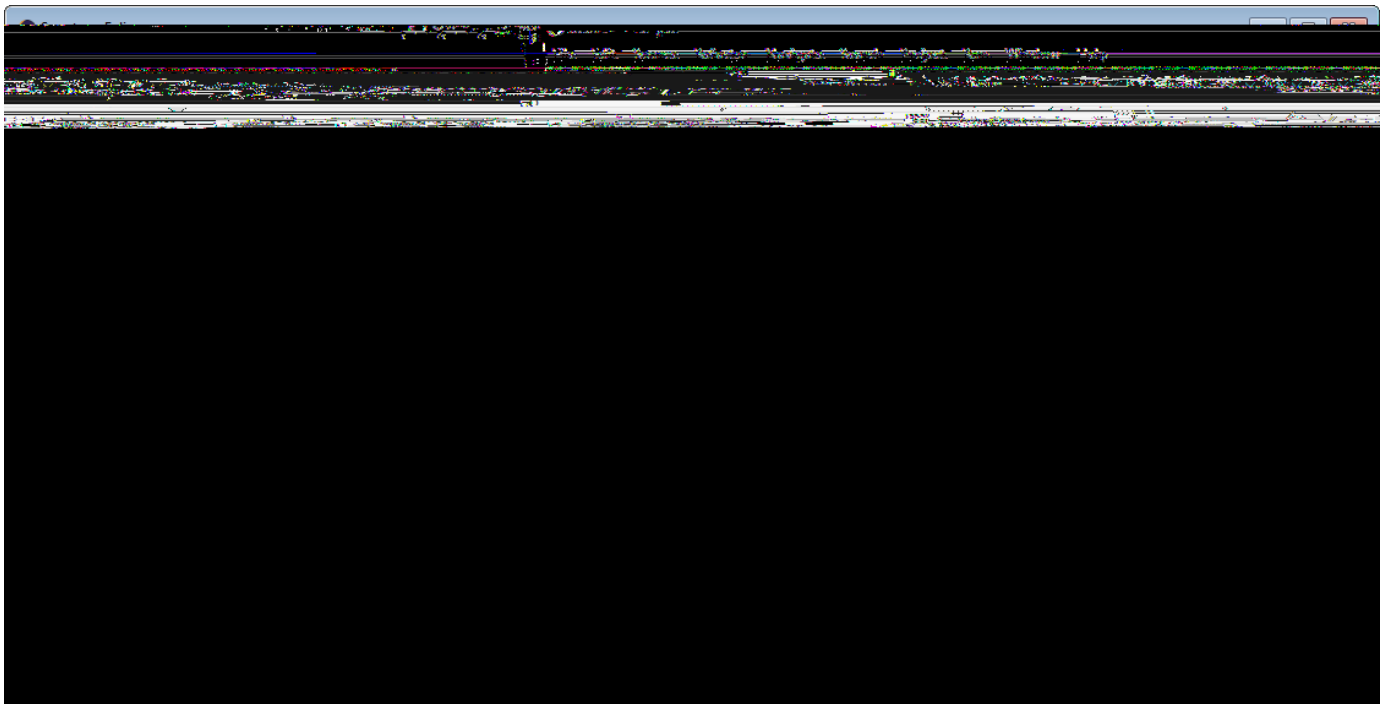


Figure 33. The NLS editor provided by the Scout SDK. This editor is opened via the Open NLS Editor É link in the Scout Object Properties of the HelloworldTextProviderService node.

When we did add the *Message* field to the desktop form of the ÒHello WorldÓ application we had to enter a new translation entry for the label of the message field as shown in Figure Add a StringField. The individual translation entries are then stored in language specific text property files. To modify translated texts we can use the NLS editor. [34: See Section [sec-nls_editor] for a detailed description of the NLS editor.] provided by the Scout SDK as shown in Figure 000.

To access the translated label field entry in the application, the Scout SDK generated the

implementation of getConfiguredLabel using TEXTS.get("Message") as shown in Listing MainBox. In the default Scout project setup, calling TEXTS.get uses the DefaultTextProviderService in the background. This text provider service then defines the access path for the text property files to use for the translation. To resolve the provided key, the user's locale settings are used to access the correct text property file.

3.5. The Server Part

In this background section we take a closer look at Scout services and calling service methods remotely. We will first discuss the setup of an ordinary Scout service. Then, the additional components to call service methods remotely are considered. To explain the concepts in a concrete context, we use the setup of the DesktopService of our "Hello World" example.

3.5.1. Scout Services

Scout services are OSGi services. [35: A good introduction to OSGi services is provided by Lars Vogel's tutorial: http://www.vogella.com/articles/OSGiServices/article.html.] which in turn are defined by standard Java classes or interfaces. Scout is just adding a convenience layer to cover typical requirements in the context of client server applications. To support Scout developers as much as possible, the Scout SDK offers wizards that generate the necessary classes and interfaces and also take care of service registration.

Listing 6. The server service class DesktopService.

```
public class DesktopService extends AbstractService implements IDesktopService {

Ê //tag::load[]
Ê @Override
Ê public DesktopFormData load(DesktopFormData formData) throws ProcessingException {
Ê   formData.getMessage().setValue("Hello World!"); !
Ê   return formData;
Ê }
```

All Scout services need to extend Scout's AbstractService class and implement their own corresponding interface. This also applies to the "Hello World" desktop service according to Listing DesktopService. As shown in Figure showing Server node, this service can be located in the Scout Explorer under the blue server node in the *Services* folder.

Before Scout services can be accessed and used, they need to be explicitly registered as a service in the correct place. For this registration mechanism, Scout is using Eclipse extension points and extensions. [36: A good introduction to Eclipse extensions and extension points is provided in the Eclipse wiki: http://wiki.eclipse.org/FAQ_What_are_extensions_and_extension_points%3F.] which are conceptually similar to electrical outlets and plugs. And in order to work, as in the case of outlets and plugs, the plug must fit to the outlet. In our "Hello World" example, the extension (plug) is represented by class

DesktopService and the service extension point (outlet) is named org.eclipse.scout.service.services. What makes the desktop service fit to the service extension point is the fact that its interface IDesktopService extends Scoutûs IService interface.

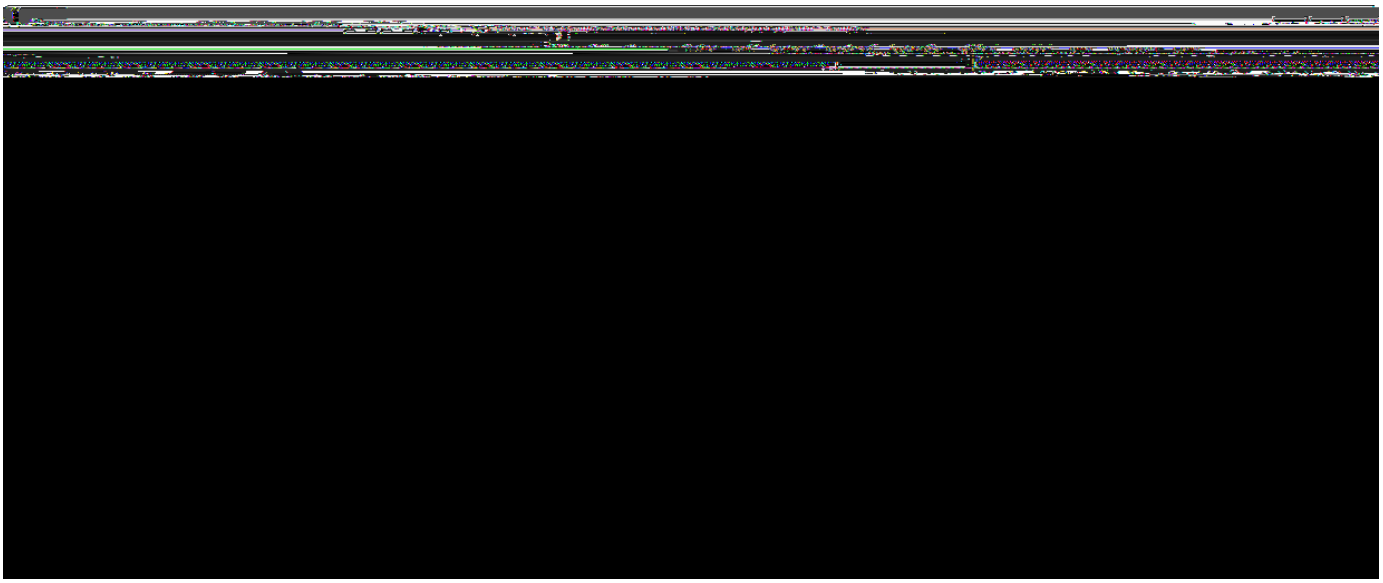


Figure 34. The Eclipse plugin editor for plugin.xml files. In the tab Extensions the ÒHello WorldÓ desktop service is registered under the extension point org.eclipse.scout.service.services.

Listing 7. The registration of the DesktopService in the serverûs plugin.xml configuration file. The remaining content of the file has been omitted.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin>

Ê  <extension
Ê        name=""
Ê        point="org.eclipse.scout.service.services">
Ê     <service
Ê           factory="org.eclipse.scout.rt.server.services.ServerServiceFactory"
Ê           class="org.eclipse.scout.helloworld.server.services.DesktopService"
Ê           session="org.eclipse.scout.helloworld.server.ServerSession">
Ê     </service>
Ê  </extension>

</plugin>
```

The registration of the desktop service under the service extension point is then defined in the plugin.xml file of the ÒHello WorldÓ server plugin. As shown in Figure 000, the plugin.xml file is located in the root path of plugin org.eclipse.scout.helloworld.server. To modify a plugin.xml, you can either use the Eclipse plugin editor or your favorite text editor. In Figure 000, the registration of the desktop service is shown in the *Extensions* tab of the plugin editor. For the corresponding XML representation in the plugin.xml file, see Listing server plugin.xml.

3.5.2. Scout Proxy Services

In the ÒHello WorldÓ application the load method of the desktop service is called remotely from the client. But so far, we have only seen how the desktop service is implemented and registered in the server application. To call server service methods remotely from Scout client applications, the Scout framework provides client proxy services and the service tunnel. As the name implies, a client proxy service acts as a local proxy service (running in the Scout client application) of a server service (running remotely in the Scout server application).

Listing 8. The registration of the IDesktopService proxy service in the client plugin of the ÒHello WorldÓ application. This is the complete content of the clientÕs plugin.xml file.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin>

Ê  <extension
Ê        name=""
Ê        point="org.eclipse.scout.service.services">
Ê     <proxy
Ê           factory="org.eclipse.scout.rt.client.services.ClientProxyServiceFactory"
Ê           class="org.eclipse.scout.helloworld.shared.services.IDesktopService">
Ê     </proxy>
Ê  </extension>

</plugin>
```

Client proxy services are defined by a Java interface located in the shared plugin of the Scout application. As shown in Listing DesktopService of the desktop service, this service interface is also implemented by the desktop service class in the server plugin. Corresponding to the registration of the desktop service in the server plugin, client proxy services need to be registered in the clientÕs plugin.xml file. The content of the ÒHello WorldÓ client plugin configuration file is provided in Listing client plugin.xml. To create proxy services in Scout clients, the ClientProxyServiceFactory is used. This is also reflected in the extension defined in Listing client plugin.xml. Internally, this service factory then uses the service tunnel to create the local proxy services.

To call a remote service method from the Scout client application, we first need to obtain a reference to the proxy service. Using the SERVICES.getService method with the interface IDesktopService, we can obtain such a reference as shown in Listing ViewHandler in DesktopForm for the view handler of the desktop form. With this reference to the clientÕs proxy service, calling methods remotely works as if the service would be running locally. Connecting to the server, serializing the method call including parameters (and de serializing the return value) is handled transparently by Scout.

3.6. Add the Rayo Look and Feel

Rayo has been designed in 2009 by BSI for its CRM. [37: Customer Relationship Management (CRM): https://en.wikipedia.org/wiki/Customer_relationship_management] application and contact center solution. Since then, Rayo has been copied for Scout web applications and also adapted to work on touch/mobile devices.

The implementation of Rayo for desktop clients is based on the Java Synth look and feel. [38: Java Synth Look and Feel: http://en.wikipedia.org/wiki/Synth_Look_and_Feel]. However, in a few cases it was necessary to adjust some of the synth classes. In order to do this, the adapted classes are copied form the OpenJDK implementation. [39: OpenJDK is an open source implementation of the Java platform: http://openjdk.java.net/.] As OpenJDK is licenced under the GNU General Public Licence (GPL) with a linking exception it is not possible to distribute Rayo under the Eclipse Public Licence. That is why Rayo is not initially contained in the Eclipse Scout package but needs to be downloaded from the Eclipse Marketplace. Fortunately, there is still no restriction to use Rayo in commercial products. The only remaining restriction applies to modifying Rayo for commercial products. In this case you will be obliged to redistribute your modified version of Rayo under the same licence (GPL with classpath exception).

With Eclipse Scout 3.8 (Juno), the Scout framework also allows to build web clients based on Eclipse RAP. Great care has been taken to ensure, that the look and feel for Scout web applications matches the look and feel of the desktop as closely as possible. As RAP is already distributed under the EPL licence the Rayo for web apps is directly contained in the Scout package. TODO: Describe what to change to use RAP default look and feel

A similar approach was chosen for Rayo on tablets and mobile devices that are supported with Eclipse Scout 3.9 (Kepler). For such devices optimized components are used to take into account the smaller screens and the absence of a mouse (no context menus!) But as far as possible, the Rayo look and feel also applies to touch devices. TODO: Pointer to more info regarding mobile devices.

3.7. Exporting the Application

In this background section we look at the content and organisation of the two WAR files generated by the Scout SDK *Export Scout Project* wizard. The first WAR file holds the Scout server including a landing page to download the Scout desktop client. The desktop client is provided in the form of a standalone ZIP file. In the second WAR file, the Ajax server based on Eclipse RAP is contained. This Ajax server provides the URLs that can be accessed by web browsers running on desktop computers or tablet and mobile devices.

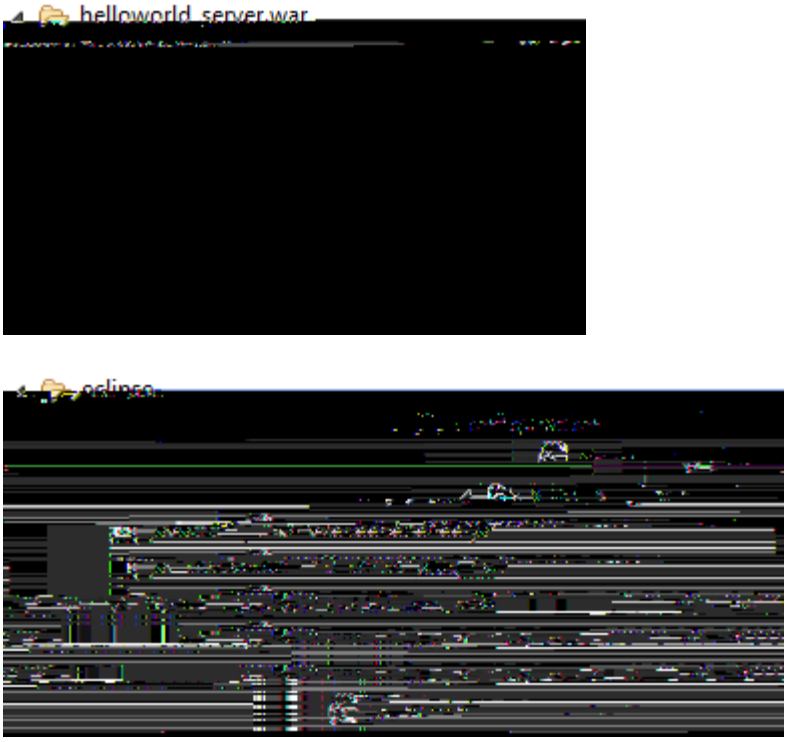


Figure 35. The organisation of the ÒHello WorldÓ server WAR file. The right side reveals the location of the config.ini file and the applicationÕs plugin files

The content and its organisation of the exported WAR files was not specifically designed for Scout applications. Rather, it is defined according to server-side Equinox. [40: See Appendix OSGi and Equinox for more information regarding server-side Equinox.], the typical setup for running Eclipse based server applications on a web server. Using file helloworld_server.war as a concrete example, we will first describe the general organisation of the WAR file. Then, we introduce individual artefacts of interest that are contained in this WAR file.

The explicit organisation of the server WAR file is shown in Figure 000. From the left hand side of the figure we can see that on the top level only folder WEB-INF exists in the WAR file. This folder contains all files and directories that are private to the web application. Inside, the web deployment descriptor file web.xml as well as the directories lib and eclipse are located. While the web.xml file and directory lib are standard for servlet based based applications. [41: See Appendix Java EE Basics for more information regarding servlets.], directory eclipse contains all necessary artefacts for servlet based Eclipse applications. [42: See Appendix OSGi and Equinox for more information regarding server-side Eclipse applications (server-side Equinox).]. Such as Eclipse Scout server applications.

On the right hand side of Figure 000 the eclipse specific content of the WAR file is shown. From top to bottom we find the configuration file config.ini introduced in Section Eclipse Configuration Files. In folder plugins the necessary plugins that constitute the eclipse application are located where the plugins are available in the form of JAR files. [43: JAR files contain a set of Java classes and associated resources. http://en.wikipedia.org/wiki/JAR_%28file_format%29.]. This includes plugins for servlet management, the eclipse platform including the servlet bridge, the scout framework parts and of course our ÒHello WorldÓ server and shared plugin. These ÒHello WorldÓ jar files exactly match with the plugin projects discussed in Section Create a new Project.

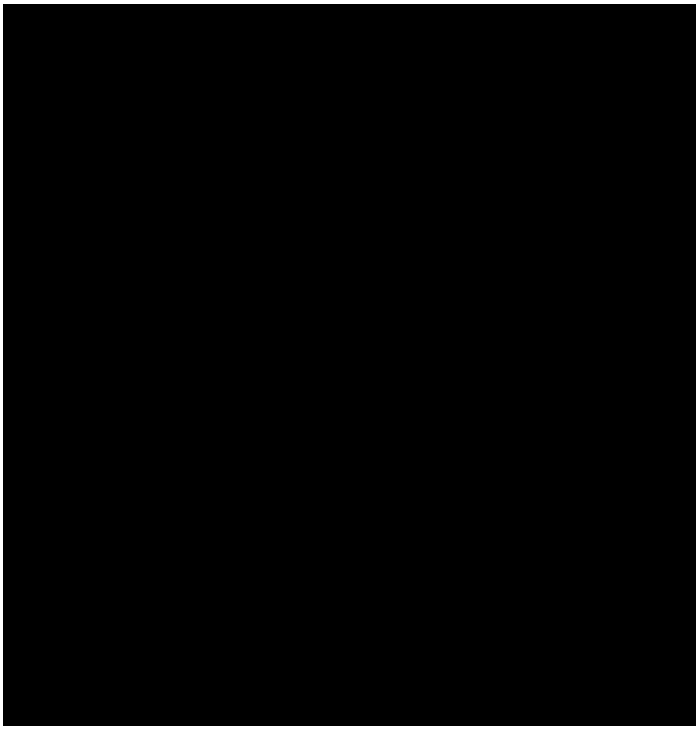


Figure 36. The content of the ÒHello WorldÓ server plugin contained in the helloworld_server.war file. The necessary files for the download page including the zipped client application are in the resources/html directory.

Listing 9. The configuration of the serverÕs resource servlet in the plugin.xml configuration file. The remaining content of the file has been omitted.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin>

Ê  <extension
Ê        name=""
Ê        point="org.eclipse.equinox.http.registry.servlets">
Ê     <servlet
Ê           alias="/"
Ê           class="org.eclipse.scout.rt.server.ResourceServlet">
Ê        <init-param
Ê              name="bundle-name"
Ê              value="org.eclipse.scout.helloworld.server">
Ê        </init-param>
Ê        <init-param
Ê              name="bundle-path"
Ê              value="/resources/html">
Ê        </init-param>
Ê     </servlet>
Ê  </extension>

</plugin>
```

In Figure 000 some of the content of the ÒHello WorldÓ server plugin is shown. The first thing to note is that the plugin file conforms to the JAR file format including a META-INF/MANIFEST.MF file and the directory tree containing the Java class files, as the DesktopService.class implemented in Section The Server Part of the ÒHello WorldÓ tutorial. In folder resources/html the necessary files for the download page shown in Figure 000 including the zipped desktop client are contained. To access this download page the Scout serverÕs resource servlet ResourceServlet is responsible. It is registered under the servlet registry as shown in Listing servlet registration. With setting "/" of the alias parameter the download page becomes available under the root path of the Scout server application. For the mapping to the contents resources/html the parameter bundle-path is used.

Figure 37. The ÒHello WorldÓ server plugin shown in the Eclipse package explorer. The files for the download page are located under resources/html.

Revisiting the ÒHello WorldÓ server plugin project in the Eclipse package explorer as shown in Figure 000, we can see how the plugin project elements are transformed and copied into the JAR file. Examples files are plugin.xml and MANIFEST.MF as well as static HTML content of the download page (files index.html and scout.gif). The zipped client is missing of course. It is assembled, zipped and added into the Scout server JAR file by the *Export Scout Project* wizard of the Scout SDK. In case you need to change/brand/amend the download page for the desktop client, you have now learned where to add and change the corresponding HTML files.

3.8. Deploying to Tomcat

In this section we will discuss two common pitfalls when working with the Scout IDE and Tomcat. The symptoms linked to these problems are Scout server applications that are not starting or Scout applications that fail to properly update.

In usual culprit behind Scout server applications that fail to start is a blocked port 8080. This setting can be created when we try to run both the Jetty web server inside the Scout SDK and the local Tomcat instance. In consequence, either Jetty or Tomcat is not able to bind to port 8080 at startup which makes

it impossible for a client to connect to the right server. To avoid such conflicts, make sure that you always stop the Scout server application in the Scout SDK (effectively killing Jetty) before you restart your Tomcat server. Alternatively, you can assign two different ports to your Jetty webserver and your Tomcat webserver.

To modify Jettyôs port number in the Scout SDK you have to update the corresponding properties in the config.ini files of the development products of your Scout server application and all client applications. In the Scout serverôs config.ini file the property is named org.eclipse.equinox.http.jetty.http.port, in the client config.ini files the relevant property is called server.url. To change the port number to 8081 for the ÒHello WorldÓ example in the Scout SDK you could use the following lines in the individual config.ini files.

Scout Server	org.eclipse.equinox.http.jetty.http.port=8081
Scout Desktop Client	server.url=http://localhost:8081/helloworld_server/process
Scout Ajax Server	server.url=http://localhost:8081/helloworld_server/ajax

The second pitfall is connected to a web application that seems to refuse to update to the content of a freshly generated WAR file. At times it seems that your changes to a deployed WAR file do not find their way to the application actually running. In many cases this is caused by a cached instance of the previous version of your application located in Tomcatôs working directory. To save yourself much frustration, it often helps just to clear Tomcatôs working directory and restart Tomcat. For this, you may follow the following procedure.

1. Stop the Tomcat web server
2. Go to folder work/Catalina/localhost
3. Verify that you are not in Tomcatôs webapps folder
4. Delete all files and directories in folder work/Catalina/localhost
5. Start the Tomcat web server

How you start and stop Tomcat depends on the platform you are running it. If you have installed Tomcat on a Windows box according to Appendix Apache Tomcat Installation it will be running as a service. This means that to stop the Tomcat web server you need to stop the corresponding Windows service. For starting and stopping Tomcat on Mac/Linux/Unix systems, you can use the command line script files startup.sh and shutdown.sh located in Tomcatôs subdirectory bin.

For those interested in more advanced aspects of Apache Tomcat we recommend the article ÒMore about the CatÓ by Chua Hock-Chuan. [44: More about the Cat: http://www.ntu.edu.sg/home/ehchua/programming/howto/Tomcat_More.html.].

4. Shared Components

In this chapter deals with the content of the shared plugin of any Scout application. As the name shared already indicates, this plugin contains code and resources that need to be available to both the Scout client and the server application.

The chapter starts with the internationalization of texts in Scout and icon resources. Then, the less visible components are introduced. These include permissions, code types, lookup calls and form data objects.

4.1. Texts / i18n / NLS Support

needs text

Existing Documentation

- concept wiki: http://wiki.eclipse.org/Scout/Concepts/Texts
- forum: http://www.eclipse.org/forums/index.php/t/319136/
- forum: http://www.eclipse.org/forums/index.php/t/326343/
- forum: overwriting texts provided by scout http://www.eclipse.org/forums/index.php/t/308273/
- forum: additional text provider service http://www.eclipse.org/forums/index.php/t/317565/
- forum: changing default language for scout apps http://www.eclipse.org/forums/index.php/t/367177/
- forum: export/import not possible: http://www.eclipse.org/forums/index.php/t/326320/
- forum: usage counts for text entries: http://www.eclipse.org/forums/index.php/t/261235/

4.2. Icons

needs text

Existing Documentation

- how-to wiki http://wiki.eclipse.org/Scout/HowTo/3.8/Add_an_icon
- how-to wiki http://wiki.eclipse.org/Scout/HowTo/3.8/Exchange_Default_Images

4.3. Code Types and Codes

Code types and codes are widely used in business applications. In general, any fixed set of named entities can be seen as a code type. Code types can be used to model the organisational structure of companies, to represent business units or to categorise or segment entities. Frequently, enumerations or enumerated types. [45: Enumerated type: http://en.wikipedia.org/wiki/Enumerated_type] are used as

synonyms for code types. The individual named entities in a code type are called codes in Scout.

Both code types and codes have associated names (translated texts) and IDs. As in the case of standard Java enumerations, Scout codes can also have associated values. A set of additional features enhances Scout code types over simple Java enumerations:

- Code types can be organized hierarchically
- Code types support multitenancy for individual codes
- Code types and codes can be accessed through a code service
- Codes can by added from external sources dynamically at runtime
- Codes are cached on both client and server side

The text below first introduces the basic features of code types and codes using a simple example with static codes. Then, hierarchical code types and the dynamic loading of codes from external sources is explained.

4.3.1. A Simple Example

As a simple example we assume that an event managing organization works with an application to plan events for customers. To distinguish public and private events it is natural to define a corresponding code type. Both the code type and all its elements will have an assigned ID and associated translated texts. See Listing Listing Code for a possible implementation of such a code type.

Listing 10. A code type with associated codes.

```
import org.eclipse.scout.rt.shared.TEXTS;
import org.eclipse.scout.rt.shared.services.common.code.AbstractCode;
import org.eclipse.scout.rt.shared.services.common.code.AbstractCodeType;

/**
Ê* @author mzi
Ê*/
public class EventTypeCodeType extends AbstractCodeType<Long, Long> {

Ê private static final long serialVersionUID = 1L;
Ê public static final Long ID = 10000L;

Ê public EventTypeCodeType() throws ProcessingException {
Ê   super();
Ê }

Ê @Override
Ê protected String getConfiguredText() {
Ê   return TEXTS.get("EventType");
Ê }
```

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

Ê @Order(10.0)
Ê public static class PublicCode extends AbstractCode<Long> {

Ê   private static final long serialVersionUID = 1L;
Ê   public static final Long ID = 10010L;

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

Ê   @Override
Ê   public Long getId() {
Ê     return ID;
Ê   }
Ê }

Ê @Order(20.0)
Ê public static class PrivateCode extends AbstractCode<Long> {

Ê   private static final long serialVersionUID = 1L;
Ê   public static final Long ID = 10020L;

Ê   @Override
Ê   protected String getConfiguredText() {
Ê     return TEXTS.get("Private");
Ê   }
```

The code type class and its codes shown in Listing Listing Code have been created using the creation wizards provided by the Scout SDK as described in Section [sec-wizard_code_type]. In Scout, code type classes are derived from class AbstractCodeType<CODE_TYPE_ID, CODE_ID>. In the provided example, both the code type ID and the code ID are typed with Long. The value of the event code type ID is assigned by Long ID = 10000L. The contained codes for public and private events are realized by the inner classes PublicCode and PrivateCode that are derived from Scout's AbstractCode<CODE_ID> class. Their individual IDs then have assigned the numbers 10010 and 10020 respectively. This pattern follows the convention to leave an ample number space between any two code type IDs. This space can then be used for the individual codes of a code type that need ID values as well.

In the above example the types for the ID values are defined by the generic parameter <Long>. And other classes from the package java.lang work well too. In fact, any Java class may be used as a key type for code types and codes as long as it satisfies the following requirements:

¥ Key types implement Serializable

¥ Key types correctly implement the equals and hashCode methods

¥ Key types are available in the Scout server and the client application.

Listing 11. A codes that is set to inactive.

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

Ê @Order(30.0)
Ê public static class ExternalCode extends AbstractCode<Long> {

Ê   private static final long serialVersionUID = 1L;
Ê   public static final Long ID = 10030L;

Ê   @Override
Ê   protected boolean getConfiguredActive() {
Ê     return false;
Ê   }

Ê   @Override
Ê   protected String getConfiguredText() {
Ê     return TEXTS.get("External");
```

To allow for language specific translations, the configuration method getConfiguredText is used for both the names of code types and codes. A frequently used code property is the active flag to mark obsolete codes. Setting individual codes to inactive is useful for codes that are still linked with existing data but should not longer be used when entering new data into the application. As shown in Listing Listing Code, codes can be marked inactive by returning false in method getConfiguredActive.

A set of additional code properties is available to control the appearance of individual codes. Clicking on an individual code in the Scout Explorer provides the list in the Scout Object Property view.

4.3.2. Hierarchical Code Types

To explain the definition and use of hierarchical codes we use the Industry Classification Benchmark as a concrete example. The Industry Classification Benchmark or ICB. [46: Industry Classification Benchmark (ICB): http://www.icbenchmark.com/] allows to hierarchically classify companies and organisations into industries, super sectors, sectors and sub sectors. Each organizational level has a unique number assigned and a name. This setup can easily be transferred to a hierarchical Scout code type.

Listing 12. A hierarchical code type for the Industry Classification Benchmark.

```
import org.eclipse.scout.commons.annotations.Order;
import org.eclipse.scout.commons.exception.ProcessingException;
import org.eclipse.scout.rt.shared.TEXTS;
import org.eclipse.scout.rt.shared.services.common.code.AbstractCode;
import org.eclipse.scout.rt.shared.services.common.code.AbstractCodeType;

public class IndustryICBCodeType extends AbstractCodeType<Long, Long> {

Ê private static final long serialVersionUID = 1L;
Ê public static final Long ID = 0000L;

Ê public IndustryICBCodeType() throws ProcessingException {
Ê   super();
Ê }

Ê @Override
Ê protected boolean getConfiguredIsHierarchy() {
Ê   return true;
Ê }

Ê @Override
Ê protected String getConfiguredText() {
Ê         public Long getId() {
```

Figure 38. A hierarchical code type class shown in the Scout SDK.

The corresponding code for the ICB code type is provided in Listing Listing Code execLoadCodes where its hierarchical nature is reflected by method getConfiguredIsHierarchy. The actual hierarchical codes are then implemented as nested inner code classes that are derived from Scout class AbstractCode. See

Figure 000 for a screenshot of the partially expanded ICB code type class in the Scout SDK.

4.3.3. Loading Codes Dynamically

Although codes types remain mostly static in their nature, they do evolve over time in any real world application. Requiring that all codes are statically defined in the application's code base would result in the necessity to update the code base for every change of a code required by the business. Clearly, such a setup is not sustainable and this is why Scout allows to dynamically update the set of codes contained in a code type.

Listing 13. Adding codes dynamically in method execLoadCodes.

```
Ê*/
public class EventTypeCodeType extends AbstractCodeType<Long, Long> {

Ê private static final long serialVersionUID = 1L;
Ê public static final Long ID = 10000L;

Ê public EventTypeCodeType() throws ProcessingException {
Ê   super();
Ê }

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

Ê @Override
Ê public Long getId() {
Ê   return ID;
Ê }

Ê @Order(10.0)
Ê public static class PublicCode extends AbstractCode<Long> {

Ê   private static final long serialVersionUID = 1L;
Ê   public static final Long ID = 10010L;

Ê   @Override
Ê   protected String getConfiguredText() {
Ê     return TEXTS.get("Public");
Ê   }
```

At startup of a Scout application the codes of all defined code types are loaded into memory. For this, Scout internally calls method loadCodes for each code type class derived from class AbstractCodeType. In this method Scout first creates objects representing the statically defined codes and then

dynamically adds additional codes in method execLoadCodes. By overriding this method, a code type class can dynamically add codes from any external sources. An example for the dynamic loading of codes is provided in Listing Listing Load Codes Dynamically. Please note that this code snipped only illustrates the principle and does therefore not access any external web services or databases.

As codes have IDs and can be defined both statically in code and dynamically from external data, conflicting definitions of codes are inevitable. In the Scout framework these conflicts are resolved in method execOverrideCode. In the default implementation provided by class AbstractCodeType, priority is given to the dynamically defined code. Only attributes that are undefined for the dynamic code are copied from the static code definition. This logic can be changed for any code type by simply overriding method execOverrideCode with the desired behaviour. In the example provided in Listing Listing Load Codes Dynamically the code with ID Color.YELLOW is defined both statically and dynamically. As a result, the translated text for key "YellowDynamic" is shown in the user interface as it has priority over the statically defined text key "Yellow".

To have access to all codes at runtime, Scout provides the convenience accessor CODES. This accessor encapsulates the access to the ICodeService and provides a number of useful methods. At startup, all codes are loaded and cached in the applications client session using method getAllCodeTypes. And for a class MyCodeType, all its codes can be retrieved by CODES.getCodeType(MyCodeType.class).getCodes().

4.4. Lookup Calls and Services

Lookup calls are are used to look up lists of lookup rows in the form of key-text pairs. The list of lookup rows returned is usually defined by some search criteria. When the look up is triggered by a key, only a single element is returned. And when a string is provided as a search criteria, the returned list typically contains the lookup rows that contain the given search text as a substring.

For lookup calls two main use cases exist. In the first case, the lookup data is locally available, not too large and can be kept in memory. In this situation, the lookup data can be directly created in the call itself. As an example, you may consider a lookup call where the lookup data is based on a code type. In the other case, the lookup data is dynamic in nature, the amount of data is large and needs to be read from some external source, such as a database or a web service. To access large amounts of external data, a lookup call typically invokes a so called lookup service that is providing the necessary data. This is exactly the scenario that was used in the ÒMy ContactsÓ application of the bookÕs first part in Section [sec-adding_the_smartfield]. In the contact form of that application, a company smart field is used to let the user select a specific entry from a list of companies. And in turn, this smart field uses a company lookup call that is backed by a company lookup service. This lookup service then accesses a database to create the list of companies required for the company smart field.

in the text below

1. general aspects:
    - ¥ getDataBy{Key|Text|Rec?|All}
    - ¥ additional properties as constraints to the result set: master field

- ¥ for a specific implementation additional properties can be added with getters/setters ? yes as they are passed to services as bind variables

2. local lookup calls in the client plugin
3. lookup calls in the shared plugin
4. lookup services are only mentioned here for completeness. full discussion in book part 3 for scout server? or not? probably needs to be here É

Listing 14. A simple local lookup call defining it̊s entries in method execCreateLookupRows.

```
import org.eclipse.scout.rt.shared.TEXTS;
import org.eclipse.scout.rt.shared.services.lookup.LocalLookupCall;
import org.eclipse.scout.rt.shared.services.lookup.LookupRow;

/**
Ê* @author mzi
Ê*/
public class FontStyleLookupCall extends LocalLookupCall<Integer> {

Ê private static final long serialVersionUID = 1L;

Ê @Override
Ê protected List<LookupRow<Integer>> execCreateLookupRows() throws ProcessingException {
Ê   ArrayList<LookupRow<Integer>> rows = new ArrayList<LookupRow<Integer>>();
```

Listing 15. A local code lookup call. This lookup removes inactive codes in the lookup data.

```
import org.eclipse.scout.commons.exception.ProcessingException;
import org.eclipse.scout.rt.shared.services.common.code.ICode;
import org.eclipse.scout.rt.shared.services.lookup.CodeLookupCall;
import org.eclipse.scout.rt.shared.services.lookup.ICodeLookupCallVisitor;
import org.eclipse.scout.rt.shared.services.lookup.ILookupRow;
import org.eclipsescout.demo.widgets.shared.services.code.EventTypeCodeType;

/**
Ê* @author mzi
Ê*/
public class EventTypeLookupCall extends CodeLookupCall<Long> {

Ê private static final long serialVersionUID = 1L;

Ê /**
Ê  * Default constructor that wires this lookup call to the event type code type.
Ê  */
Ê public EventTypeLookupCall() {
Ê   super(EventTypeCodeType.class);
Ê   setFilter(new LookupVisitor());
Ê }
```

TODO: fix Figure 000: MyLookupService extends AbstractLookupService and implements IMyLookupService that extends ILookupService which

Figure 39. Implementing a lookup call with a corresponding lookup service. Scout framework components are shown in orange, user code in blue.

Existing Documentation

- presentation: http://wiki.eclipse.org/images/c/c9/20111102_EclipseConEurope2011-EclipseScout-DiscoverThePotential.pdf
- tutorial: https://wiki.eclipse.org/Scout/Tutorial/Lookup_Calls
- tutorial minicrm: https://wiki.eclipse.org/Scout/Tutorial/4.0/Minicrm/Lookup_Calls_and_Lookup_Services
- concept wiki: http://wiki.eclipse.org/Scout/Concepts/LookupCall
- concept wiki: http://wiki.eclipse.org/Scout/Concepts/Lookup_Service
- forum: http://www.eclipse.org/forums/index.php/t/279108/
- javadoc lookupcall: https://eclipse.googlesource.com/scout/org.eclipse.scout.rt/+/Luna_RC3/org.eclipse.scout.rt.shared/src/org/eclipse/scout/rt/shared/services/lookup/LookupCall.java
- javadoc lookuprow: https://eclipse.googlesource.com/scout/org.eclipse.scout.rt/+/Luna_RC3/org.eclipse.scout.rt.shared/src/org/eclipse/scout/rt/shared/services/lookup/LookupRow.java
- javadoc codelookupcall

https://eclipse.googlesource.com/scout/org.eclipse.scout.rt/+/Luna_RC3/org.eclipse.scout.rt.shared/src/org/eclipse/scout/rt/shared/services/lookup/CodeLookupCall.java

- javadoc locallookupcall
  https://eclipse.googlesource.com/scout/org.eclipse.scout.rt/+/Luna_RC3/org.eclipse.scout.rt.shared/src/org/eclipse/scout/rt/shared/services/lookup/LocalLookupCall.java
- javadoc ilookupservice
  https://eclipse.googlesource.com/scout/org.eclipse.scout.rt/+/Luna_RC3/org.eclipse.scout.rt.shared/src/org/eclipse/scout/rt/shared/services/lookup/ILookupService.java
- javadoc lookupservices
  https://eclipse.googlesource.com/scout/org.eclipse.scout.rt/+/Luna_RC3/org.eclipse.scout.rt.server/src/org/eclipse/scout/rt/server/services/lookup/
- lookup service
- sqllookup service

4.5. Permissions

needs text, topic is relevant for client, server, and security. what to present where to be decided

```
Existing Documentation
```

- how-to wiki: http://wiki.eclipse.org/Scout/HowTo/3.8/Create_Permissions
- concept wiki: http://wiki.eclipse.org/Scout/Concepts/Permission
- forum: http://www.eclipse.org/forums/index.php/t/243966/

4.6. Form Data Objects

needs text, explain that form data objects are data transfer objects

Existing Documentation

- form data/dto http://www.eclipse.org/forums/index.php/t/169334/

4.6.1. Data Binding

needs text, this is about Form Data Export and Import

4.6.2. Automatic Updates by the Scout SDK

needs text

4.6.3. Manual Form Data Updates

needs text

5. Client components

needs text

5.1. Client Model

needs text

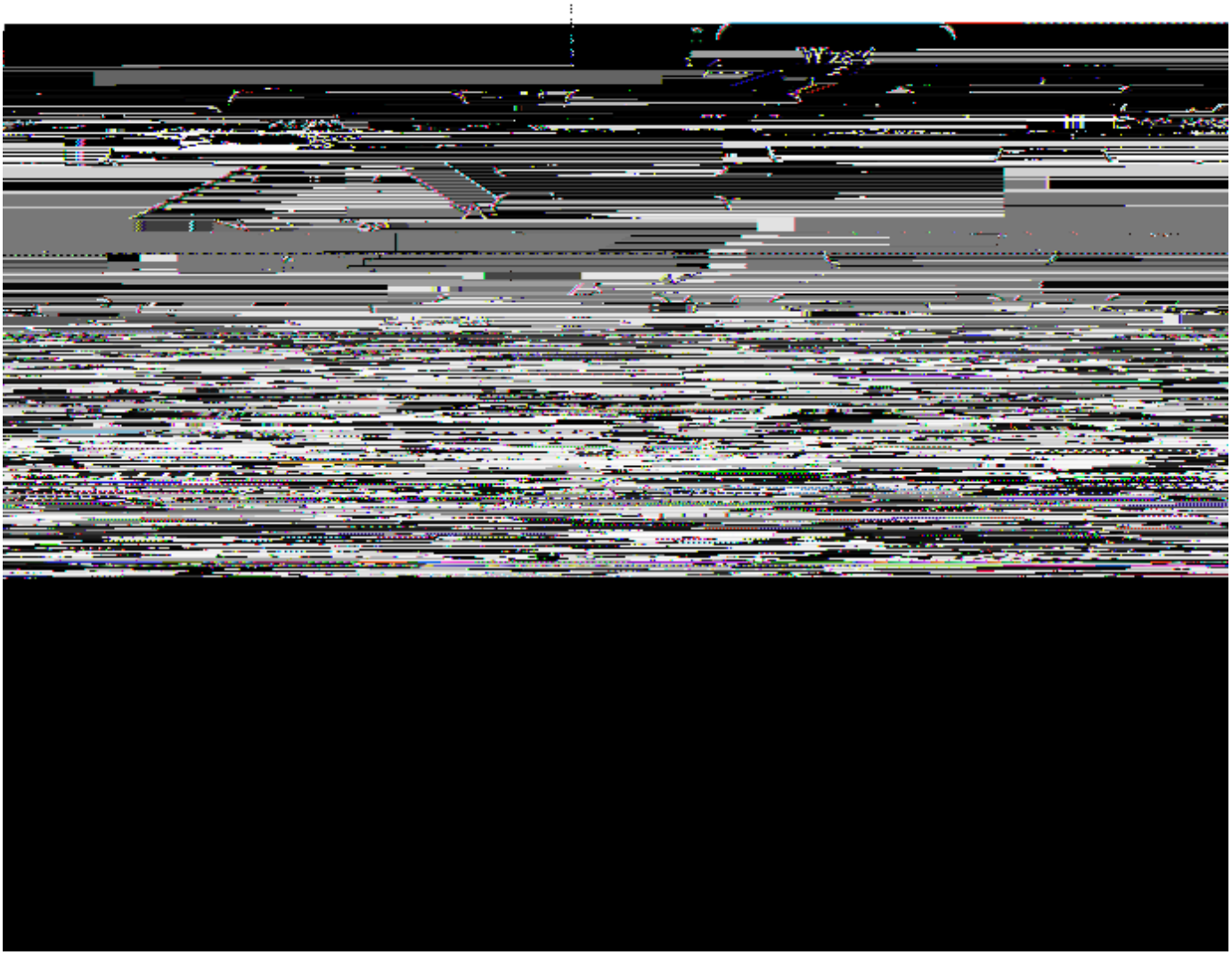


Figure 40. A class diagram for the Scoutò s client model

5.2. Splash Screen

needs text

5.3. Login Box

needs text

5.4. Client Session

needs text

5.5. Desktop

needs text

Existing Documentation

- concept wiki http://wiki.eclipse.org/Scout/Concepts/Desktop

5.5.1. Info Dialog

needs text

5.5.2. Toolbar

needs text

Existing Documentation

- forum: feature request http://www.eclipse.org/forums/index.php/t/366440/
- concept wiki http://wiki.eclipse.org/Scout/Concepts/Tool

5.5.3. Status Line

needs text

5.6. Menus

needs text

Existing Documentation

- concept wiki http://wiki.eclipse.org/Scout/Concepts/Menu
- forum: hard coded swt menues http://www.eclipse.org/forums/index.php/t/236071/. is this still an issue with scout kepler?

5.7. Outlines

needs text

Existing Documentation

- concept wiki http://wiki.eclipse.org/Scout/Concepts/Outline

5.8. Tools

needs text

5.9. Forms

needs text

Existing Documentation

- concept wiki http://wiki.eclipse.org/Scout/Concepts/Form
- concept wiki form handlerhttp://wiki.eclipse.org/Scout/Concepts/Form_Handler
- how-to wiki http://wiki.eclipse.org/Scout/HowTo/3.8/Open_a_Form_in_a_View
- forum: layout manager http://www.eclipse.org/forums/index.php/t/404048/
- forum: life cycle http://www.eclipse.org/forums/index.php/t/369890/
- form validation

5.10. Form Fields

needs text

Every Scout form contains one or several form fields. Form fields therefor represent the basic building blocks of a forms content. Depending on their nature, form fields can display information, accept user input or act as container holding inner form fields. As such container fields can hold inner container fields it is possible to create forms that meet compolex requirements.

Existing Documentation

- concept wiki (links) http://wiki.eclipse.org/Scout/Concepts/Client_Plug-In#Form_fields
- concept wiki screenshots http://wiki.eclipse.org/Scout/Concepts/Field

5.10.1. Common Aspects

needs text

Existing Documentation

- forum: label position http://www.eclipse.org/forums/index.php/t/369109/
- model component
- ui component
- extension point registration
- model
- label
- value
- exec methods
- field validation

5.11. Trees

needs text

- tree nodes
- tree form
- tree field

5.12. Pages

needs text

Existing Documentation

- how-to wiki: http://wiki.eclipse.org/Scout/HowTo/3.8/Display_images_in_a_table_page
- concept wiki: http://wiki.eclipse.org/Scout/Concepts/Page
- forum: pages linking to forms http://www.eclipse.org/forums/index.php/t/367595/
- forum: changing page icons http://www.eclipse.org/forums/index.php/t/262151/
- page with table
- page with nodes

5.13. Search Forms

needs text

Existing Documentation

- ¥ forum: position of search form http://www.eclipse.org/forums/index.php/t/353895/
- ¥ forum: statement builder stuff http://www.eclipse.org/forums/index.php/t/165805/

5.14. Tables

needs text

Existing Documentation

- ¥ forum: editable column http://www.eclipse.org/forums/index.php/t/220019/
- ¥ forum: default visibility of columns http://www.eclipse.org/forums/index.php/t/166052/
- ¥ forum: row deletion http://www.eclipse.org/forums/index.php/t/210744/
- ¥ context menues
- ¥ editable tables
- ¥ column types

5.14.1. Image Columns

needs text

Existing Documentation

- ¥ forum: http://www.eclipse.org/forums/index.php/t/369626/

5.14.2. HTML inside Table Cells

needs text

Existing Documentation

- ¥ forum: http://www.eclipse.org/forums/index.php/t/370714/
- ¥ forum: summary row http://www.eclipse.org/forums/index.php/t/235749/

5.14.3. Table Status Bar

nees text

Existing Documentation

- ¥ forum: http://www.eclipse.org/forums/index.php/t/367326/

5.14.4. Injecting Columns at Runtime

needs text

Existing Documentation

- forum: http://www.eclipse.org/forums/index.php/t/364715/
- forum : dynamic columns http://www.eclipse.org/forums/index.php/t/216731/

5.15. Workflows and Wizards

Needs text

Existing Documentation

- concept wiki http://wiki.eclipse.org/Scout/Concepts/Wizard
- forum: http://www.eclipse.org/forums/index.php/t/391607/
- forum: http://www.eclipse.org/forums/index.php/t/382579/
- forum: http://www.eclipse.org/forums/index.php/t/366971/

6. The Widgets Demo Application

This chapter introduces the "Scout Widgets Demo App". The purpose of this demo application is to present Scout's most commonly used UI widgets. Therefore, the application does not contain any business logic but only serves as a hands-on reference on how to use and configure Scout UI widgets.

It is interesting to note that the widget demo application works out-of-the-box with any of the currently supported UI technologies. This means that with the same code base the widget demo application is capable to run as a native desktop application, as a web application in browsers, as well as on touch-enabled mobile devices. Comparing individual aspects of the the desktop application with its mobile/tablet version reveals the default strategies used to map desktop widgets to the substantially different usage/form factor found on mobile devices. Please observe that for the complete widget application only a handful of lines of code actually depend on a specific UI technology. The interested reader can easily verify this by searching the application's code for the occurrences of class UserAgentUtility.

In the text below the organisation of the widget demo application is first described in Section The User Interface. And in Section Client Only Architecture, the setup in the form of a Scout client only application is explained.

6.1. The User Interface

The application is organized into separate outlines for thematic groups of widgets. Each of the application's outline then presents a list of widgets in a navigation tree. This is shown in Figure 000 for the *Simple Widgets* outline that contains examples for simple UI widgets such as label fields or string fields.

Figure 41. The "Scout Widgets Demo App". The widgets demo application features examples of the most commonly used Scout widgets. For each widget shown, example use cases are presented in a individual form. As shown in the screenshot, the corresponding Scout source code is made available via the View Source on GitHub *context menu.*

For each UI widget a corresponding example form presents a number of typical use cases and configuration options. The example forms are designed to be independent from each other. It should therefore be possible to read and understand the source code of each example form with minimal effort. Via the Open in Dialog É context menu the content of the view is displayed in a modal scout form. As shown in Figure 000, the complete source code for the selected form can be accessd via the View Source on GitHub context menu.

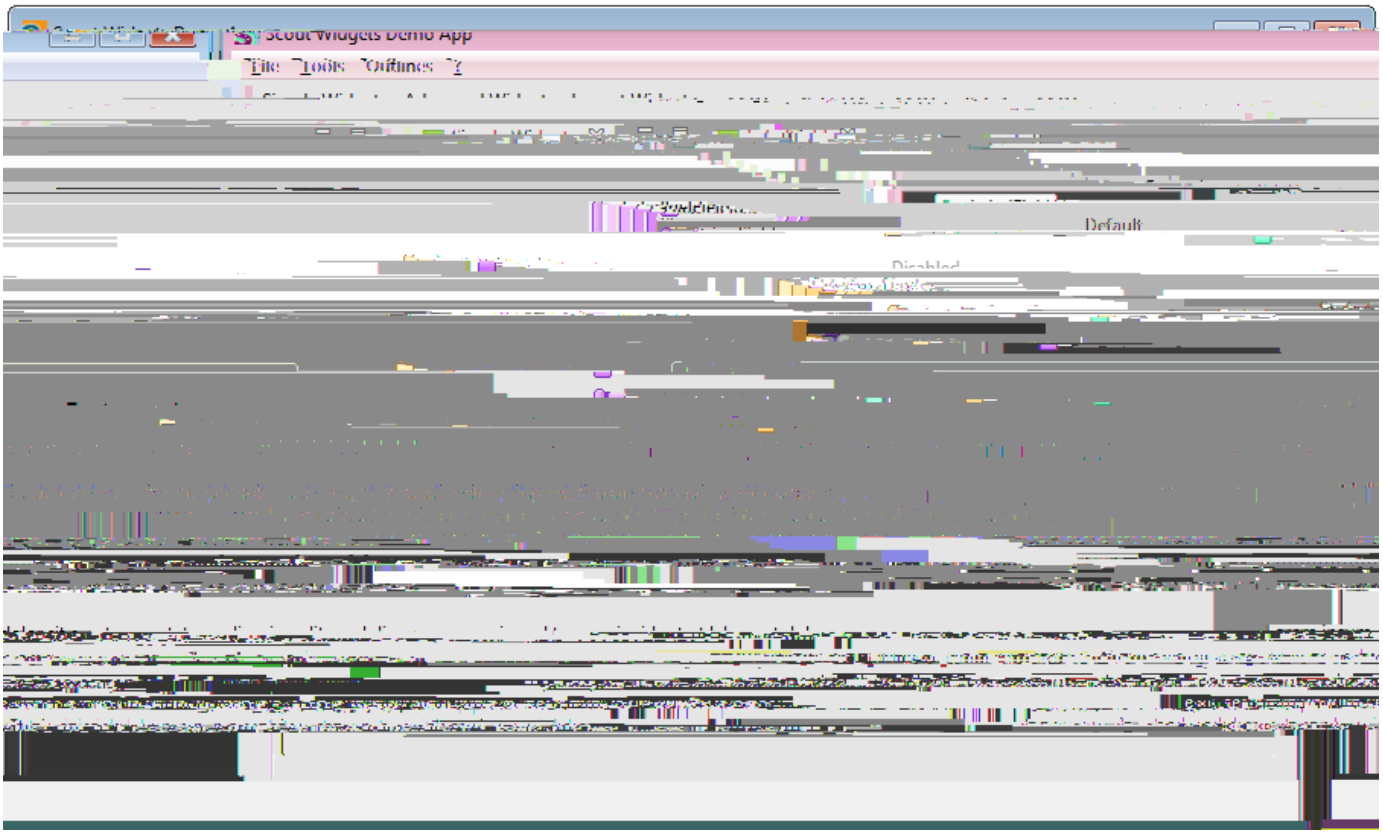
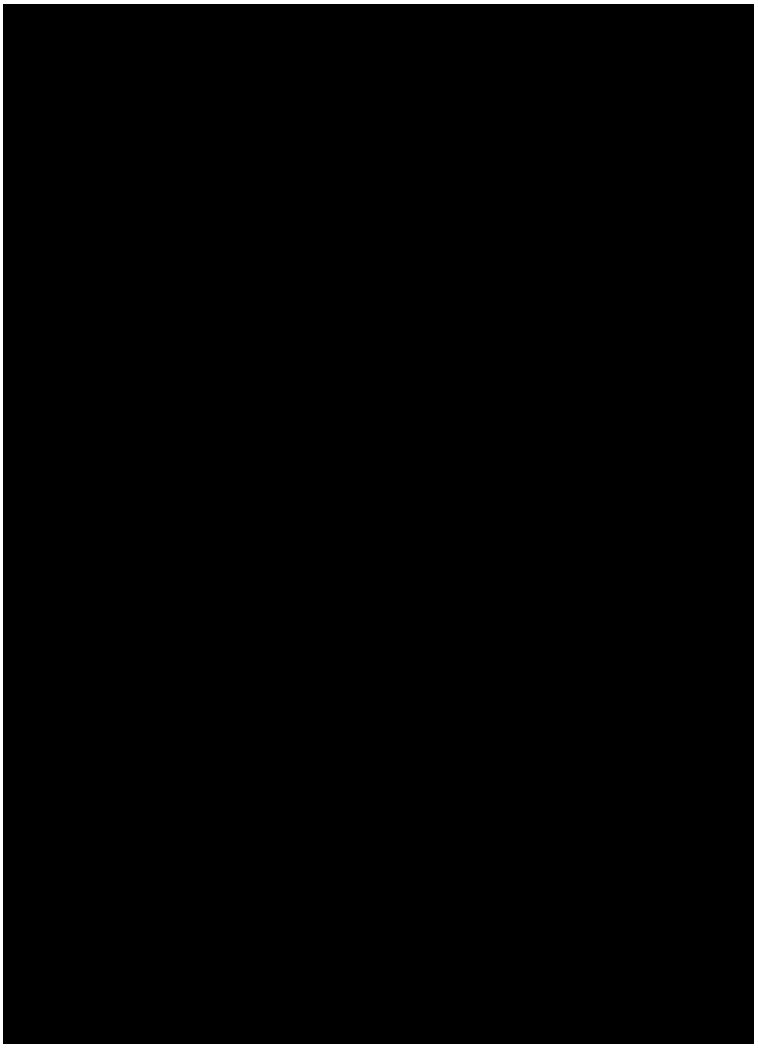
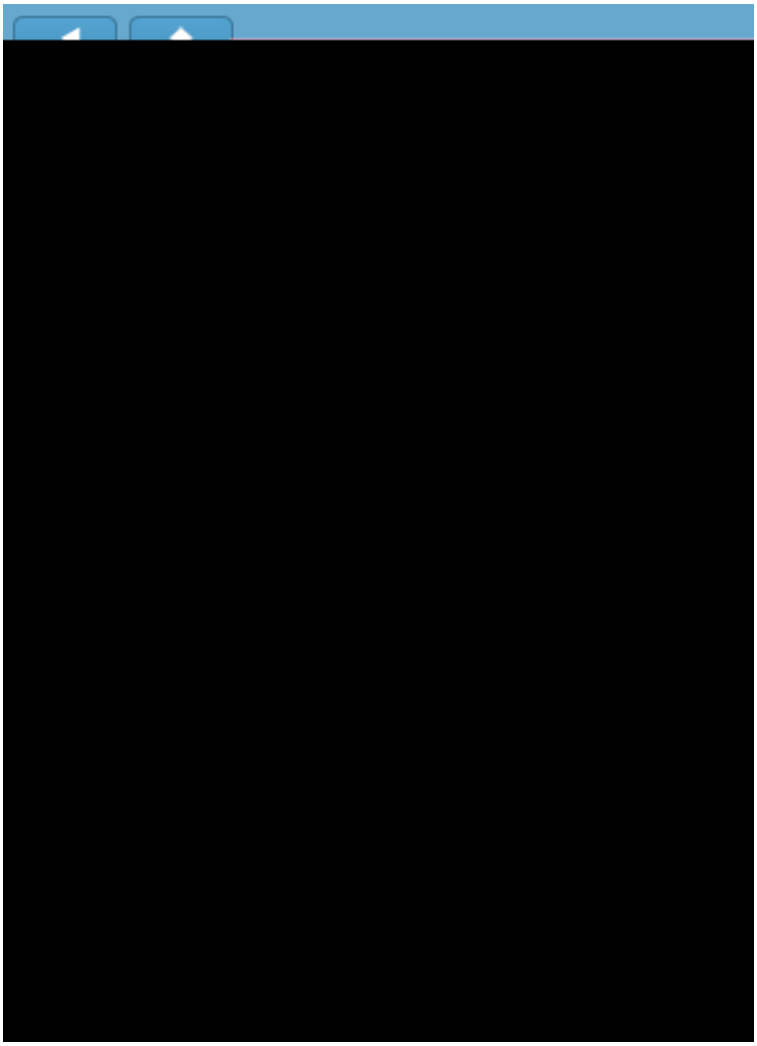


Figure 42. The "Scout Widgets Demo App" running as an SWT desktop application.

Figure 43. The "Scout Widgets Demo App" running in the browser.

The widget demo application can also be run as a native SWT desktop application. This is shown in Figure 000. And the exact same application also runs in a browser as a web application shown in Figure 000.





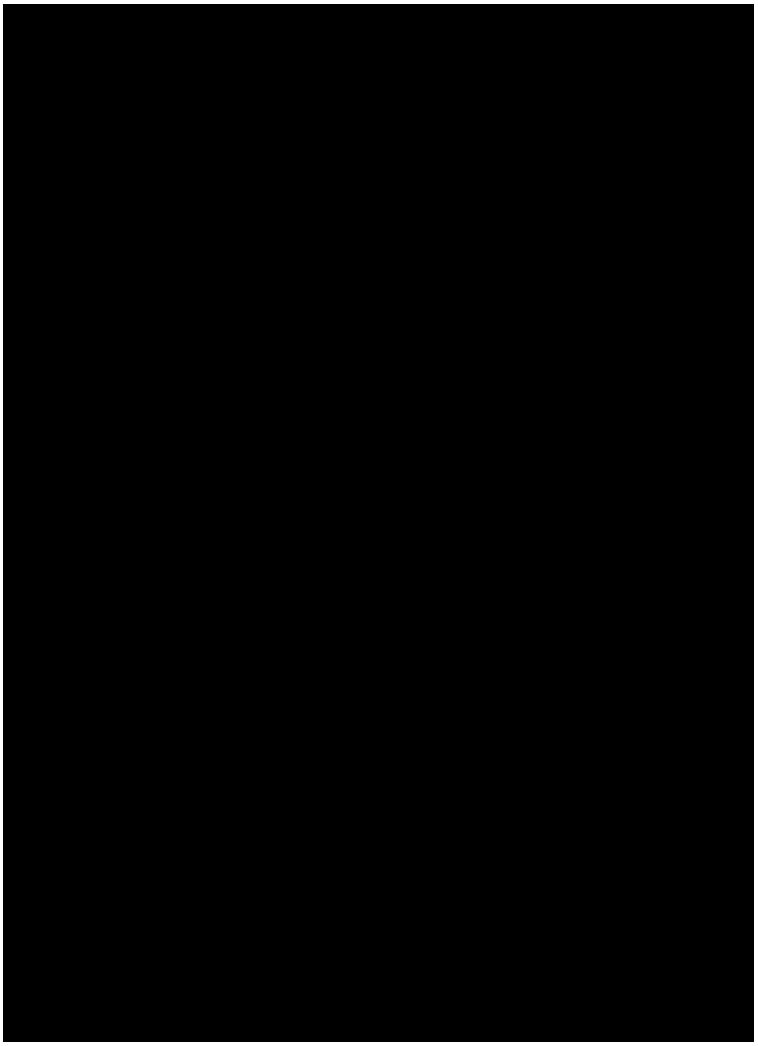


Figure 44. The "Scout Widgets Demo App" running on a mobile device. On the left, all outlines of the application are displayed on the home screen. The screens shown in the middle allows to navigate through ÒSimple WidgetsÓ outline while the selected ÒStringFieldÓ form is shown on the right.

In Figure 000, the content of the widget demo application is shown on a mobile device. On the home screen of the mobile application, the available outlines are presented. When the user selects a specific outline, the associated navigation tree is then shown as a scrollable list. Finally, when the user selects a specific widget, the associated example form is shown.

6.2. Client Only Architecture

As the sole purpose of the widget demo application is to demonstrate the usage of the UI elements provided by the Scout framework, no server side data access and/or business logic is required. Therefore, the widget demo application has been designed as a client only application. As a side effect, the architecture of this widget application may also be used as a template to build similar client only applications with the Scout framework. It is important to note that this architecture can only be recommended for very simple applications. For more complex software packages, for example personal digital archives or private accounting software, it is not recommended to implement the business logic and the access to the database in the applicationÕs client plugin. A much cleaner

approach is to take advantage of the offline support provided by the Scout framework. As in typical Scout client server projects, the presentation logic can be implemented in the client plugin and business logic and access to a (local) database are located in the server plugin. A detailed description of this setup including step-by-step instructions is available on the Scout Wiki. [47: Standalone Client with DB Access: http://wiki.eclipse.org/Scout/HowTo/Create_a_Standalone_Client_with_DB_Access]. In the text below the client only setup of the widget demo application is explained by looking at the main differences to the setup of the ÒHello WorldÓ application introduced in Chapter ÒHello WorldÓ Tutorial.

Listing 16. The setup of the client session in the Scout widget demo application.

```
import org.eclipse.scout.service.SERVICES;
import org.eclipsescout.demo.widgets.client.ui.desktop.Desktop;

public class ClientSession extends AbstractClientSession {

Ê private static IScoutLogger logger = ScoutLogManager.getLogger(ClientSession.class);
Ê private String m_product;
Ê private boolean m_footless;
```

The most obvious difference of the widget demo application to the ÒHello WorldÓ client server example is the missing server (plugin). Consequently, the widget demo application also does not need a service tunnel to handle client server communication. As the setup of this service tunnel is typically initiated in method execLoadSession of the clientÕs ClientSession class, the setup of the service tunnel has been commented out in the widget demo application according to Listing Listing LabelField.

The next difference between typical Scout applications and the widget demo lies in the handling of code types and codes. Accessing code types and codes is the responsibility of the serverÕs CodeService class in Scout applications. As the widget demo does not have a server but we still want to work with codes and code types, a LocalCodeService class has been added to the clientÕs plugin.

The last difference of note between the ÒHello WorldÓ example and the widget demo application lies in the implementation of the form handler classes. In the desktop formÕs view handler of the ÒHello WorldÓ application the data to be displayed is retrieved via the serverÕs DesktopService as shown in Listing Listing ViewHandler in DesktopForm. As the forms of the widget demo application do not load/persist any data from/to a server, no such logic is required in the form handlers and the corresponding classes remain empty.

7. Simple Widgets

This chapter presents the most commonly used Scout widgets based on the widget demo application introduced above. For each widget the most frequently observed use cases are presented and illustrated with the widget specific example forms. And exemplary code snippets taken from the widget application help to close the loop to the actual implementation of these widgets in the applications source code.

7.1. Label Fields

Label fields are used to place a read only text anywhere in a Scout form. As shown in the configuration section of Figure 000 such texts can occupy the typical area assigned to a label field on the left, the area typically assigned to data entry on the right or on both sides. The form shown in Figure 000 is implemented in class LabelFieldForm of the Scout widget application.

Scout fields and example use cases.

```
ÊIn the examples section of the form the standard usage of label fields is shown.
ÊTo display text over the whole width of a column or in the area right to the label use
method [java]+setValue+ as shown in the configuration section of the form.
image::{imgsdir}/labelfield.png[]
```

To set the label text in the default area method getConfiguredLabel is used. If the label text is too long for the reserved area the label gets truncated with trailing É. The complete label text is then shown in a tooltip.

As for any form field derived from AbstractFormField, a number of getConfigured* styling properties exist for label fields. Most of these styling properties can directly be set in the Scout Object Properties view. A subset of these properties are shown in the list below.

- Label Position
- Label Horizontal Alignment
- Label Foreground Color
- Label Background Color
- Label Font

Listing 17. A simple LabelField.

```
Ê    @Order(10.0)
Ê    public class LabelField extends AbstractLabelField {

Ê      @Override
Ê      protected String getConfiguredLabel() {
Ê        return TEXTS.get("Default");
Ê      }
Ê    }
```

To display text in the right hand area an empty string can be used as label text and the text for the value area of the label field can be set with method setValue in method execInitField according to Listing [lst-field.label.value].

Listing 18. A label field displaying multi-line text that covers the whole width of a column.

```
Ê    @Order(40.0)
Ê    public class VeryLongLabelTextField extends AbstractLabelField {

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

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

Ê      @Override
Ê      protected boolean getConfiguredWrapText() {
Ê        return true;
Ê      }

Ê      @Override
Ê      protected void execInitField() throws ProcessingException {
Ê        if (UserAgentUtility.isSwingUi()) {
Ê          setValue("<html>" + TEXTS.get("Lorem") + "</html>");
Ê        }
Ê        else {
Ê          setValue(TEXTS.get("Lorem"));
Ê        }
Ê      }
Ê    }
```

To display multiline text across both the label and the value area a combination of label field properties has to be used. See Listing Listing LabelField Multiline for the configuration used in the last label field shown in Figure 000.

7.2. String Fields

String fields are used to enter simple text strings. In addition, string fields are also useful to enter multiline text or capture masked input. The form shown in Figure 000 is implemented in class StringFieldForm of the Scout widget application.

String fields and example use cases.

```
ÊText content shown in disabled fields can be read and copied to the clipboard, but not
edited.
ÊIn multi line string fields the text can be displayed as entered or the text may be
wrapped to fit into the available column with.
image::{imgsdir}/stringfield.png[]
```

Some of the most typical use cases for string fields are represented in the examples section of Figure 000. In the configuration section of the form a multiline text field is shown. As in the case of the label text, font and color styling possibilities are available for the text shown in a string field. Clicking on the [!Sample Format!] button and on the [!Sample Content!] button prefills some of the fields to create the effect shown in Figure 000.

In the case of multi line string fields, the field can be configured to either display text lines as they have been entered or wrapped to fit into the available width of the field. This property can be set with the string fieldÕs method getConfiguredWrapText or dynamically with method setWrapText. Please note that changing this property dynamically at runtime currently only works with the SWT and the Swing rendering components.

(AbstractStringField,String Field

Listing 19. A masked string field.

```
Ê          this.setErrorStatus("\"" + rawValue + "\" " + TEXTS.get("NoColor"));
Ê        }

Ê        return rawValue;
Ê      }
Ê    }

Ê    @Order(110.0)
Ê    public class BackgroundColorField extends AbstractStringField {

Ê      @Override
Ê      protected String getConfiguredLabel() {
Ê        return TEXTS.get("BackgroundColor");
```

Additional use cases for the string field are shown in the right column of the configuration section of the string field demo form. Specific string fields are located for the default case, a string field only accepting upper case letters, and a masked field. To code to represent the masked string field is provided in Listing Listing masked string field. Independent of what the user is typing the masked field keeps the entered content visually hidden. In contrast to all other forms of string fields, the content cannot be copied to the system clipboard. It can only be accessed programmatically with the getValue method of the string field.

String fields also allow for the configuration of the maximum length of the text that can be entered into the field. In its default configuration, a maximum number of 4'000 characters can be entered into a string field. Using method setMaxLength this limit can be updated dynamically at runtime. Alternatively, this limit can also be set in method getConfiguredMaxLength.

7.3. Number Fields

For entering numbers, Scout provides three different fields. Depending on the valid range of numbers that may be entered, an integer field, a long field or a big integer field best matches the given use case. These fields are represented by the classes AbstractIntegerField, AbstractLongField and AbstractBigIntegerField, each one of them extinding class AbstractNumberField.

Number fields and example use cases.

```
ÊDistinct number fields are available for [java]+Integer+, a [java]+Long+ or a
[java]+BigInteger+ value classes.
image::{imgsdir}/numberfield.png[]
```

The form NumberFieldsForm of the Scout widget application shown in Figure 000 contains examples for all three number field types. Separate buttons are available to demonstrate the use of the minimum and the maximum value a number field can hold. In the case of the *BigInteger* field, arbitrarily large or small values may be entered. See Listing Listing Integer field for the code corresponding to the input integer field in the configuration section of the example form.

(AbstractIntegerField,Integer Field

Listing 20. A simple Integer field.

```
Ê    }

Ê    @Order(10.0)
Ê    public class InputField extends AbstractIntegerField {

Ê      @Override
Ê      protected String getConfiguredLabel() {
Ê        return TEXTS.get("IntegerFieldInput");
```

As already indicated by the class names of the three number fields, the fields serve to enter values that fit into the ranges defined by the Java classes Integer, Long and BigInteger. To further restrict the bounds of valid numbers you may use the methods getConfiguredMinValue and getConfiguredMaxValue. The effect of setting such bounds can be tested by entering values into the *Minimum Value* field and the *Maximum Value* field of the example form. If, for example, a minimum value of 0 is entered in the *Minimum Value* field and the user tries to enter the value -1 into one of the input fields, an error marker becomes visible on the input field. A tooltip with the text `"The value is too small; must be between 0 É`" further explains the issue to the user. In such a case the value

entered into the user interface is not propagated to the number field's value. This is why the read only *getValue()* field is not updated in such a case.

To textually format the entered numbers the grouping number field property can be used. In the example form the *Grouping* checkbox can be used to control this property. Ticking/unticking this checkbox will affect the three number input fields in the configuration section of the example form.

More extensive options to specify the formatting of the numbers is provided by the method setFormat of class AbstractNumberField. Method setFormat is accepting an argument of the Java class DecimalFormat. To demonstrate an example for such a format click on [!Sample Format!] button in the example form. For more information please consult the Javadoc for class DecimalFormat.

7.4. Decimal Fields

Scout provides two different form fields for entering decimal values. Depending on the required precision and range of values to be entered a double field or a big decimal field can be used. The two field types are represented by Scout's classes AbstractDoubleField and AbstractBigDecimalField and can hold values of the Java types Double and BigDecimal respectively. Both the double and the big decimal field extend class AbstractDecimalField which in turn extends class AbstractNumberField.

Decimal fields and example use cases.

```
ÊDistinct number fields are available for the [java]+Double+ and [java]+BigDecimal+ value
classes.
image::{imgsdir}/decimalfield.png[]
```

The example form shown in Figure 000 demonstrates the usage and some of the available options to configure Scout's decimal fields. This example form is defined in class DecimalFieldsForm of the Scout widget application.

(AbstractDoubleField,Double Field

Listing 21. A styled decimal field holding Java Double values. Negative values are shown in red

```
Ê    }

Ê    @Order(50.0)
Ê    public class StyledField extends AbstractDoubleField {

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

Ê      @Override
Ê      protected void execChangedValue() throws ProcessingException {
Ê        if (getValue() < 0) {
Ê          setForegroundColor("FF0000");
Ê        }
Ê        else {
Ê          setForegroundColor(null);
Ê        }
Ê      }

Ê      @Override
Ê      protected void execInitField() throws ProcessingException {
Ê        setValue(-3.1415);
Ê        setForegroundColor("FF0000");
Ê      }
```

The styled double field in the example section of the form displays negative values in red and postive values in the default black color. This behaviour is implemented in method execChangedValue according to Listing Listing Integer field. The sample value shown initially is provided in the execInitField method. Setting the red foreground color explicitely is needed in method execInitField as the execChangedValue is only triggered after the initial form is displayed on the screen.

For displaying and storing the fraction digits of decimal values three different properties exist. Two of them, the getConfiguredMinFractionDigits and the getConfiguredMinFractionDigits affect the optical representation of the decimal value. To configure the amount of fraction digits that is effectively represented the property getConfiguredFractionDigits is used. The need for three different properties might not be immediately clear. To illustrate the concept, let us look at an example use case where a decimal field always has to display exactely 3 fraction digits. Should the user provide more fraction digits we would like to capture this additional information up to 5 fraction digits.

To configure this behaviour the following settings for this decimal field may be used.

- *Min Fraction Digits*: 3
- *Max Fraction Digits*: 3

¥ *Fraction Digits*: 5

If the user enters the text ò3.1415926535897930ó into this field and tabs to the next field, the field will then display the text ò4.142ó but actually hold the value 3.14159. And if the user just enters a ò3ó, the field will display ò3.000ó and hold the value 3.0.

Decimal fields can also be configured to enter percentages in a convenient way. For this use case the *Multiplier* property can be set to 100 and the *Multiplier* property to true. If the user now enters ò5ó into such a decimal field, it will show the text ò5%ó and hold the value 0.05.

As in the case of number fields more extensive options to specify the formatting of the numbers is provided by setFormat method of the decimal field. Method setFormat is accepting an argument of the Java class DecimalFormat. To demonstrate an example for such a format click on [!Sample Format!] button in the example form. For more information please consult the Javadoc for class DecimalFormat.

7.5. Date and Time Fields

To work with date and time values Scout offers three dinstinct form fields. The AbstractDateField allows the user to enter a date and the AbstractTimeField is used to enter a time. The third field AbstractDateTimeField combines date and time entry into a single form field. Both classes AbstractTimeField and AbstractDateTimeField are extending the AbstractDateField field.

Figure 45. Example use cases for a date, a time and a combined date/time field.

The form DateTimeFieldsForm of the Scout widget application shown in Figure 000 contains examples for the date and time fields of Scout. Separate buttons are available to provide samle values and to demonstrate the formatting options for displaying date and time values. Displaying dates and times is highly depending on the used locale. That is why the currently used locale is shown in the example section of the form.

To enter date and time values the user can either click on the date and time icons/buttons provided by the fields or directly enter text into the fields. For entering dates the key arrows provide a number of shortcuts. Entering the currnt date can be done by pressing the Up. Once a day is entered in a date or a combined day time field, the Up and the Down can be used to step to the next/previous day. Simultaneously pressing the Shift or the Ctrl allows to step to the next/previous month or year.

Listing 22. A disabled combined date time field initialized with the current time

```
Ê    public class PlaceholderField extends AbstractPlaceholderField {
Ê    }

Ê    @Order(110.0)
Ê    public class DateTimeColumnField extends AbstractLabelField {

Ê      @Override
Ê      protected String getConfiguredFont() {
Ê        return "BOLD";
Ê      }

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

Ê      @Override
Ê      protected void execInitField() throws ProcessingException {
```

The code of the DateTimeDisabledField field shown in Listing Listing date time field represents the disabled combined date time in the example form. Before the form is opened Scout executes its execInitField method and sets the fields value to the current date and time.

In the configuration section the local can be used at runtime to test the effect of the locale to displaying date and time fields. As changing the locale at runtime only works reliably in rich clients the field is only editable in the Swing or the SWT client. To specify the exact formatting of the displayed date and time values a specific format can be set in the getConfiguredFormat method of the date and time fields. Internally, Scout is using the provided string to create a Java SimpleDateFormat for formatting. Valid examples for the formatting are entered into the format fields of the example dialog by pressing the [!Sample Formats!] button. The string ÒEEEEÓ shown in date field format field represents the day of the week as shown in configuration section of Figure 000. As expected, the textual representation of the day of the week is depending on the used locale.

7.6. Checkbox Fields

Check boxes can be used to enter/represent simple boolean values. In Scout, check boxes are derived from class AbstractCheckBox.

Figure 46. Check box field and example use cases.

In the Scout widget application the use of check boxes is demonstrated in the form CheckboxFieldForm that is shown in Figure 000. To access the current value, Scout provides the method isChecked for check box fields. This naming reflects the boolean state of a check boxes and differs from the other Scout value fields that provide a getValue method.

Listing 23. A disabled check box field initialized with a checked state

```
Ê     @Order(20.0)
Ê     public class DisabledField extends AbstractCheckBox {

Ê       @Override
Ê       protected boolean getConfiguredEnabled() {
Ê         return false;
Ê       }

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

Ê       @Override
Ê       protected void execInitField() throws ProcessingException {
Ê         setChecked(true);
Ê       }
Ê     }
Ê   }
```

A coding example is provided in Listing Listing check box field for the disabled check box. The initial

value is set in method execInitField using the method setChecked.

7.7. Radio Button Fields

With radio buttons the user can select a single element out of a number of distinct choices. For this, a number of radio buttons may be placed into a radio button group field where a radio button group extends class AbstractRadioButtonGroup. The contained individual buttons are extending class AbstractRadioButton.

Radio buttons in a radio button group field and example use cases.

```
ÊAssigning distinct values to the individual radio buttons allows to query the selected
radio button.
image::{imgsdir}/radiobuttonfield.png[]
```

Figure 000 demonstrates the use of radio button groups in Scout. It is implemented in class RadioButtonGroupFieldForm of the Scout widget application. As shown in the example section of the form, radio buttons may have labes and/or icons assigned.

Listing 24. A radio button group defined by a code type

```
Ê       }

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

Ê       @Override
Ê       protected void execInitField() throws ProcessingException {
Ê         setValue(EventTypeCodeType.ExternalCode.ID);
Ê       }
Ê     }

Ê     @Order(20.0)
Ê     public class DisabledGroup extends AbstractRadioButtonGroup<Long> {

Ê       @Override
```

The simplest way to define the content of a radio button group is by using its configuration properties getConfiguredCodeType or getConfiguredLookupCall. Listing Listing radio button group provides the implementation of ÒDefaultÓ radio button group DefaultGroup. This radio button group is typed by the generic parameter Long and the individual radio buttons are obtained from the code type EventTypeCodeType specified in method getConfiguredCodeType. Please note that the Long type

matches the key type of the codes in the EventTypeCodeType. The generic parameter used in the definition of a radio button group also determines the type that will be returned by the group's getValue method.

Listing 25. A complete radio button group with two radio buttons with individual radio values assigned

```
Ê       public class ErrorButton extends AbstractRadioButton {

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

Ê         @Override
Ê         protected Object getConfiguredRadioValue() {
Ê           return Long.valueOf(-2L);
Ê         }
Ê       }
Ê     }
Ê   }

Ê   @Order(20.0)
Ê   public class ConfigurationBox extends AbstractGroupBox {

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

Ê     @Override
Ê     protected boolean getConfiguredLabelVisible() {
Ê       return true;
Ê     }

Ê     private void updateRadioValues() {
Ê       if (getValueButton1Field().getValue() != null) {
Ê         getNo1Button().setRadioValue(getValueButton1Field().getValue());
Ê       }
Ê     }

Ê     @Order(10.0)
Ê     public class RadioButtonGroup extends AbstractRadioButtonGroup<Long> {

Ê       @Override
Ê       protected String getConfiguredLabel() {
Ê         return TEXTS.get("RadioButtonGroup");
```

Alternatively, the individual buttons in a radio button group can also be defined as inner classes. This approach has been used for the ÒStyledÓ radio button group in the example section of the RadioButtonGroupFieldForm. The corresponding code is provided in Listing Listing complete radio button group. The type of the value that is returned by the getValue method is defined on the radio button group. In the provided listing, the StyledGroupBox is configured to return a value of the type Long. Consequently, the individual radio buttons are returning radio values of the type Long as well in the getConfiguredRadioValue methods. The field PlaceholderField in the radio button group only serves layouting purposes. Without this place holder field, the two radio buttons of the styled radio button group would be evenly distributed in the available space.

7.8. Buttons and Links

Buttons and links are used to trigger actions in Scout. Buttons come in two variations, normal push buttons and toggle buttons. While buttons have an associated label and/or icon, links can only have a label. Buttons extend class AbstractButton and links extend AbstractLinkButton.

Buttons and links may be placed on a Scout form to initiate actions.

```
ÊButtons may have an associated icon and/or a label.
ÊLinks only have a label.
image::{imgsdir}/buttonlink.png[]
```

Example uses of buttons and links are shown in the form ButtonLinkFieldsForm of the Scout widget application shown in Figure 000.

Listing 26. A button with a label and an icon that horizontally stretches over the whole column

```
Ê    @Order(80.0)
Ê    public class StyledField extends AbstractButton {

Ê      @Override
Ê      protected boolean getConfiguredFillHorizontal() {
Ê        return true;
Ê      }

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

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

Ê      @Override
Ê      protected boolean getConfiguredProcessButton() {
Ê        return false;
Ê      }
Ê    }
```

In it̂s simplest form a button just extends class AbstractButton and overrides getConfiguredLabel to set the label. The example in Listing Listing button with a label also has an icon assigned and stretches over the whole column width. In addition, the example button overrides method getConfiguredProcessButton to return false. This has the effect that buttons (and links) appear at the exact location where they are defined in a field container. Otherwise, buttons (and links) are placed at the bottom of the container they are defined in.

Listing 27. A toggle button implementation that changes the label text depending on its toggled state

```
Ê     @Order(100.0)
Ê     public class ToggleButtonDefaultField extends AbstractButton {

Ê       @Override
Ê       protected int getConfiguredDisplayStyle() {
Ê         return DISPLAY_STYLE_TOGGLE;
Ê       }

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

Ê       @Override
Ê       protected boolean getConfiguredProcessButton() {
Ê         return false;
Ê       }

Ê       @Override
Ê       protected void execSelectionChanged(boolean selected) throws ProcessingException
{
Ê         if (selected) {
Ê           setLabel(TEXTS.get("Selected"));
Ê         }
Ê         else {
Ê           setLabel(TEXTS.get("PushToSelect"));
Ê         }
Ê       }
```

The code of the default toggle button in the example form of the widget application is provided in Listing Listing Toggle Button field. To query the state of a toggle button method isSelected can be used.

In the configuration section of the example form the use of mnemonics using the character & is demonstrated. Please note, that this feature is not identically available across the different supported UI technologies. In the Swing UI shown in Figure 000 the label Ò&ToggleÓ has the effect, that pressing the ALT+T key combination changes the toggle state of this button. In the case of the SWT UI the alt key is not necessary. Pressing the T changes the toggle state. In contrast to the Swing UI the letter ÔTÕ is not underlined. To optically indicate shortcut letters in the SWT UI it is recommended to adapt the label from Ò&ToggleÓ to Ò[&T]oggleÓ. The RAP UI does not currently support mnemonics on buttons.

7.9. Message Boxes

Message boxes are used to provide information to a user or ask the user simple yes/no questions. In Scout, class MessageBox provides a number of static convenience methods for this purpose.

Additionally, message boxes are shown to the user in the case of a processing exception or a veto exception. [48: The processing exceptions type ProcessingException represents Scout's core exception class. The veto exception type VetoException is a direct subclass of the processing exception and is typically used in service calls for subjects with insuffiecient authorization.].

Message boxes are available for different use cases.

```
ÊThe message box shown in front is defined by the properties entered in the configuration
section.
image::{imgsdir}/messagebox.png[]
```

In the examples section of the MessageBoxForm form shown in Figure 000 a number of links is provided. Clicking on any of these links opens a corresponding message box via the static convenience methods available with class MessageBox. For example, calling MessageBox.showOkMessage(title, header, info) opens a message box with a title, a header text and some additional information.

Listing 28. Configuring and starting of a message box.

```
Ê      protected void execClickAction() throws ProcessingException {
Ê        String title = getTitleField().getValue();
Ê        String introText = getIntroTextField().getValue();
Ê        String actionText = getActionTextField().getValue();
Ê        String yesButtonText = getYesButtonTextField().getValue();
Ê        String noButtonText = getNoButtonTextField().getValue();
Ê        String cancelButtonText = getCancelButtonTextField().getValue();
Ê        String hiddenText = getHiddenTextContentField().getValue();
Ê        String iconId = getIconIdField().getValue();

Ê        long autoCloseMillis = NumberUtility.nvl(getAutoCloseMillisField().getValue(),
-1);
Ê        int defaultReturnValue = NumberUtility.nvl(getDefaultReturnValueField()
.getValue(), IMessageBox.CANCEL_OPTION);

Ê        MessageBox msgbox = new MessageBox(title, introText, actionText, yesButtonText,
noButtonText, cancelButtonText, hiddenText, iconId);
Ê        msgbox.setAutoCloseMillis(autoCloseMillis);

Ê        int result = msgbox.startMessageBox(defaultReturnValue);
Ê        getReturnValueField().setValue(result);
Ê      }
Ê    }
```

In addition to the static convenience methods, message boxes can be configured to meet specific requirements by a number of parameters. These parameters are shown in Figure 000 in the configuration section of the sample form MessageBoxForm of the Scout widget application. Clicking on

the [!Sample Content!] button will fill in example values for the message box parameters. The configured message box can them be started by clicking on the *Open the configured MessageBox* link. This behaviour is implemented in the link's execClickAction method according to Listing Listing MessageBox. The text below details the purpose of less evident message box properties.

Message box buttons only appear if an non-emtpy label text is assigned to the *Yes Button Text* property, the *No Button Text* property and *Cancel Button Text* property. If the *hidden text* property has a non-empty text assigned, an additional [!Copy!] button will be added to the message box. An example use case is the case of elaborate error messages (or complete stack traces in cases where this does not negatively impact the application's security). Here the copy button allows to transport this text into the system clipboard. From the clipboard the user may decides to paste this text into an email to the companys help desk. The *default return value* property specifies the return value of the message box if the box closes autmatically after the time provided in the *auto close millisecond* property has passed. If the *auto close millisecond* property is set to -1, the message box will not clause automatically.

Once a message box is started with method startMessageBox the user interface is blocked until the user chooses any of the options or clicks the dialog away. The selected option is the provided by the method's return value. If the user closes the message box by hitting the esc or clicking on the icon to close a dialog, the start method of the message box will always return the value CANCEL_OPTION of the IMessageBox interface.

8. Advanced Widgets

This chapter presents some of the more complex widgets of the Scout framework. This set of widgets includes fields to handle and display list, tree and table data. Also included in this chapter are widgets to display images in both raster and vector formats. As in the previous chapter, for each described widget, screenshots of example use cases and corresponding code snippets are provided.

8.1. List Box

List boxes allow a user to select a subset of a predefined list of elements. The individual elements are displayed in the form of checkable options. To check or uncheck a specific element, the user may click on a element or press the Space when an element has the focus. To define the list of elements that are presented in a list box, code types or a lookup calls can be used. Example use cases using both code types and lookup calls for the definition of the presented elements are implemented in class ListBoxForm of the Scout widget application. See Figure 000 for a screenshot of the example form.

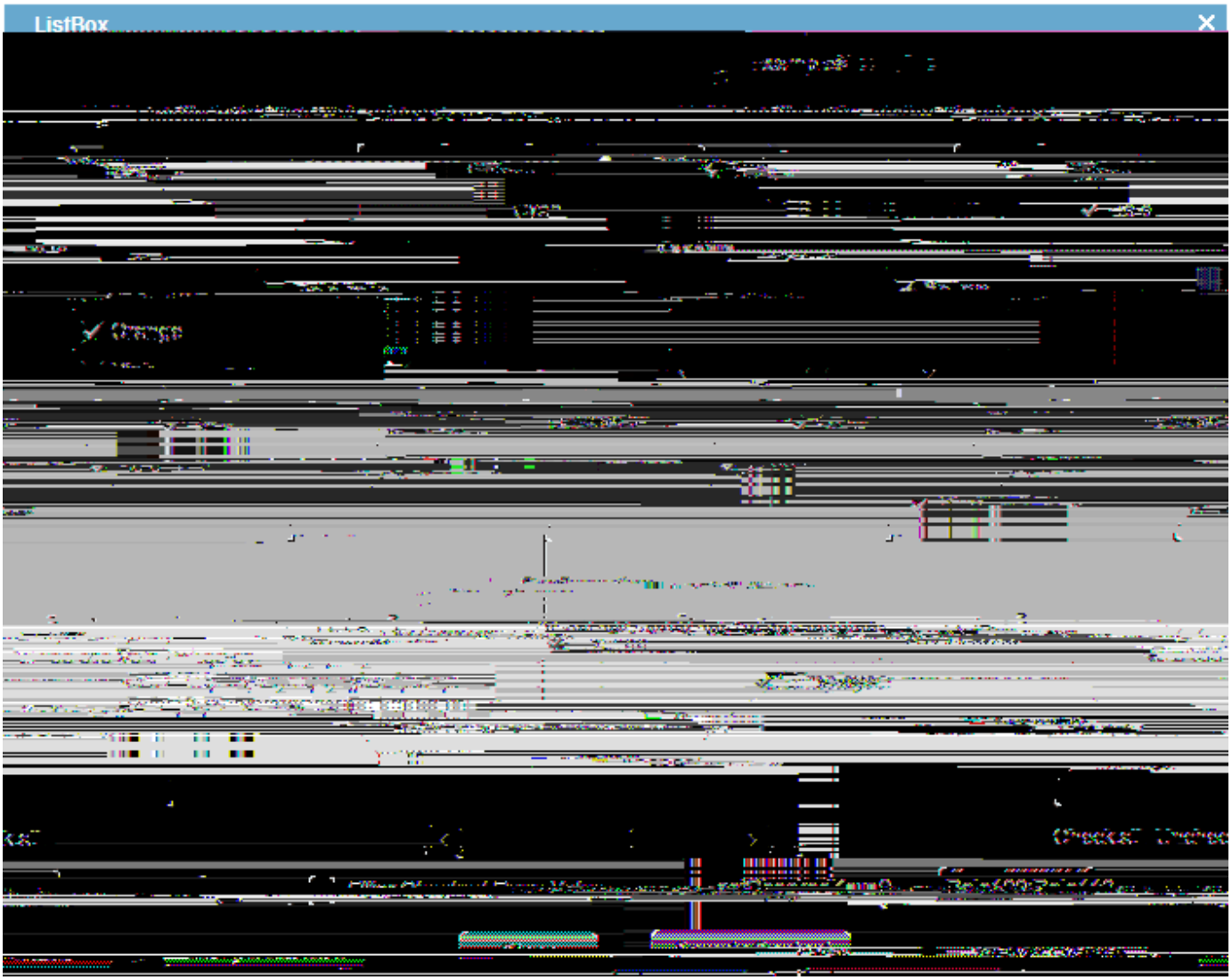


Figure 47. List boxes to select elements represented by code types and lookup calls.

Listing 29. A simple ListBox field backed by a code type that returns elements of type Color.

```
Ê    }

Ê    @Order(20.0)
Ê    public class DefaultField extends AbstractListBox<Color> {

Ê      @Override
Ê      protected Class<? extends ICodeType<?, Color>> getConfiguredCodeType() {
Ê        return ColorsCodeType.class;
Ê      }

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

Ê      @Override
Ê      protected String getConfiguredLabel() {
Ê        return TEXTS.get("Default");
```

In the example section of Figure 000 the list boxes in the left column retrieve the elements to be displayed from code types. And in the right column, the displayed elements are retrieved from lookup calls. The implementation of the top left *Default* field is provided in Listing Listing button field. List boxes are derived from class AbstractListBox and parameterized by the type of keys of the elements. Please note that the specified key type of a list box must match the key type of the elements provided by code types or lookup calls. In our example, the list box type Color matches the key type of the code type ColorsCodeType configured in method getConfiguredCodeType.

In the *List Content* field located on the right hand side of the configuration section of Figure 000 it is also possible to enter user defined content. After entering some example data and pressing the Tab, the user content is parsed and used to update the elements displayed in the *ListBox* field. To get the initial example data shown in Figure 000, use the [!Sample Content!] button of the list box example form. Basically, the content of each text row is parsed into a lookup row according to the format provided in the first row of the sample content: `# key;text;iconId;É`. The resulting lookup rows are then used to dynamically update the content of the lookup call associated with the *ListBox* field. List boxes can be configured to only display the checked elements. For this, the configuration method getConfiguredFilterCheckedNodes may be used.

Listing 30. Updating the getCheckedKeys field whenever the use changes the selection of elements

```
Ê        protected void execClickAction() throws ProcessingException {
Ê          getListBoxField().uncheckAllKeys();
Ê        }
Ê      }
Ê    }

Ê    @Order(30.0)
Ê    public class GetCheckedKeysField extends AbstractStringField {

Ê      @Override
Ê      protected boolean getConfiguredEnabled() {
Ê        return false;
Ê      }

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

Ê      @Override
Ê      protected Class<? extends IValueField> getConfiguredMasterField() {
Ê        return ListBoxForm.MainBox.ConfigurationBox.ListBoxField.class;
Ê      }
```

To access the currently selected elements of a list box method getCheckedKeys may be used. This is demonstrated in Listing Listing ListBox.getCheckedKey using the *getCheckedKeys* field in the example form. In order to get notified for every change of the list box field, the list box field is registered as the master in method getConfiguredMasterField. The implementation of method execChangedMasterValue is then called whenever the state of the master field is changed. In the case of Listing Listing ListBox.getCheckedKey, the list of selected keys can be retrieved from the list box and shown in the *getCheckedKeys* field.

To read or write the content of a list box field from or to a form data on the server side, methods getValue and[java]setValue have to be used. Both methods work with typed sets, requiring the type defined for the list box field.

8.2. Tree Box

Tree box fields allow a user to select a subset of a predefined list of elements. The difference to list boxes lies in the organisation of the presented elements. Instead of presenting the elements as a list, a tree structure is used with tree box fields.

To check and uncheck a specific element, the user may click on a element or press the Space when an element has the focus. In addition, tree boxes can also be configured to check or uncheck the complete

sub tree when clicking on an element. To define the trees presented in tree boxes, hierarchical code types and lookup calls can be used. Example use cases for tree box fields are implemented in class TreeBoxForm of the Scout widget application. A screenshot of the tree box example form is shown in Figure 000.

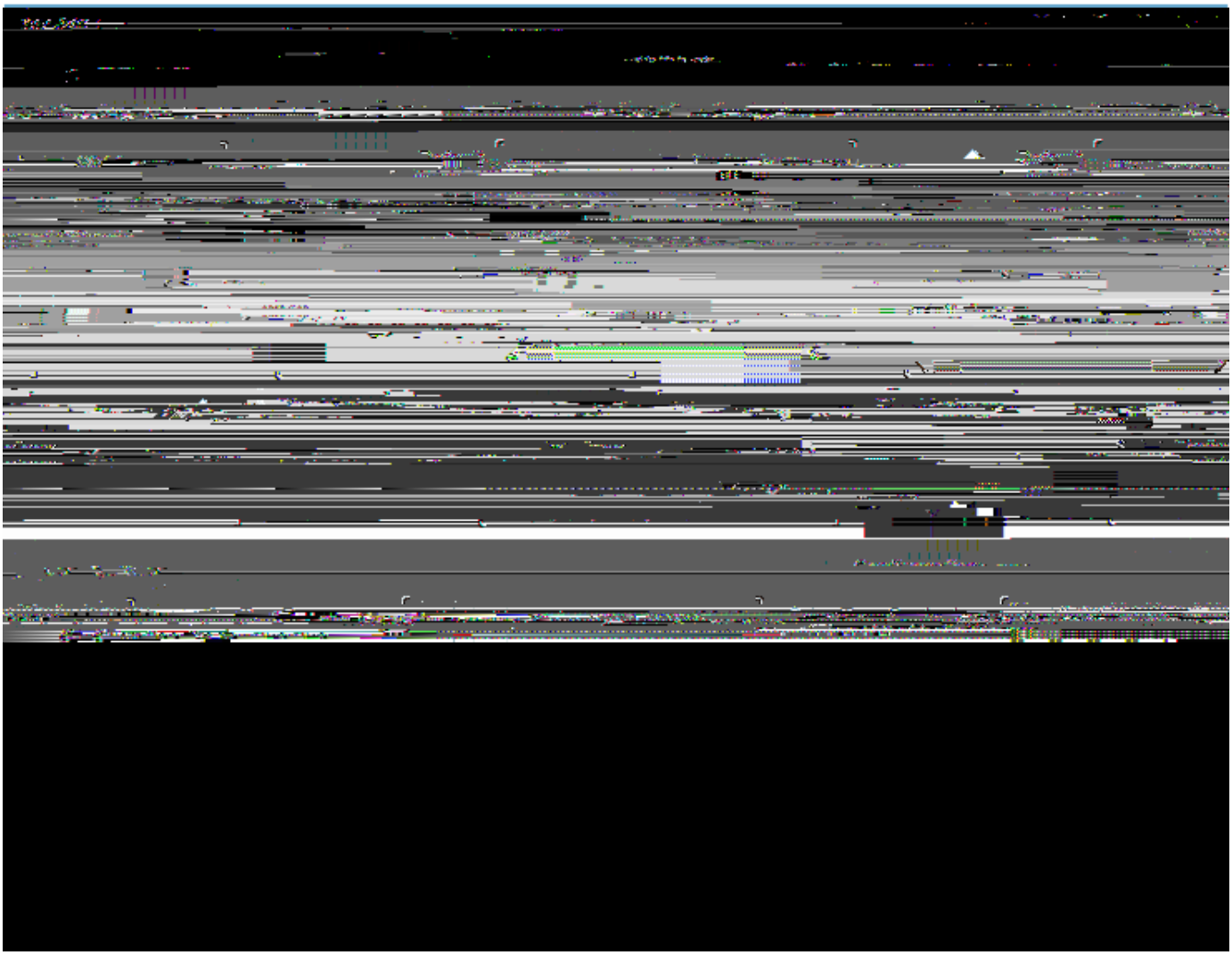


Figure 48. Tree box fields displaying elements defined by hierarchical code types and lookup calls.

Listing 31. A simple TreeBox field backed by a lookup call that returns element keys of type String.

```
Ê    @Order(50.0)
Ê    public class DefaultTreeBoxField extends AbstractTreeBox<String> {

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

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

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

In the example section of Figure 000 the tree boxes in the left column retrieve the elements to be displayed from hierarchical code types. And in the right column, the displayed elements are retrieved from hierarchical lookup calls. The implementation of the top right *Default* field is provided in Listing xxx. Tree boxes are derived from class AbstractTreeBox and parameterized by the type of keys of the tree elements. As in the case of list box fields, the the specified type of a tree box must match the key type of the code type or the lookup call. In our example, the tree box type String matches the type of the lookup call YearsMonthsLookupCall configured in method getConfiguredLookupCall.

In the configuration section of the form shown in Figure 000 the user may enter any tree data into the *Tree Content* field. To get the initial tree data shown in Figure 000, use the [!Sample Content!] button. Tree boxes can be configured in several ways. Using method getConfiguredAutoExpandAll, the element tree will be initially expanded. Configuration method getConfiguredFilterCheckedNodes hides all elements that are not checked. And with method getConfiguredAutoCheckChildNodes, checking or unchecking a tree element automatically updates the elements in the subtree accordingly.

To access the currently selected elements, the tree box method getCheckedKeys returns a typed set of keys. The type of the key set is determined by the type of the tree box. To read or write the content of a tree box field on the server side, methods getValue and[java]setValue have to be used. Both methods work with typed sets, requiring the type defined for the tree box field.

8.3. Smart Field

Smart fields are used to select a single value from a set of named elements. As smart fields offer Òsearch-as-you-typeÓ functionality, the field works well for very large sets of elements. The content of

smart fields can either be provided by code types or lookup calls. Consequently, smart fields work with both lists and hierarchical structures and the content may come from a static set of values or is dynamically provided at runtime.

To select an elements in a smart field the user can either use the mouse or the keyboard. Pressing the Up Arrow or the Down Arrow in the field shows the list of entries when the smart field has the focus. Alternatively, the use can click with the mouse on the smart field icon. By typing a part of the name of the desired entry into the field, the Òsearch-as-you-typeÓ support kicks in and a filtered list of elements is displayed. The selected entry can be confirmed by using the Enter or the Tab.

Figure 49. Smart field examples. Smart fields support Òsearch-as-you-typeÓ and are used to select a value from of a list of elements or a tree.

Example use cases for smart fields are provided in class SmartFieldForm of the Scout widget application and a screenshot of this example form is shown in Figure 000. The left hand side of the demo form contains smart fields based on list content and on the right hand side smart fields backed with hierarchical content are shown. In the example section of the demo form, some smart fields are backed by code types and others by lookup calls. And in the configuration section the content shown in the smart fields can be entered manually at runtime. To obtain the content shown in Figure 000, use the [!Sample Content!] button.

Existing Documentation

¥ presentation: http://wiki.eclipse.org/images/c/c9/20111102_EclipseConEurope2011-EclipseScout-

DiscoverThePotential.pdf

¥ forum: http://www.eclipse.org/forums/index.php/t/369542/

8.3.1. Menus

Each smart field can have menus attached. The menus will be shown when the user clicks on the *arrow* symbol next to the smart field. Figure [fig-smartfield_menu] shows an example of a smart field along with a set of menus.

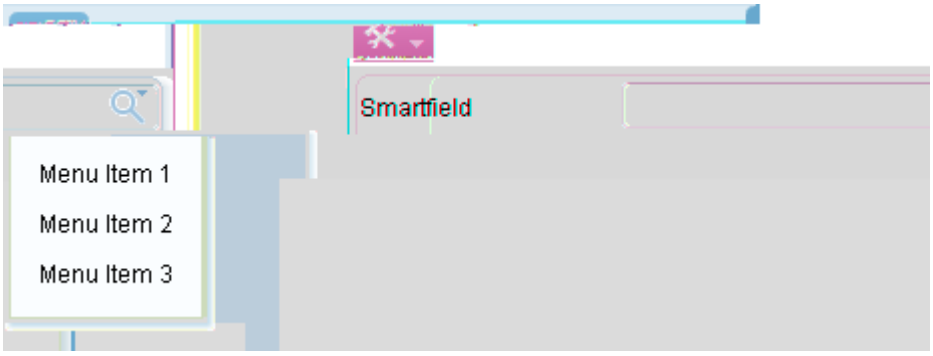


Figure 50. Menus attached to a smart field.

By default, the menus will only be shown if a value of the smart field has been selected. To show the menus even if the smart field is empty, one has to override the menuÕs `getConfiguredEmptySpaceAction` method:

8.4. Tree Field

needs text

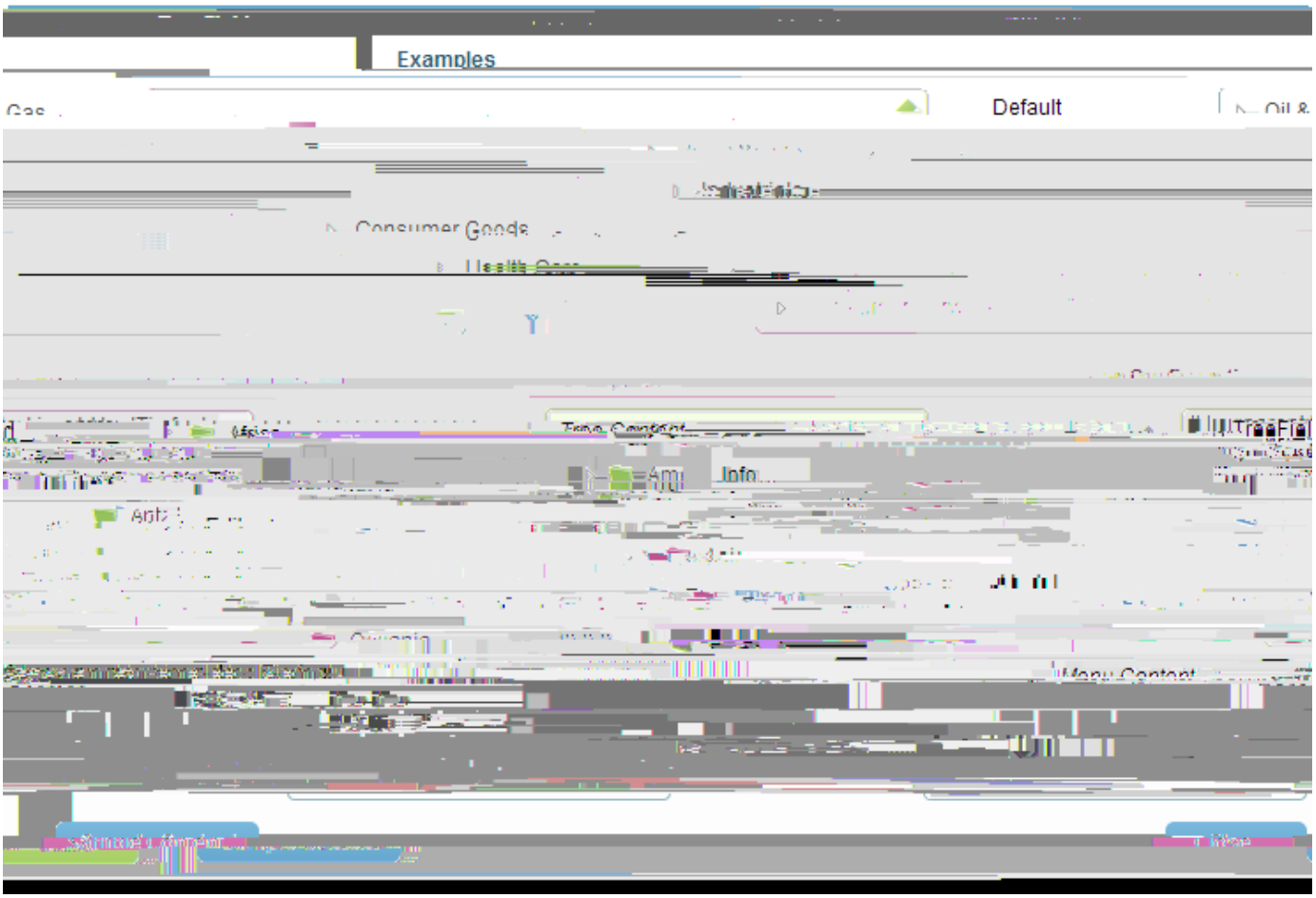


Figure 51. Tree fields and example use cases. More text.

8.5. Table Field

needs text

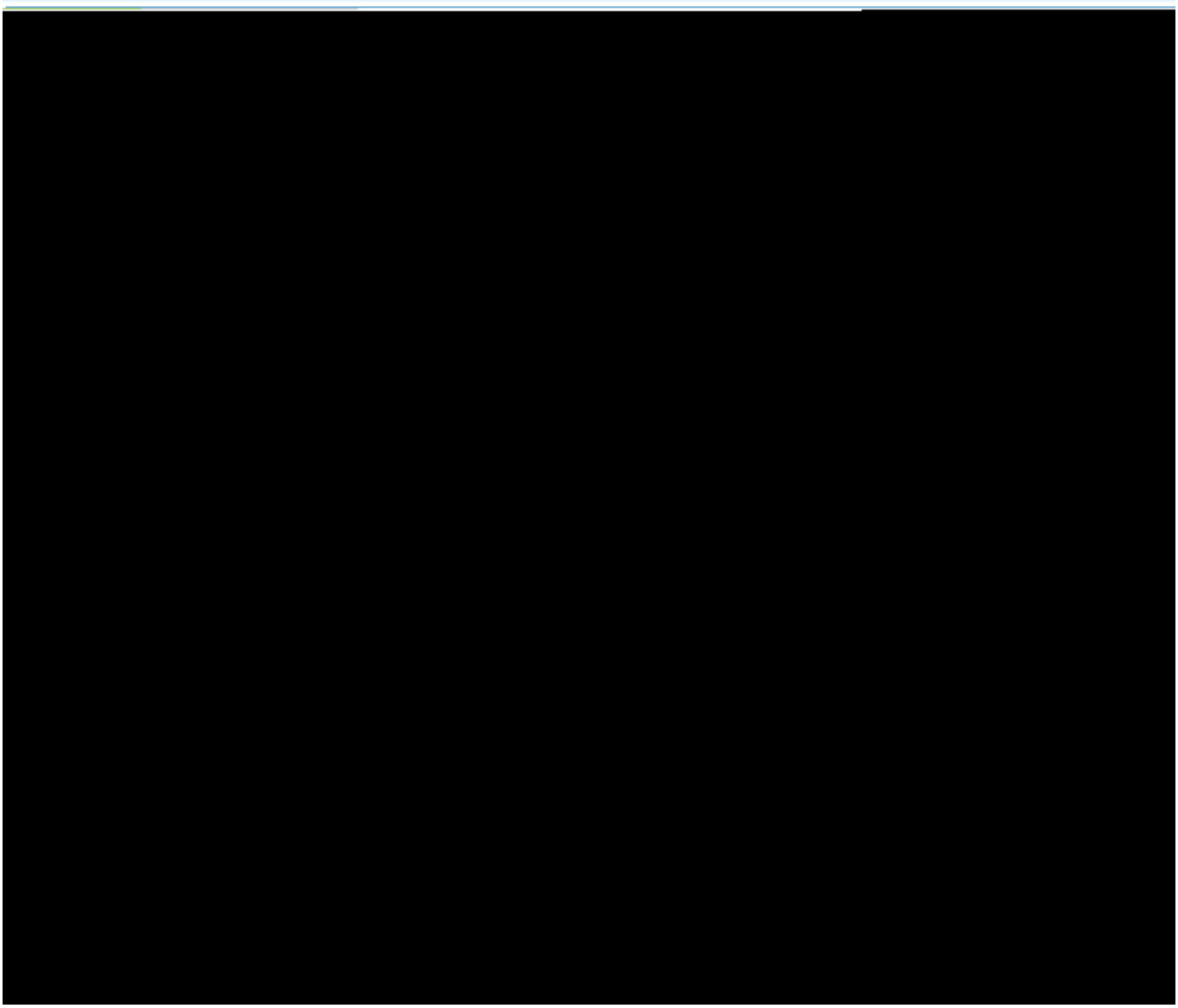


Figure 52. Table fields and example use cases. More text.

Figure 53. An editable table field. More text.

Existing Documentation

- wiki tutorial http://wiki.eclipse.org/Scout/Tutorial/3.8/Minicrm/Table_Field
- forum: http://www.eclipse.org/forums/index.php/t/392053/
- forum: load/save data http://www.eclipse.org/forums/index.php/t/253311/

8.6. Image Field

needs text

Figure 54. Image fields are used to display images and icons.

Existing Documentation

- forum: scrollbars http://www.eclipse.org/forums/index.php/t/291205/

8.7. SVG Field

needs text

Figure 55. SVG fields and example use cases. More text.

```
Existing Documentation
* wiki tutorial http://wiki.eclipse.org/Scout/Tutorial/3.8/SVG_Field
```

8.8. HTML Field

needs text

8.9. Browser Field

needs text

Existing Documentation

- forum: http://www.eclipse.org/forums/index.php/t/414483/,
- forum: http://www.eclipse.org/forums/index.php/t/369963/,
- forum: mozilla as default: http://www.eclipse.org/forums/index.php/t/342433/

8.10. Calendar Field

needs text

Existing Documentation

- forum: calendar field http://www.eclipse.org/forums/index.php/t/370052/
- forum: execloaditems http://www.eclipse.org/forums/index.php/t/277447/
- forum: filtering items http://www.eclipse.org/forums/index.php/t/285644/
- forum: usage example http://www.eclipse.org/forums/index.php/t/265028/

9. Layout Widgets

9.1. Group Box

needs text

Figure 56. Group boxes and example use cases. More text.

9.2. Tab Box

needs text

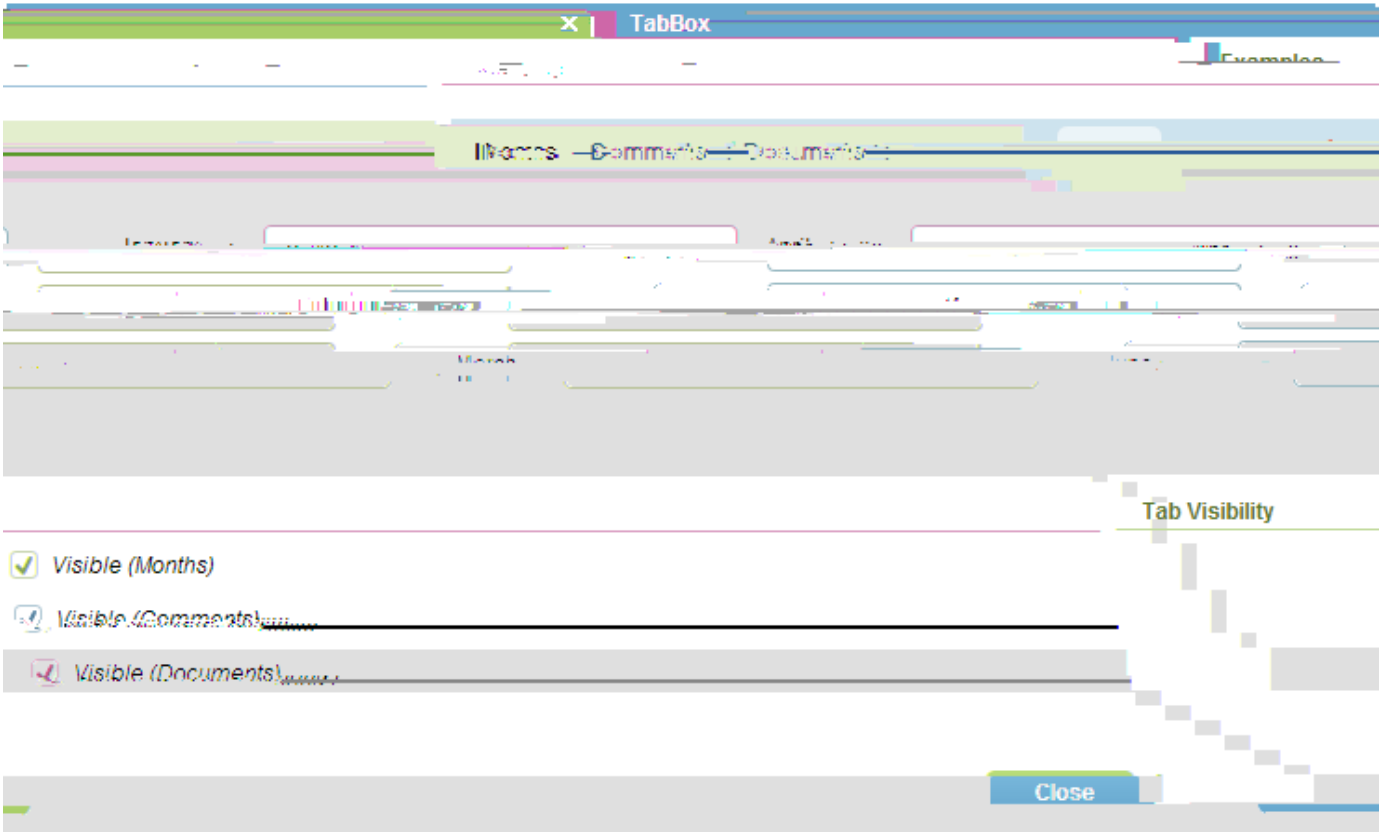


Figure 57. Tab boxes and example use cases. More text.

9.3. Sequence Box

needs text

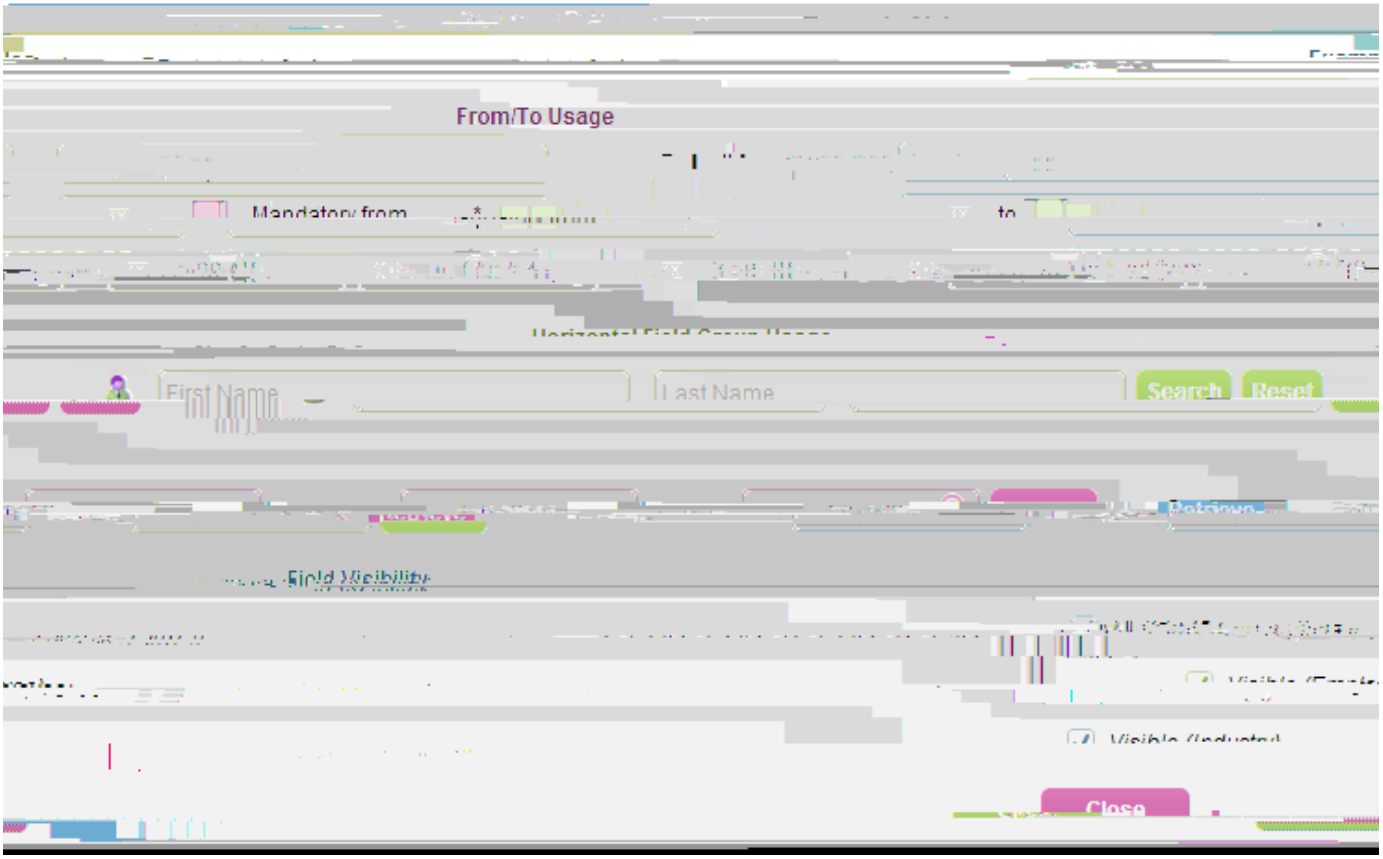


Figure 58. Sequence boxes and example use cases. More text.

Existing Documentation

- forum: http://www.eclipse.org/forums/index.php/t/414629/
- exec methods
- field validation

9.4. Split Box

needs text

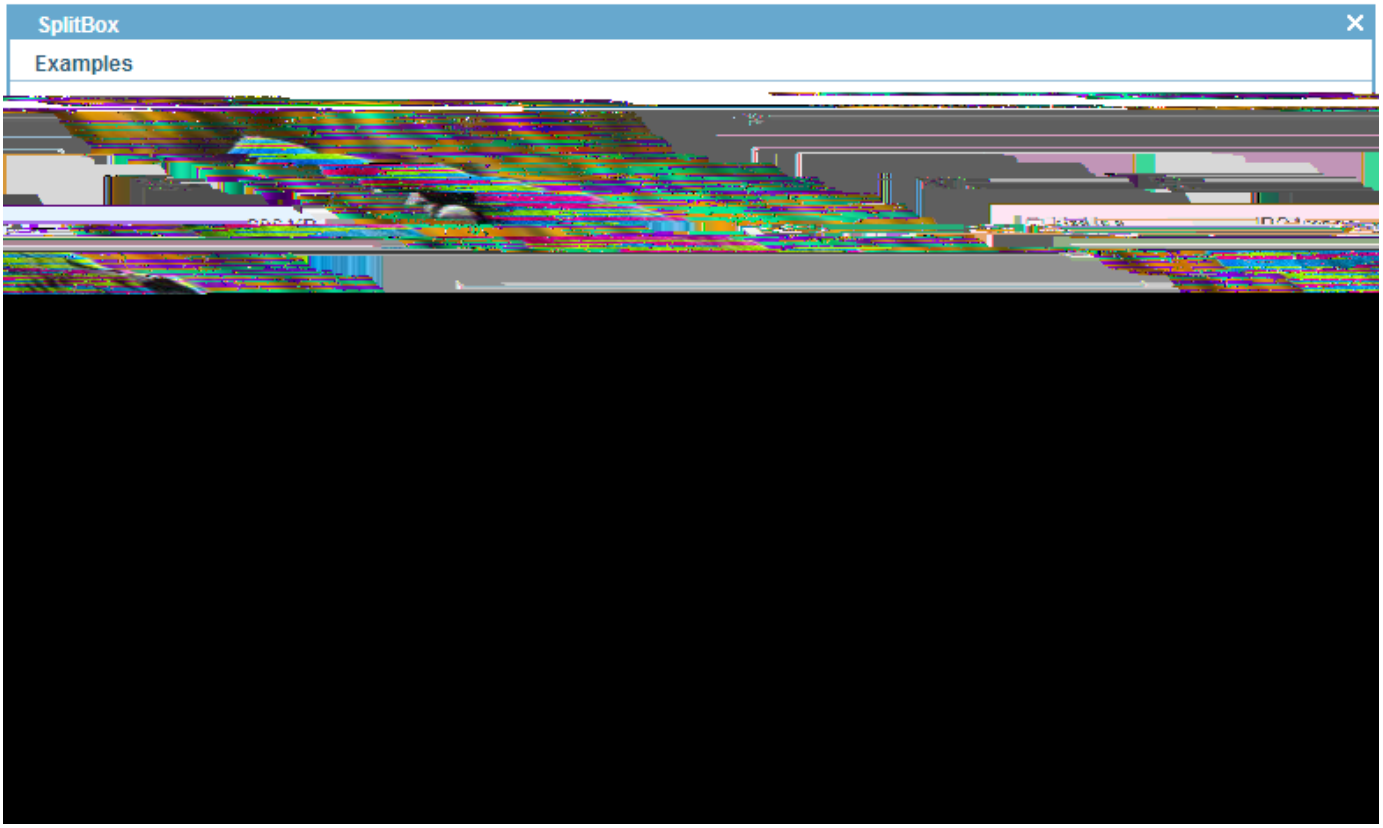


Figure 59. Split boxes and example use cases. More text.

9.5. Page Field

needs text

Existing Documentation

- forum: http://www.eclipse.org/forums/index.php/t/395360/

9.6. File Chooser Field

needs text

Existing Documentation

- forum: file chooser field http://www.eclipse.org/forums/index.php/t/377581/
- forum: open with default file name: http://www.eclipse.org/forums/index.php/t/351352/
- how-to wiki: rap file chooser http://wiki.eclipse.org/Scout/HowTo/3.8/Add_FileChooser_support_for_RAP_UI

9.7. Master Slave Fields

needs text

Existing Documentation

- forum: http://www.eclipse.org/forums/index.php/t/366931/

10. Custom Fields

As we have seen in the previous chapter, Scout already comes with a large amount of ready to use form fields. However, real life projects often need to meet special business requirements that can not be covered by the existing Scout form fields. For such situations the flexibility of the Scout framework allows the project to extend the exiting set of form fields with custom form fields.

Custom form fields concists of a couple of components. Modeling and UI and registration and extension point?

Existing Documentation

- how-to wiki: http://wiki.eclipse.org/Scout/HowTo/3.8/Add_a_custom_GUI_component
- forum: wrap existing javaview.de swing component http://www.eclipse.org/forums/index.php/t/262755/
- concept
- showcase: drawing application

11. Template Fields

needs text

Existing Documentation

- concept wiki: http://wiki.eclipse.org/Scout/Concepts/Template
- forum: form data for template fields http://www.eclipse.org/forums/index.php/t/261235/
- forum: form Òmodularisation Ó http://www.eclipse.org/forums/index.php/t/245857/

12. Layouting

needs text

Existing Documentation

- concept wiki http://wiki.eclipse.org/Scout/Concepts/Client_Plug-In#Layouting

12.1. The Desktop

needs text

12.2. Form Layout

needs text

13. Bookmarks

needs text

14. Client Notification

needs text

Existing Documentation

- presentation: http://wiki.eclipse.org/images/e/ea/20121022_BahBah_Slides.pdf
- concept wiki: http://wiki.eclipse.org/Scout/Concepts/Client_Notification
- forum: http://www.eclipse.org/forums/index.php/t/241053/

15. File Upload and Download

needs text

Existing Documentation

- how-to wiki: http://wiki.eclipse.org/Scout/HowTo/3.8/Transfer_a_file_from_the_client_to_the_server
- how-to wiki: http://wiki.eclipse.org/Scout/HowTo/3.8/Use_RemoteFileService
- forum with error message box exampl http://www.eclipse.org/forums/index.php/t/441101/
- forum: http://www.eclipse.org/forums/index.php/t/368166/
- forum: http://www.eclipse.org/forums/index.php/t/366585/
- forum: remotefileservice http://www.eclipse.org/forums/index.php/t/266862/
- forum: file download http://www.eclipse.org/forums/index.php/t/263896/
- forum: load & display file http://www.eclipse.org/forums/index.php/t/440934/

16. Application Branding

needs text

Existing Documentation

- forum: http://www.eclipse.org/forums/index.php/t/373921/
- forum: Splash http://www.eclipse.org/forums/index.php/t/263003/,
- forum: Splash http://www.eclipse.org/forums/index.php/t/164495/
- forum: Login Box http://www.eclipse.org/forums/index.php/t/417248/
- forum: App Icon http://www.eclipse.org/forums/index.php/t/263221/
- forum: App Name http://www.eclipse.org/forums/index.php/t/262121/
- forum: Desktop http://www.eclipse.org/forums/index.php/t/373921/
- forum: Scout info form http://www.eclipse.org/forums/index.php/t/236630/
- Icons
- Fonts / Colors
- Look and Feel (Swing)

16.1. Rayo Look and Feel

needs text

Existing Documentation

- forum http://www.eclipse.org/forums/index.php/t/369809/
- wiki tutorial http://wiki.eclipse.org/Scout/Tutorial/3.8/Rayo_Look_and_Feel

16.2. Branding the Swing Client

needs text

Existing Documentation

- how-to wiki: for logo http://wiki.eclipse.org/Scout/HowTo/3.8/Branding_the_Swing_Client
- how-to wiki: app logo http://wiki.eclipse.org/Scout/HowTo/3.8/Exchange_Default_Images

16.3. Branding the SWT Client

needs text

Existing Documentation

- how-to wiki: for logo http://wiki.eclipse.org/Scout/HowTo/3.8/Branding_the_Swing_Client
- how-to wiki: app logo http://wiki.eclipse.org/Scout/HowTo/3.8/Exchange_Default_Images

16.4. Branding the Webclient

needs text

Existing Documentation

- forum http://www.eclipse.org/forums/index.php/t/367983/

17. Advanced Topics

needs text

17.1. Modifying the UI at Runtime

needs text

Existing Documentation

- forum: inject fields in form http://www.eclipse.org/forums/index.php/t/367124/

17.2. Focus Handling

needs text

Existing Documentation

- forum: http://www.eclipse.org/forums/index.php/t/369585/

17.3. Keyboard Control

needs text

Existing Documentation

- forum: http://www.eclipse.org/forums/index.php/t/351417/

17.4. Master Detail Pages

needs text

Existing Documentation

- http://www.eclipse.org/forums/index.php/t/405999/

17.5. Client Only Applications

needs text

Existing Documentation

- how-to wiki: http://wiki.eclipse.org/Scout/HowTo/3.8/Create_a_Standalone_Client_with_DB_Access
- forum: client only http://www.eclipse.org/forums/index.php/t/210183/
- forum: offline capable client http://www.eclipse.org/forums/index.php/t/210183/

17.6. Headless Client

needs text

Existing Documentation

- headless client forum http://www.eclipse.org/forums/index.php/t/262563/

17.7. Client Startup

needs text

Existing Documentation

- reading command line parameters forum http://www.eclipse.org/forums/index.php/t/281816/
- do something right after login forum http://www.eclipse.org/forums/index.php/t/261999/

17.7.1. Config.ini File

needs text

Existing Documentation

- config ini file forum http://www.eclipse.org/forums/index.php/t/365140/
- os independent *product/confg.ini forum http://www.eclipse.org/forums/index.php/t/261674/

17.8. Client Shutdown

needs text

17.9. Threading and Jobs

needs text

Existing Documentation

- threading and jobs concept wiki http://wiki.eclipse.org/Scout/Concepts/Client_Plug-In#Threading_and_Jobs

17.10. Caching

needs text

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

- ¥ Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
- ¥ The author0s 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

Copyright (c) 2012-2014.

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 4.0 which is delivered as integral part of the Eclipse Luna release train, 2014. Detailed information regarding the scheduling of this release train is provided in the Eclipse wiki. [49: Luna release plan: http://wiki.eclipse.org/Luna/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. As Scout applications have been built primarily on the Windows platform in the past, Scout also has the highest maturity level on this platform.

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 7 or 8. It is

currently recommended to go for the most recent download of Java 7.

Using Scout with Java 8 is possible and has been tested as part of Eclipse release train testing. [50: Scout 4.0 platforms: https://wiki.eclipse.org/Scout/Release/Luna#Tested_Platforms]. We are currently not aware of any productive installation so far, but this is likely to change in the near future as Oracle's published end of public updates for Java 7 is scheduled for April 2015. [51: Java 7 end of public support: http://www.oracle.com/technetwork/java/eol-135779.html].

You may still use Scout with Java 6. However, this version is no longer tested with Scout and has reached Oracle's end of public updates on February 2013. Older Java versions will no longer work together with the Scout framework.

Currently, we recommend to install the Oracle JDK 7 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. [52: 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. [53: 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. [54: Official JDK 7 download: http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.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 7 Documentation*. The Java API documentation package is available from the official download site. [55: Java API documentation download: http://www.oracle.com/technetwork/java/javase/downloads/index.html], located further down under section *Additional Resources*.

Figure 60. Installer download for Oracle JDK 7. The Windows 64bit installer package is highlighted.

Once you have successfully downloaded the JDK installer, follow the Windows installation guide. [56: 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. [57: Windows Java ÒHello World!Ó: http://docs.oracle.com/javase/tutorial/getStarted/cupojava/win32.html].

Installation instructions for Linux. [58: Install the JDK on Linux: http://docs.oracle.com/javase/7/docs/webnotes/install/linux/linux-jdk.html] and Mac. [59: 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 of version 7 or 8. To download the Eclipse Scout package visit the official Eclipse download page as shown in Figure 000.

Figure 61. 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 "1.7.0_55"
Java(TM) SE Runtime Environment (build 1.7.0_55-b13)
Java HotSpot(TM) 64-Bit Server VM (build 24.55-b03, 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 62. 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.

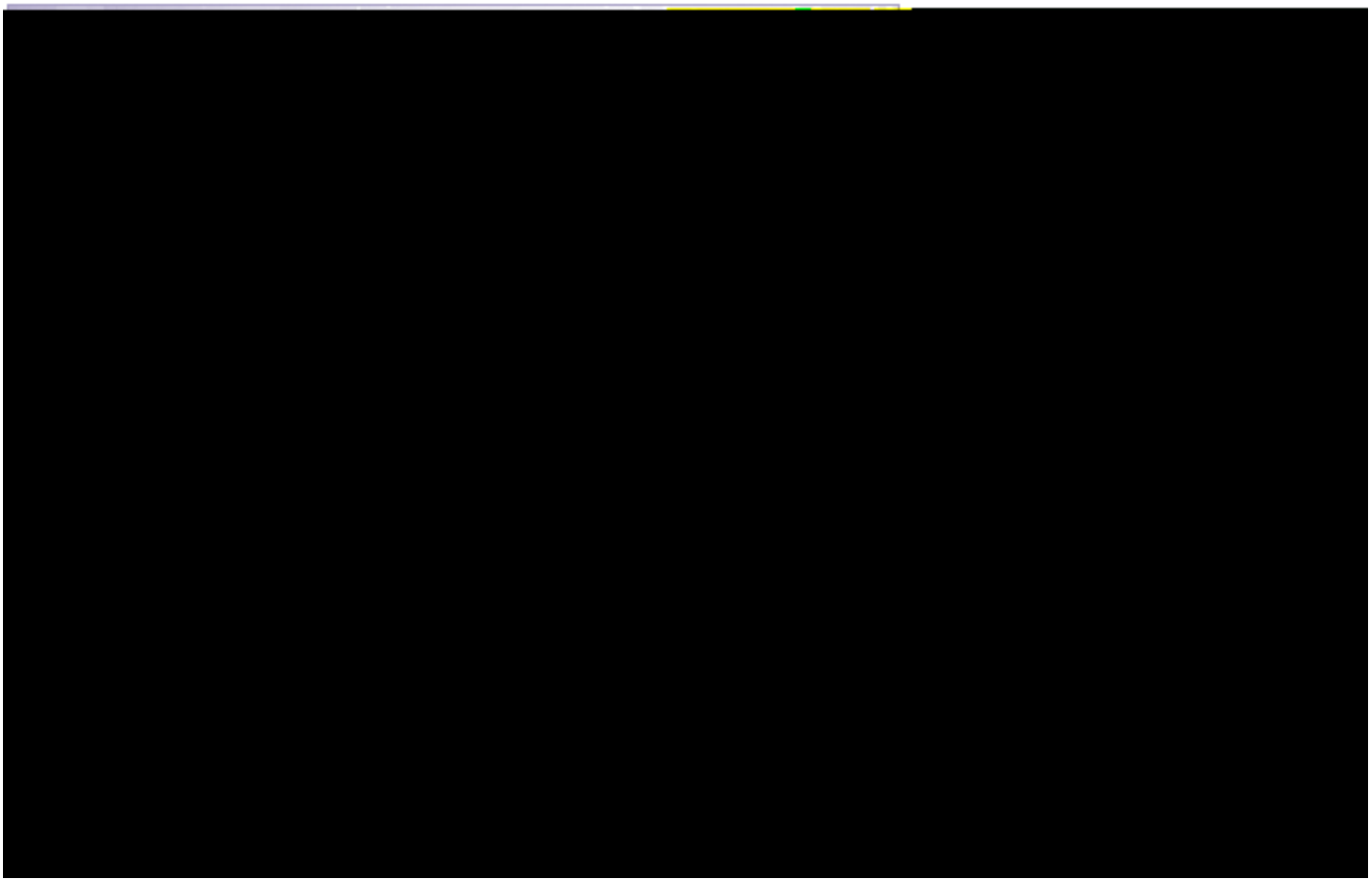


Figure 63. Starting the Eclipse Scout package and selecting an empty workspace.

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 64. 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 Scout perspective is displayed according to Figure 000.

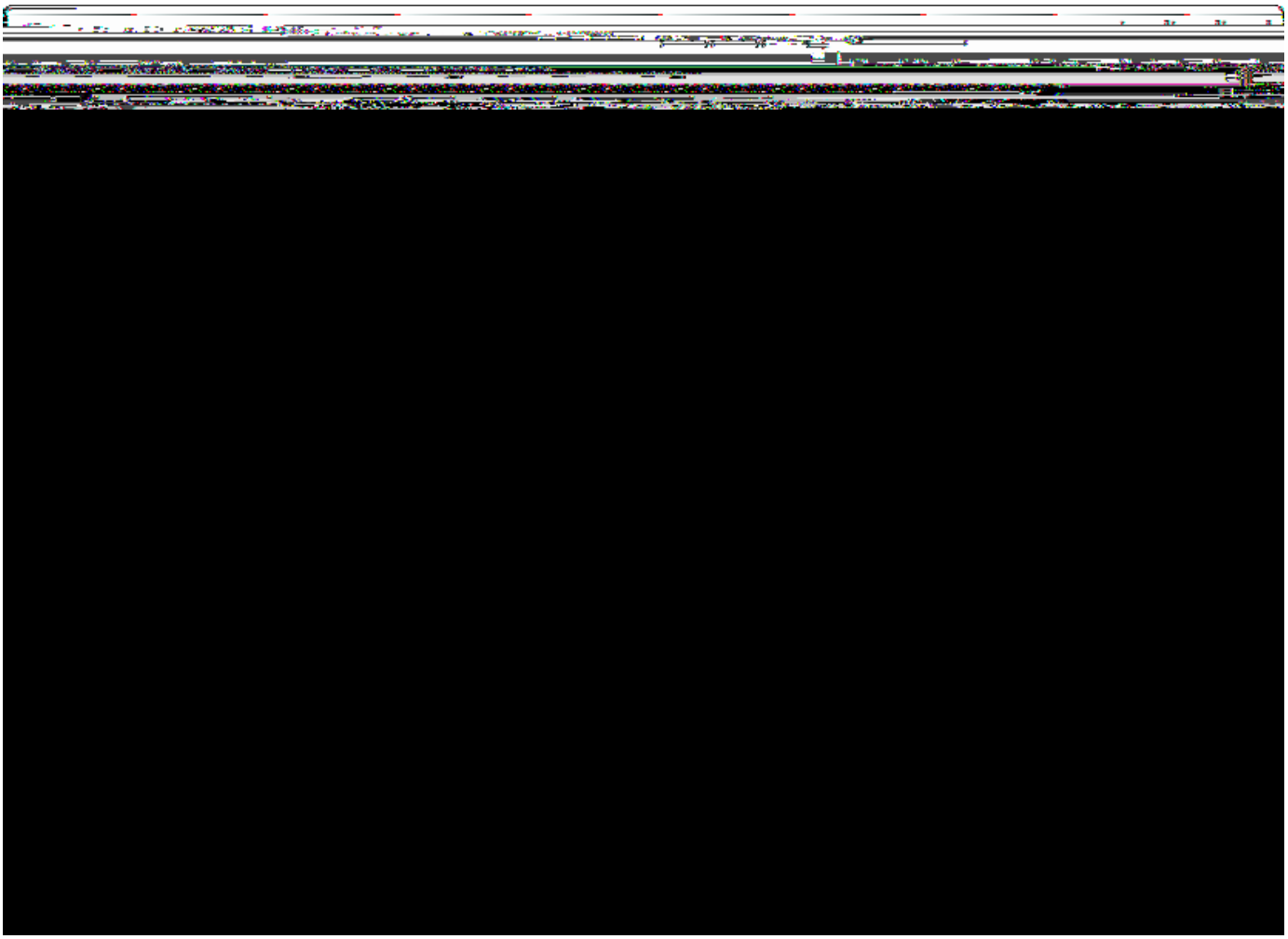


Figure 65. Eclipse Scout started in the Scout SDK perspective.

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\jre7\bin\javaw.exe
```

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\jre7\bin\javaw.exe
...
sun.java.command=... vm C:\java\jre7\bin\javaw.exe -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*.

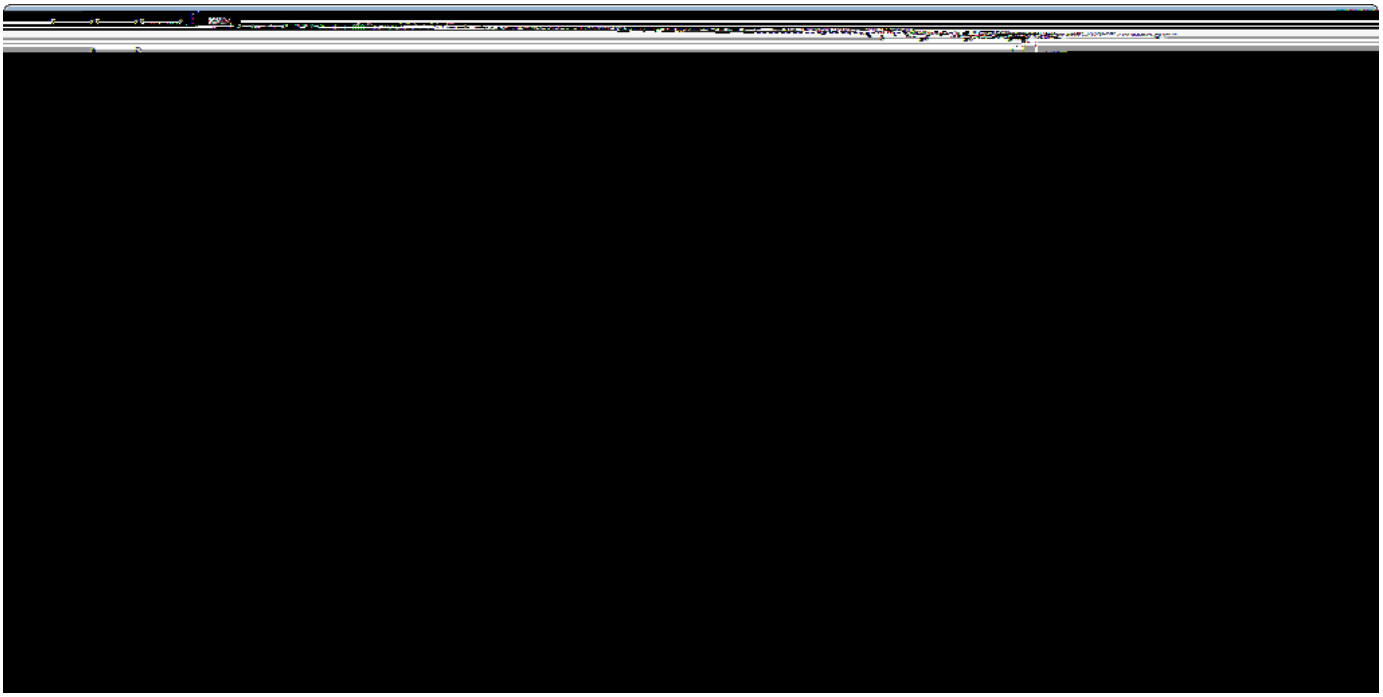


Figure 66. Eclipse menu to install additional software

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

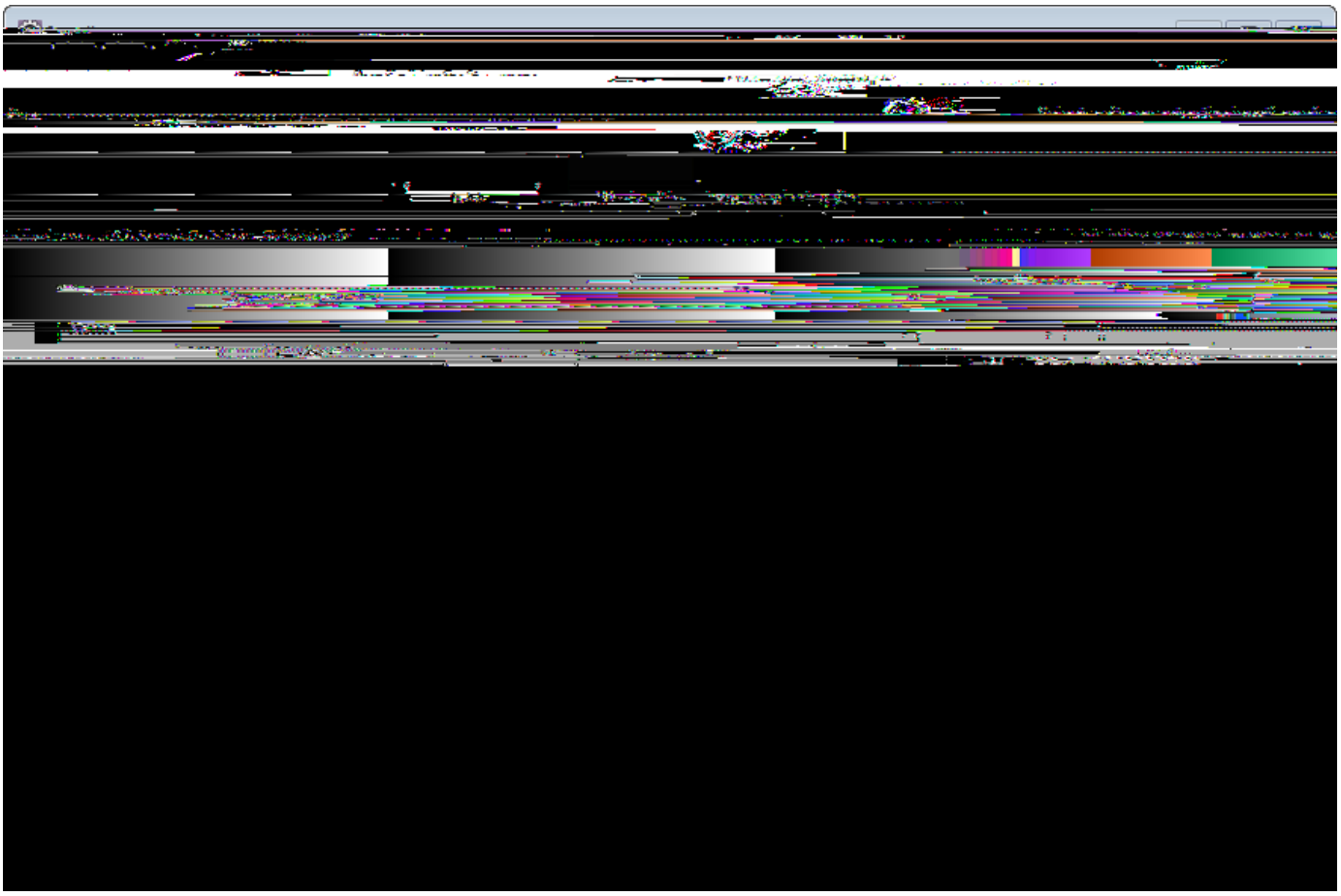


Figure 67. Add the current Scout repository

In the install dialog, click on the [!AddÉ!] 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 68. 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 É 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 the server part of Scout client server

applications. In case you are interested in getting some general context around Tomcat you could start with the Wikipedia article. [60: Apache Tomcat Wikipedia: http://en.wikipedia.org/wiki/Apache_Tomcat.]. Then get introduced to its core component ÒTomcat CatalinaÓ. [61: Mulesoft's introduction to Tomcat Catalina: http://www.mulesoft.com/tomcat-catalina.] before you switch to the official Tomcat homepage. [62: 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 7.0. Start your download from the official download site. [63: Tomcat 7 Downloads: http://tomcat.apache.org/download-70.cgi].

Figure 69. A successful Tomcat 7 installation.

Once you have downloaded and installed Tomcat 7 (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. [64: Tomcat Windows setup:

http://tomcat.apache.org/tomcat-7.0-doc/setup.html#Windows] download the package Ò32-bit/64-bit Windows Service InstallerÓ from the Tomcat 7 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 7 download site. Then, follow the installation guide. [65: Installing Tomcat on OS X: http://wolfpaulus.com/journal/mac/tomcat7] provided by Wolf Paulus.

For Linux systems download the Ò.tar.gzÓ package from the Tomcat 7 download site. Then, follow the description of the Unix setup. [66: Tomcat Linux setup: http://tomcat.apache.org/tomcat-7.0-doc/setup.html#Unix_daemon] to run Tomcat as a deamon. If you use Ubuntu, you may want to follow the tutorial. [67: 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

TomcatÕs 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.

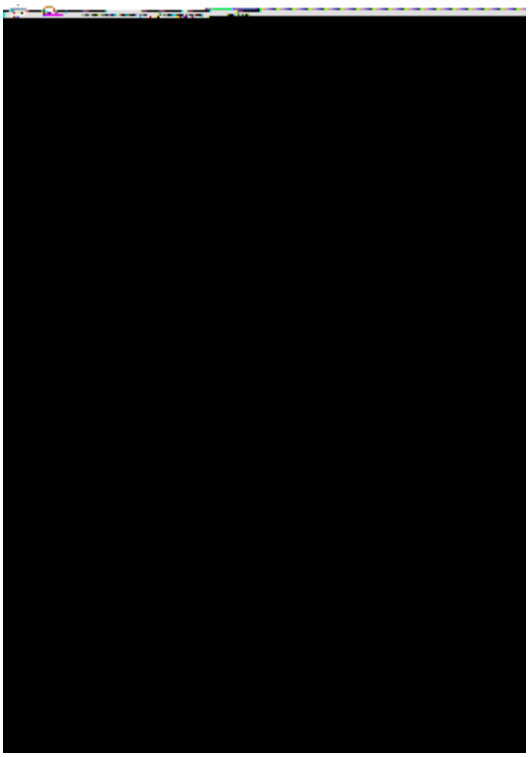


Figure 70. The organisation of a Tomcat installation including specific files of interest. As an example, the Ò Hello WorldÓ server application is contained in subdirectory webapps.

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 ÒcacheÓ 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 ÒManager AppÓ. 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 ÒManager AppÓ can be found under the Tomcat homepage. [68: The Tomcat Manager Application: http://tomcat.apache.org/tomcat-7.0-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 ÒManager AppÓ 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.

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

Appendix D: Scout Utilities

text needed.

1. <ctrl-shift-t>fileutility
2. click into package org.eclipse.scout.commons;
3. <alt-shift-w> (or context menu) show-in package explorer

D.1. StringUtility

text needed

also mention apache StringUtils http://commons.apache.org/lang/api-2.3/org/apache/commons/lang/StringUtils.html

D.2. DateUtility

text needed

D.3. FileUtility

text needed

Existing Documentation

¥ bug https://bugs.eclipse.org/bugs/show_bug.cgi?id=394784

Appendix E: Java Basics

E.1. Java SE Basics

TIP	The goal of this section is to provide the reader with a solid overview of the non-trivial Java concepts relevant for scout applications and central aspects of the framework itself. The focus of this section is on the Java Standard Edition (Java SE). Where appropriate, provide links to high quality online material, that is likely to exist for at least the next year or two.

E.1.1. Learning Java

To progam Scout applications you need to have a solid understanding of the Java language. Scout will only work for you if you have achieved a certain proficiency level in Java.

Luckily, free online tutorials to learn Java are offered in many places. A good starting point is the official Java documentation site. [69: Official online Java tutorial: http://docs.oracle.com/javase/tutorial/]. If you prefer to work with video tutorials we recommend ÒEclipse and Java for Total BeginnersÓ. [70: Eclipse and Java for Total Beginners: http://eclipsetutorial.sourceforge.net/totalbeginner.html], although the installation used is somewhat out of date. As for printed books, we suggest to start with either ÒHead First JavaÓ\cite{batessierra05} or ÒThinking in JavaÓ\cite{eckel06}. Highly recommended but slightly more advanced is ÒEffective JavaÓ\cite{bloch08}.

To solve really tricky Java problems there is often no way around the Java specification. [71: The Java Language Specification http://docs.oracle.com/javase/specs/] itself. Just make sure to pick the right Java version for your context.

E.1.2. Advanced Java SE Concepts

- say which non-trivial things are vital to good understanding
- threading
- generics
- annotations

E.1.3. JAR Files

- purpose
- directory structure
- example

E.2. Java EE Basics

TIP

The goal of this section is to provide the reader with a solid overview of the non-trivial Java enterprise concepts relevant for scout applications and central aspects of the framework itself. The focus of this section is on the Java Enterprise Edition (Java EE) Where appropriate, provide links to high quality online material, that is likely to exist for at least the next year or two.

needs text

- ¥ maybe the same as for java foundation, maybe not
- ¥ jaas
- ¥ http comm
- ¥ servlet
- ¥ servlet filters

E.2.1. Servlets

A very comprehensive and detailed step to step description has been written by Chua Hock-Chuan. [72: Get Start with Java Servlet Programming: http://www.ntu.edu.sg/home/ehchua/programming/howto/Tomcat_HowTo.html.].

may be found online do servlet stuff with annotations (JEE6) not JEE5?

Listing 32. The index.html start page for the tiny servlet application.

```
<html>
<head>
<title>tiny servlet title</title>
</head>
<body style="color:green">
<!-- link must correspond to url-pattern in servlet-mapping of web.xml -->
go to <a href="servlets/Tiny">tiny servlet</a>
</body>
</html>
```

Listing 33. The web.xml file of the tiny servlet application.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
Ê <display-name>tiny servlet app</display-name>

Ê <!-- declaration of servlet for this app -->
Ê <servlet>
Ê   <!-- unique servlet symbolic name -->
Ê   <servlet-name>Tiny</servlet-name>
Ê   <!-- full path to class (relative from WEB-INF/classes) -->
Ê   <servlet-class>TinyServlet</servlet-class>
Ê </servlet>

Ê <!-- mapping of url to servlet defined above -->
Ê <servlet-mapping>
Ê   <servlet-name>Tiny</servlet-name>
Ê   <url-pattern>/servlets/Tiny</url-pattern>
Ê </servlet-mapping>
</web-app>
```

Listing 34. The complete TinyServlet source code.

```
// servlet browser link http://localhost:8080/tinyservlet/
// direct link to servlet: http://localhost:8080/tinyservlet/servlets/Tiny

// tomcat overview: http://localhost:8080/
// application admin: http://localhost:8080/manager/html/list

// compiling
// servlet jar has to be found in WEB-INF/lib/ (to be found in apache servlet container,
as well for compiling above)
// mzi@bsim013 ~/Desktop/jee/servlet1/WEB-INF/sources
// (0) cd /cygdrive/C/Documents\ and\ Settings/mzi/Desktop/jee/tinyservlet/WEB-
INF/sources
// (1) /cygdrive/C/java/jdk1.5.0_16/bin/javac -classpath
"../lib/javax.servlet_2.4.0.v200806031604.jar;." TinyServlet.java
// (2) mv TinyServlet.class ../classes

// deploying
// servlet jar has to be found in WEB-INF/lib/ (to be found in apache servlet container,
as well for compiling above)
// (1a) zip contents of C:\Documents and Settings\mzi\Desktop\jee\tinyservlet
// (1b) rename to tinyservlet.zip to tinyservlet.war
```

```
// (2) copy war to C:\tomcat\tomcat70\webapps
// (3) restart tomcat (or remove folder tinyservlet from webapps and restart with war
file only)

import java.util.Date;
import java.io.IOException;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;

public class TinyServlet
Ê extends HttpServlet
{
Ê public void service(HttpServletRequest request, HttpServletResponse response)
Ê   throws IOException, ServletException
Ê {
Ê   response.setContentType("text/html;charset=ISO-8859-1");
Ê   response.getWriter().println(
Ê     "<html>"+
Ê     "<head><title>tiny servlet</title></head>"+
Ê     "<body style=\"text-align:center\">" +
Ê     "TinyServlet's server time: " + new Date() +
Ê     "</body>" +
Ê     "</html>"
Ê   );
Ê }
}
```

DONT include servlet jar inside of war file (tomcat doesnÛt like it)

E.2.2. Servlet Filters

hello world example (JEE6)

E.2.3. WAR Files

war file organisation: http://documentation.progress.com/output/Iona/orbix/6.1/tutorials/fnb/dev_intro/j2ee_overview8.html

Appendix F: Eclipse Basics

F.1. Eclipse as an IDE

Excellent Eclipse IDE tutorial by L. Vogel http://www.vogella.com/articles/Eclipse/article.html.

F.1.1. Project Workspace

F.1.2. Perspectives

A perspective contains the visual elements and the arrangement of those elements to support a specific development task within the Eclipse IDE. Perspectives relevant to the development of Scout applications are the Scout perspective, the Java perspective, the Debug perspective, and many others. To open a perspective available in the Eclipse IDE, the [!Open Perspective!] button or the Window | Open Perspective | OtherÉ menu can be used.

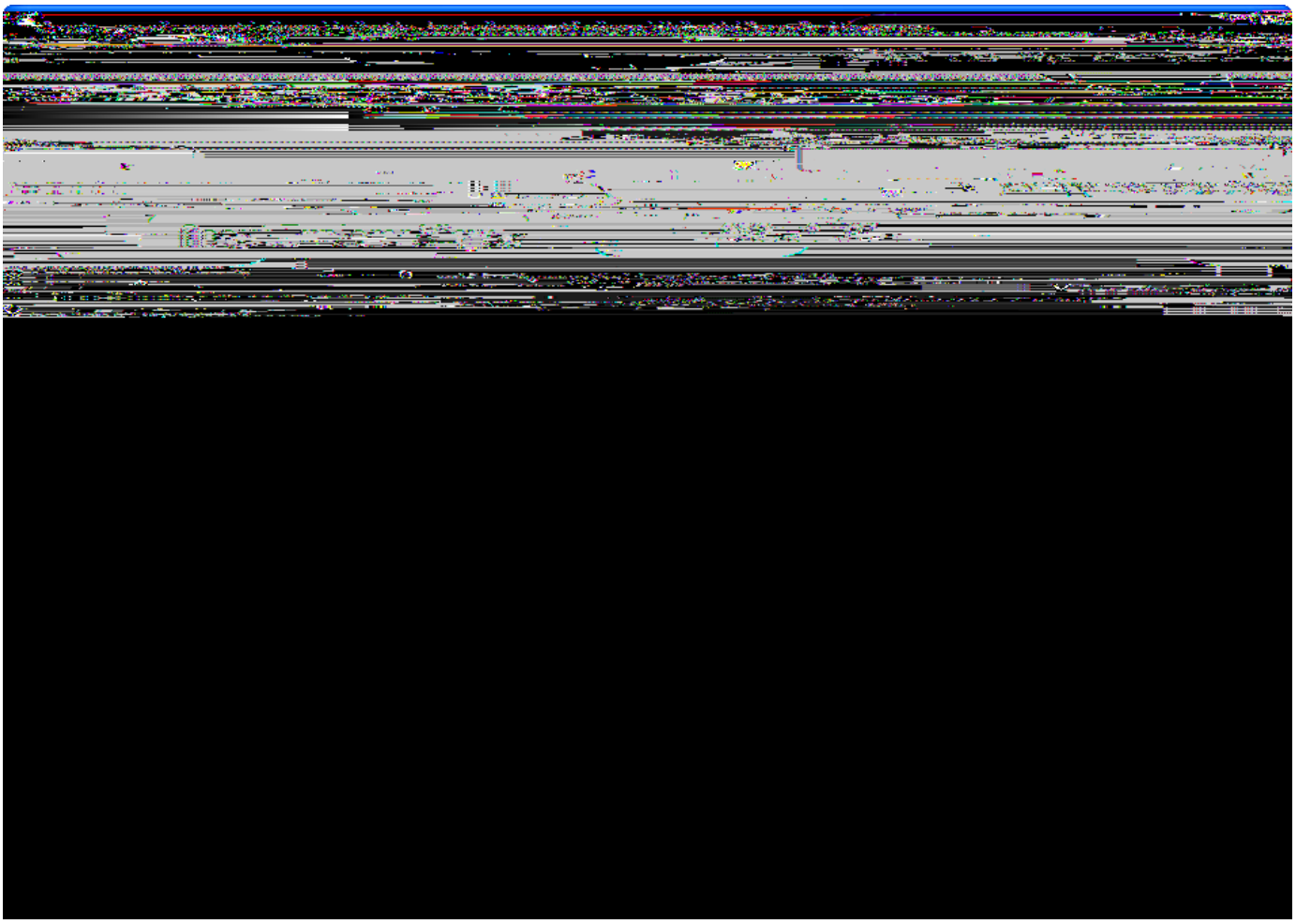


Figure 71. The Eclipse IDE with the Scout perspective. The colors indicate the different elements. View parts (blue), editor parts (green) and perspective buttons (purple).

Figure 000 provides a screenshot of the Eclipse Scout perspective indicating the corresponding perspective button and the main view parts and editor parts involved. Using drag and drop, views and editors can be freely moved around in the Eclipse IDE to suit the developerÕs needs. Perspectives can be further individualized using the Window | Customize PerspectiveÉ menu. Here, the visibility of

the toolbar items and menu entries can be defined. Once a suitable layout of all desired elements has been defined, this organisation may be saved as a personal perspective using the Eclipse IDE Window | Save Perspective AsÉ menu.

In case a customizing step does not turn out as intended, with the Window | Reset PerspectiveÉ menu is always possible to go back to the last saved state of the current perspective.

F.2. OSGi and Equinox

TIP	The goal of this section is to provide the reader with a solid overview of OSGi concepts and its Equinox implementation. Where appropriate, provide links to high quality online material, that is likely to exist for at least the next year or two.

What is OSGi: http://www.osgi.org/Technology/WhatIsOSGi What is Equinox: http://www.eclipse.org/equinox/

Server-side Equinox: http://www.eclipse.org/equinox/server/http_in_container.php

The web.xml, the lib/servletbridge.jar and eclipse/plugins/servlet, equinox and bla stuff

bundle example

needs text

- ¥ bundles
- ¥ services
- ¥ classloading

F.3. Eclipse

TIP	The goal of section is to provide the reader with a solid overview of standard Eclipse concepts relevant for scout projects and central parts of the Scout framework and Scout SDK tooling. where appropriate, provide links to high quality online material that is likely to exist for at least the next year or two

needs text

F.4. Eclipse Plugins

release engineering artefacts vs runtime artefacts. start with runtime artefacts

- ¥ plugins
- ¥ fragments

- features
- products
- targets
- servlet bridge
- client exe files