BAHBAH TUTORIAL

Implement a Multi-Frontend Chat Application based on Eclipse Scout

http://www.eclipse.org/scout/

24.10.2012

Authors: Matthias Zimmermann, Matthias Villiger, Judith Gull

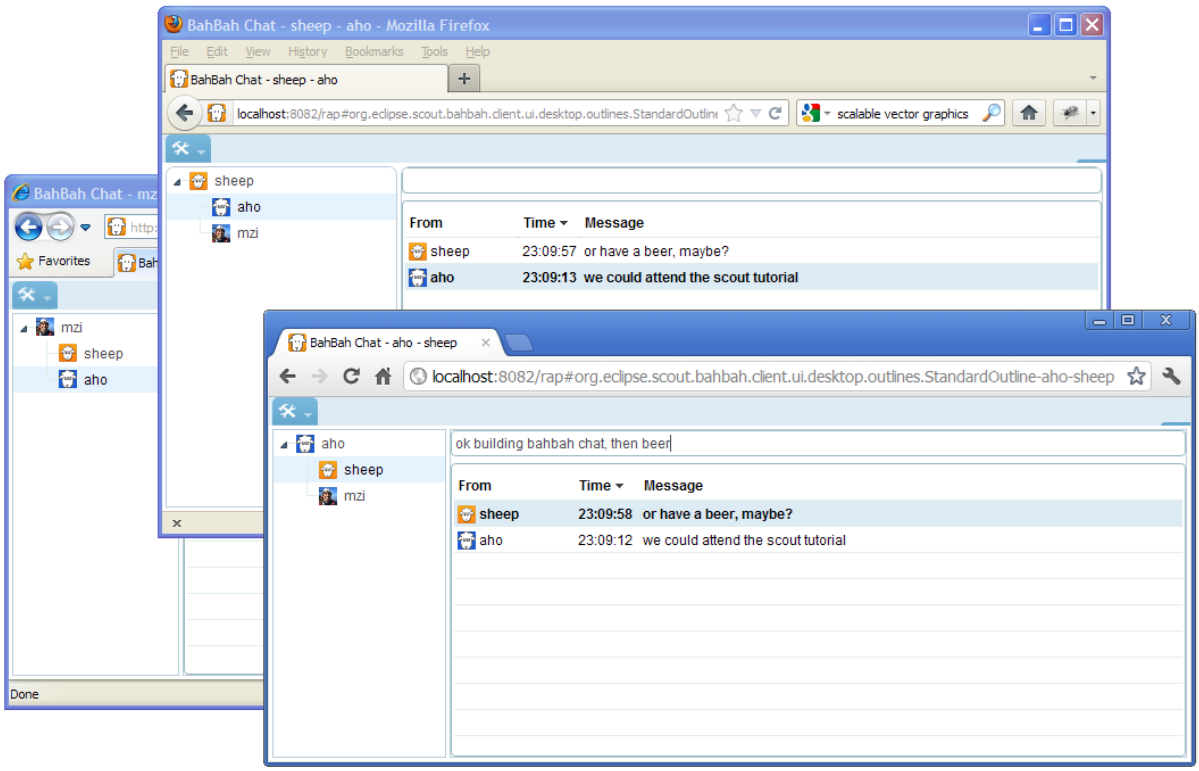


TABLE OF CONTENTS

1 PREREQUISITES

In order to do this tutorial you need

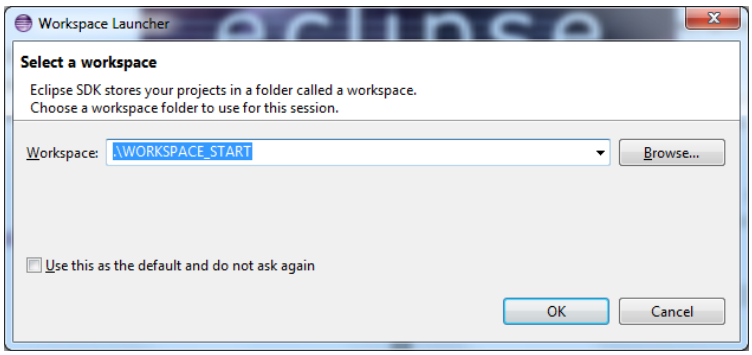
- The Bahbah package delivered with this tutorial depending on your operating system:
  - Bahbah_eclipse_win32.exe (for Windows)
  - BahBah_eclipse_macosx.zip
  - Bahbah_eclipse_linux_32.tar.gz (for Linux 32 bit)
  - Bahbah_eclipse_linux_64.tar.gz (for Linux 64 bit)
- JDK (1.6 or later) for macosx and Linux. In the Windows package, a JDK is included in the Bahbah package.

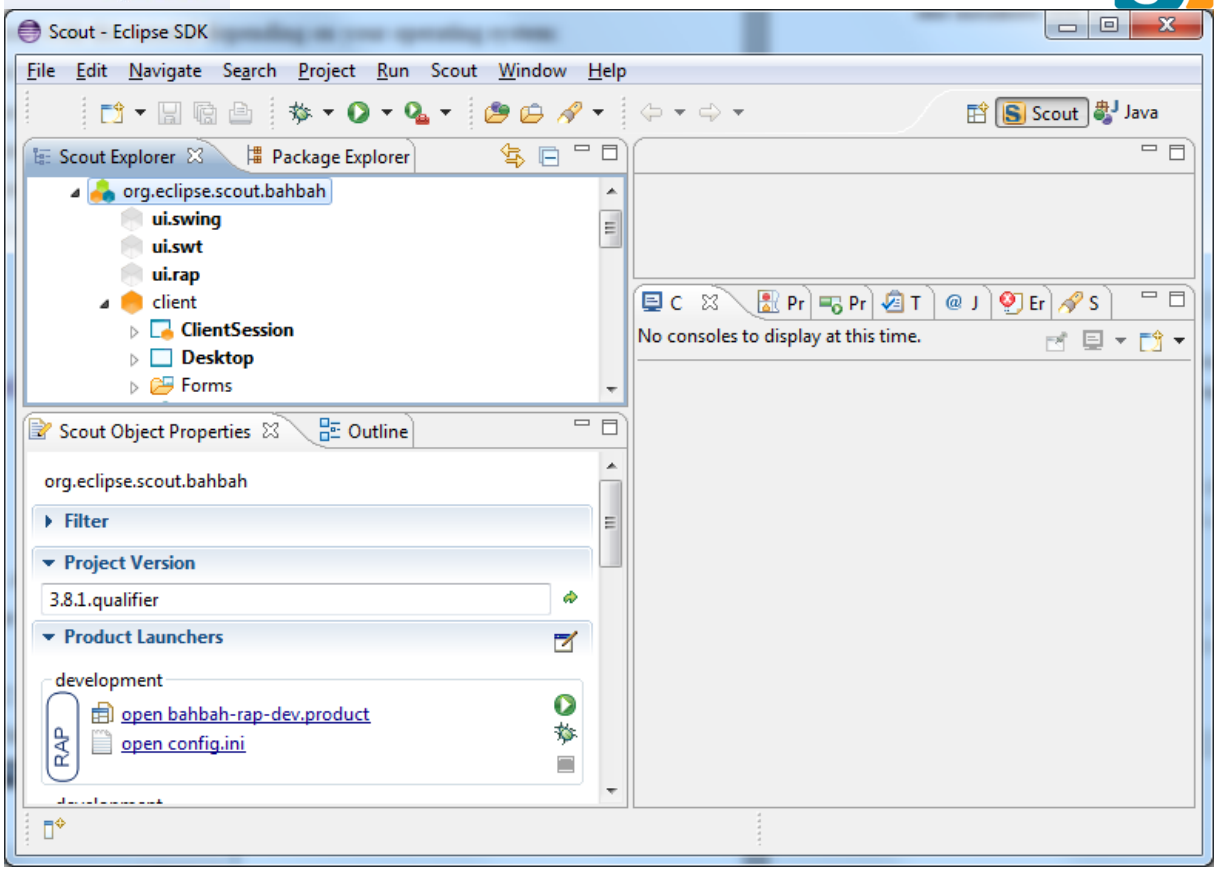
The Bahbah package contains

- Eclipse 3.8.1 (http://download.eclipse.org/eclipse/downloads/drops/R-3.8.1-201209141540/)
- Subversive SVN (http://www.eclipse.org/subversive/)
- Scout SDK 3.8.1 (Juno) (http://www.eclipse.org/scout)
- TARGET_SCOUT39: (some scout plugins that will be delivered with Scout 3.9.0 for mobile support)
- TARGET_RAP: (Version 1.5, see http://www.eclipse.org/rap/)
- WORKSPACE_FINISHED: A workspace with the finished Bahbah application.
- WORKSPACE_START: The initial workspace for this tutorial
- apache-tomcat-6.0.35.zip (see http://tomcat.apache.org/)

1.1.1 Starting the Bahbah Package

Extract the package to a folder (e.g. C:\dev\bahbah_eclipse for windows) and start the eclipse executable. Select the workspace *WORKSPACE_START* and proceed with ok.

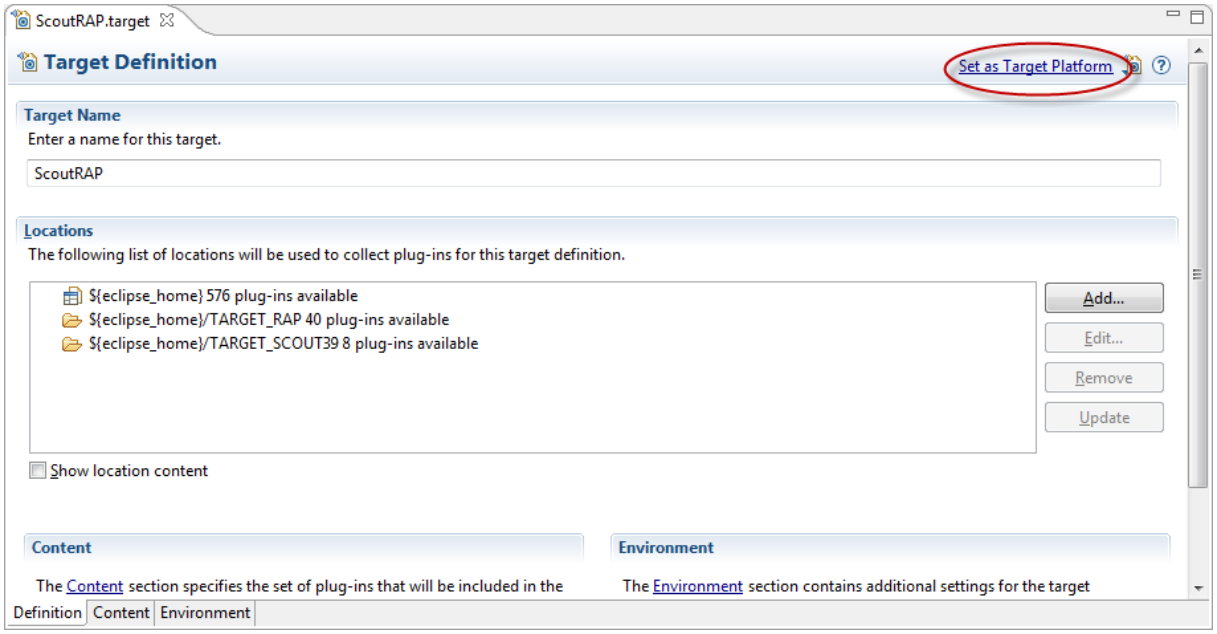




Eclipse opens with the initial Bahbah workspace. The perspectives Scout and Java are available containing the Scout project org.eclipse.scout.bahbah.

1.1.2 Compiling and Running the Initial Application

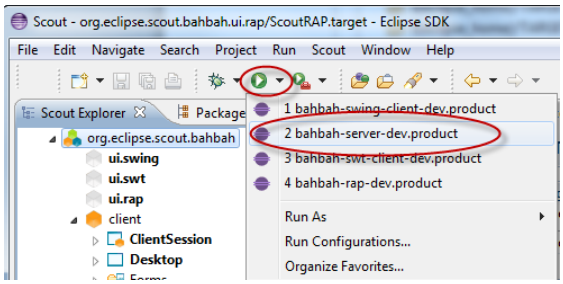
To compile Bahbah you need to set the target platform to ScoutRAP.target: Open the resource editor CTRL+SHIFT+R and select the file *ScoutRap.target*. Click on *Set as Target Platform!*



This triggers an automatic recompilation of the workspace. After the compilation is finished there should be no compilation errors.

Now you are ready to start Bahbah:

First start the server using the run configuration bahbah-server-dev.product:



You can see in the console that the server is running:

```
…
!ENTRY org.eclipse.scout.bahbah.server 1 0 2012-10-20 10:37:11.073
!MESSAGE
org.eclipse.scout.bahbah.server.ServerApplication.start(ServerApplication.java:56)
bahbah server initialized
```

Now you can start one or more clients. The available products are:

- bahbah-swing-client-dev.product (Rich Client SWING)
- bahbah-swt-client-dev.product (Rich Client SWT)
- bahbah-rap-dev.product (Web Client)

  Access in a browser with the URL
  - http://localhost:8082/web,
  - http://localhost:8082/mobile (Mobile Version)
  - http://localhost:8082/tablet (Tablet Version)

Login with

Username: admin

Password: admin

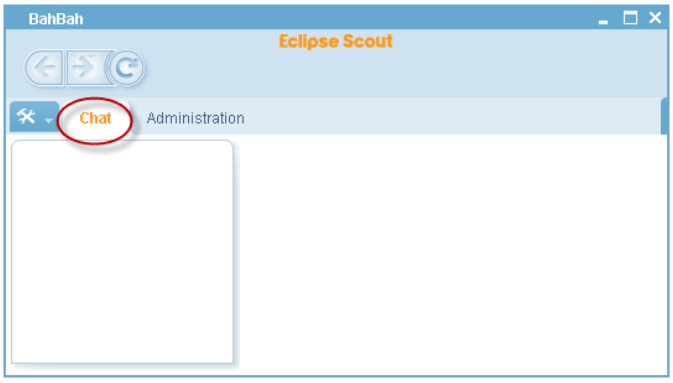
Once you logged in you can create and delete users in the administration view.

2 CREATING THE CHAT OUTLINE AND PAGES

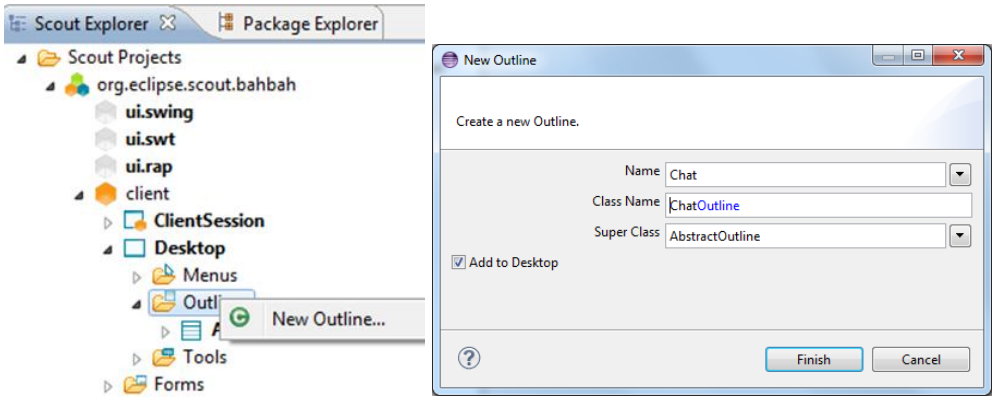
In this chapter we will extend the GUI of the Bahbah application with the help of the Scout SDK.

2.1 CHAT OUTLINE

The first goal is to create a new chat outline in the GUI for the chat conversations:



Add a new Outline to the desktop by right-clicking on the Node *org.eclipse.scout.bahbah/client/Desktop/Outlines* and selecting *New Outline…*

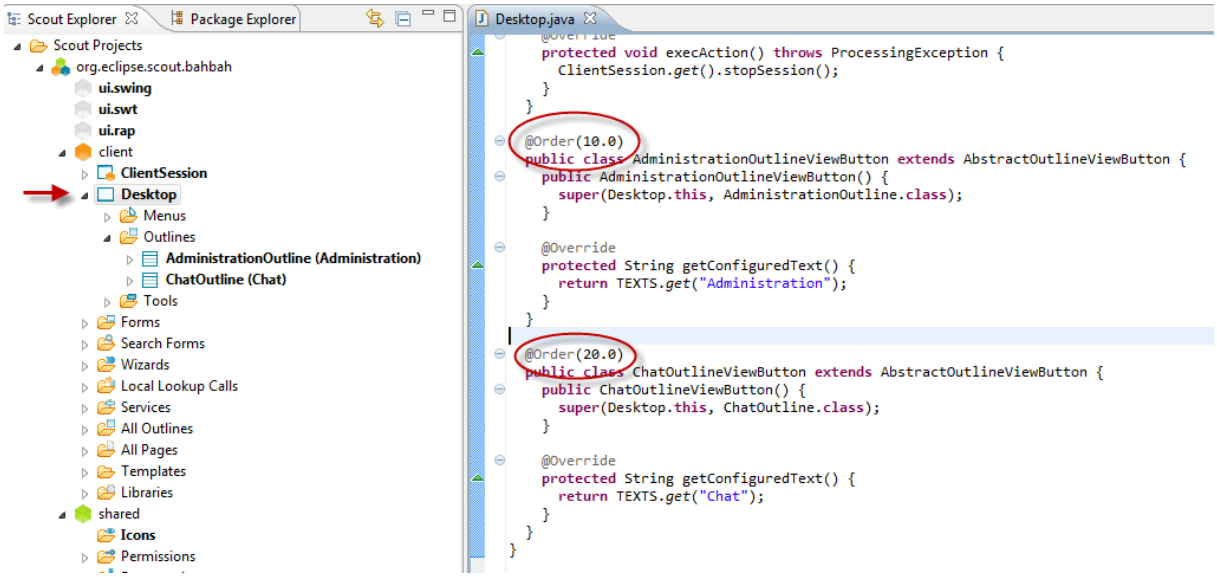


Enter *Chat* as Name and finish the wizard.

Now you can open the new class *ChatOutline* generated by the Scout SDK, which represents the client model for the chat outline by double-clicking the new *ChatOutline* in the Scout Explorer.

Scout SDK also added a new tool button in to the `Desktop`. Double-click on Desktop to open the corresponding class.

The order of the Buttons is given by the *@Order* Annotation. By default, new Buttons are added at the end. To display the *ChatOutlineViewButton* as the first button just change its order annotation to something smaller than the one of *AdministrationOutlineViewButton* (e.g. @Order(5.0)).



And because we want to have the just created Chat outline active when starting the application, we also change the order in which the outlines are added to the *Desktop*: In the *Desktop* class find the method *getConfiguredOutlines* and change the order in which the outlines are added so that the *ChatOutline* comes first:

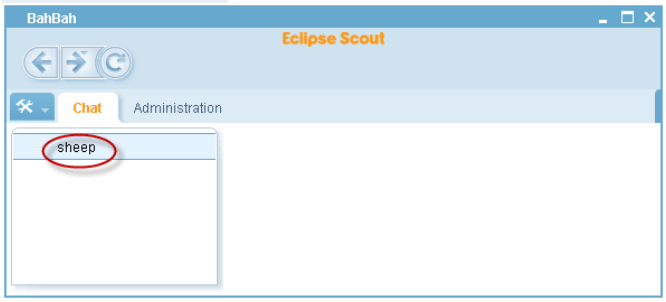
```
ArrayList<Class> outlines = new ArrayList<Class>();
outlines.add(ChatOutline.class);
outlines.add(AdministrationOutline.class);
return outlines.toArray(new Class[outlines.size()]);
```

At this point you might want to restart the clients (SWT, SWING and RAP Version) and see, if everything looks as expected.

Note that all the changes you have done so far are in the client model in the plugin org.eclipse.scout.bahbah.client. You do not have to write UI specific code.

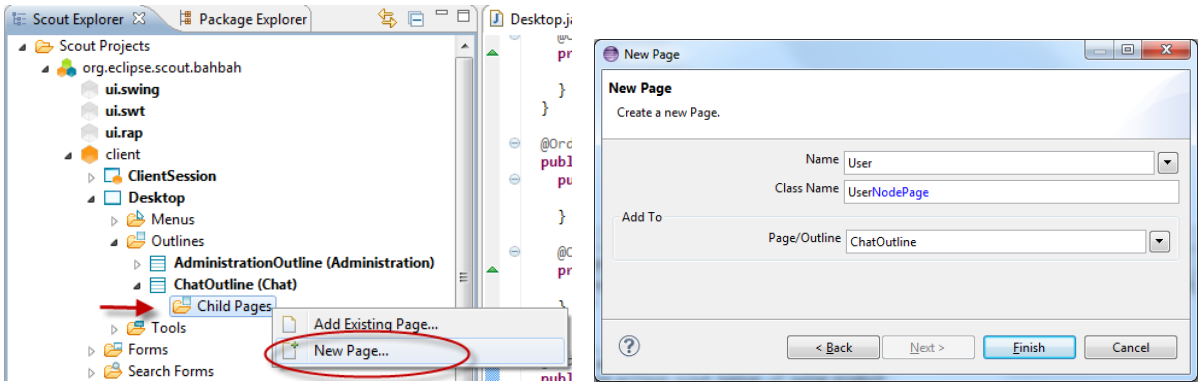
2.2 User Node Page

The next step is to add some data to the chat outline. To get an overview of the currently connected users we organize the chat outline tree in the following way: The top-level node represents the currently logged-in user while all the other connected users are displayed as child nodes. Let's start with the top-level node.



2.2.1 Adding a Node Page to the Outline

We can easily add this node by adding a new child page to the ChatOutline: Open the Scout Explorer and right-click on the subfolder *Child Pages* of *ChatOutline* and choose *New Page…*. Choose *AbstractPageWithNodes* as template for your page and proceed with Next. Type *User* in the Name field and Finish the Wizard.



2.2.2 Setting the Tree Node Text

Remember that we want the text of the tree node to represent the logged-in user. To set this text we have to edit the UserNodePage class:

Open the UserNodePage class by double-clicking the new node in the *Scout Explorer*. If the new page is not visible already, select the *Child Pages* node below *ChatOutline* in the *Scout Explorer* and press F5.

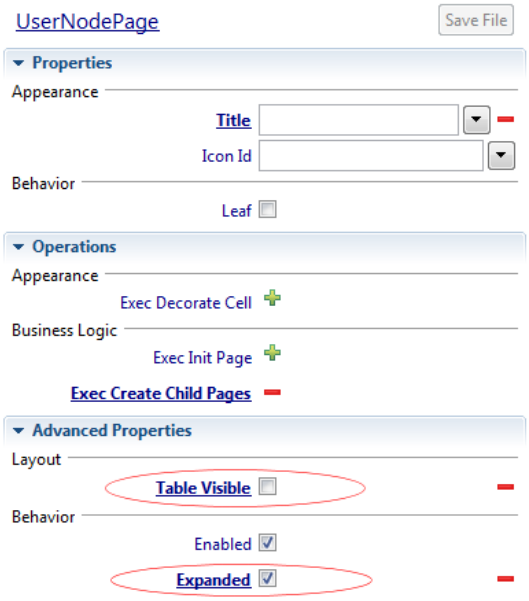
The text of the tree node is given by the title property of the page, which can be set by overriding the method `getConfiguredTitle`. Currently the title property is set to the translated text “User”.

Update the getConfiguredTitle method to return the username stored in the client session.

```
@Override
protected String getConfiguredTitle() {
  return ClientSession.get().getUserId();
}
```

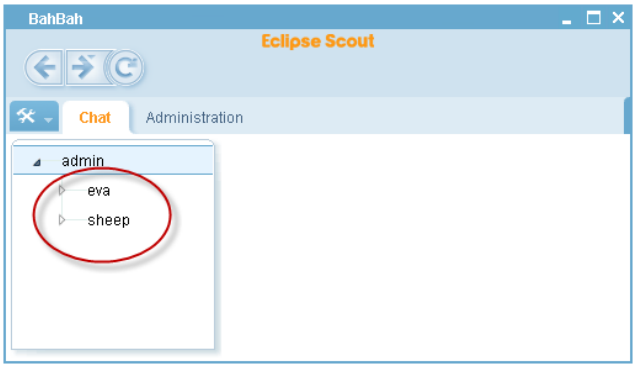
2.2.3 Configuring the Page

We do not have any data to be displayed for this node. Therefore we set the table invisible for this page: Click on the node *UserNodePage* in the Scout Explorer, open *Advanced Properties* in the *Scout Object Properties* View and uncheck the property *Table Visible*.
Furthermore we would like to have this page expanded by default. Therefore we also tick the check-box *Expanded*:



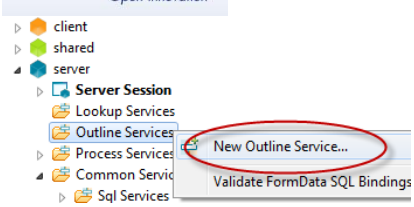
2.3 Adding Online Users to the Chat Outline

In this section we add all the currently connected users to the chat outline.

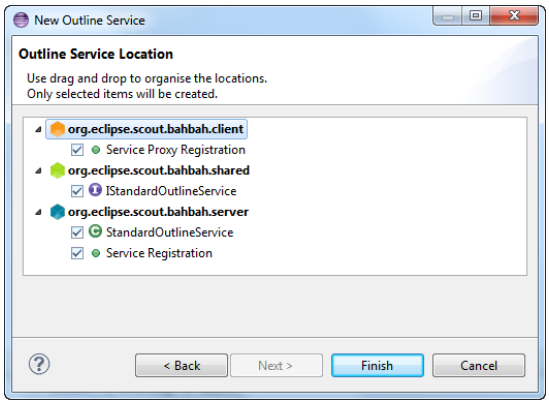


2.3.1 Creating an Outline Service for Connected Buddies

First we prepare a service to retrieve all connected users, except the logged-in user, i.e. the buddies: Expand the *server* node in the Scout Explorer and create a new outline service. Enter a class name and click *next*.

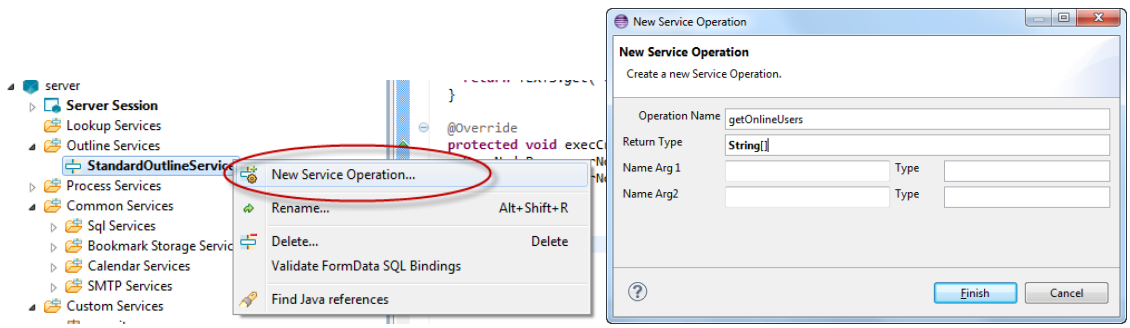


The following screen shows an overview of the service, its interface and registration. In the example below the StandardOutlineService class contains the actual implementation of the service. The interface IStandardOutlineService is located in the shared plugin and therefore visible in the client and server plugin. Additionally, the service is registered in the OSGi service registry of the server and a service proxy is created and registered on the client side. Press *Finish* to create the service.



2.3.2 Implementing a Service Operation

In order for the service to do something useful we need to add a service operation. Select *New Service Operation…* in the Scout Explorer and enter the name and return type of the operation as shown in the screenshot:



The Scout SDK simply adds a new public method to the service class and interface. Open the service class to implement the method.

In the initial workspace there already exists a service operation to get all online users. (See UserProcessService). We only need to call this operation and remove the logged-in username.

The class org.eclipse.scout.service.SERVICES contains useful convenience methods to get services. E.g. IUserProcessService svc = SERVICES.getService(IUserProcessService.class); finds the service with the interface IUserProcessService in the registry.

```
//get the service
IUserProcessService svc = SERVICES.getService(IUserProcessService.class);
//find all online users
Set<String> allOnlineUsers = svc.getUsersOnline();
```

On the server, the logged in user is stored in the server session:

```
String user = ServerSession.get().getUserId();
```

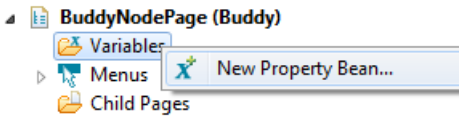
Putting it all together we end up with something like this:

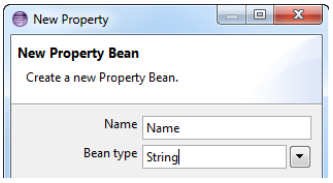
```
@Override
public String[] getOnlineUsers() throws ProcessingException {
  Set<String> allUsers = SERVICES.getService(IUserProcessService.class).getUsersOnline();
  Set<String> users = new HashSet<String>(allUsers);
  // remove myself
  users.remove(ServerSession.get().getUserId());
  return users.toArray(new String[users.size()]);
}
```

2.3.3 Creating and Configuring a NodePage for the Connected Users

Now we are ready to update the client. We create another NodePage for the buddies as a child page of the *UserNodePage*: Open the Scout Explorer and right-click on the subfolder *Child Pages* of *UserNodePage* and choose *New Page…*. Select *AbstractPageWithNodes* as template for your page and proceed with Next. Type *Buddy* in the Name field and Finish the Wizard.

We add a property to the BuddyNodePage to later set the name. Right-click on Variables of the BuddyNodePage and select *New Property Bean…* and create a property with name Name and *Bean type* String.





Change the method getConfiguredTitle to return the name property.

```
@Override
protected String getConfiguredTitle() {
  return getName();
}
```

Check *Leaf* in the Scout Object Properties View for the *BuddyNodePage* to indicate that the node never contains any children and uncheck *Table Visible*, because no table will be needed for this page.

2.3.4 Adding an Instance of the NodePage for Every Online User

In the last section we added a single BuddyNodePage as a child page of the UserNodePage. However, instead of only one *BuddyNodePage* we would like to have one instance per buddy.

The child pages are added to the UserNodePage in the method *execCreateChildPages*. Open the UserNodePage class to edit this method.

To get the buddies we can call the service operation created in chapter 2.3.2. Remember that a service proxy was registered on the client when we created the service. We can again use the SERVICES class to get the registered service proxy and receive the data from the server by calling the service operation:

```
String[] buddies = SERVICES.getService(IStandardOutlineService.class).getOnlineUsers();
```

Now it's your turn to implement `execCreateChildPages(Collection<IPage> pageList)`, such that all buddies are added to the pageList. Do not forget to set the name of the page.

Start multiple clients and verify that the connected users are now displayed in the chat outline.

2.3.5 Dynamically Adding and Removing Buddies upon Connect and Disconnect

So far the connected buddies are only evaluated once when the page is instantiated. In this section we add the functionality to update the outline dynamically.

We implement this functionality using client notifications: Whenever a user connects or disconnects the server sends a notification event to all clients (see `NotificationProcessService`). When the client receives such a notification it refreshes the outline page.

The method *updateBuddyPages* below handles the refresh of the outline and adds or removes nodes, if needed. Copy the following code snippet to the UserNodePage:

```
public void updateBuddyPages() throws ProcessingException {
  HashSet<String> newBuddy = new HashSet<String>();
  ArrayList<String> updatedList = new ArrayList<String>();
  String[] buddies = SERVICES.getService(IStandardOutlineService.class).getOnlineUsers();

  for (String buddy : buddies) {
    newBuddy.add(buddy);
  }

  // keep track of known buddies and remove buddies that are no longer here
  for (IPage page : getChildPages()) {
    BuddyNodePage buddyPage = (BuddyNodePage) page;

    if (newBuddy.contains(buddyPage.getName())) {
      updatedList.add(buddyPage.getName());
    }
    else {
      getTree().removeChildNode(this, buddyPage);
    }
  }
```

```
    // add new buddies
    for (String buddy : newBuddy) {
      if (!updatedList.contains(buddy)) {
        BuddyNodePage buddyPage = new BuddyNodePage();
        buddyPage.setName(buddy);
        getTree().addChildNode(this, buddyPage);
      }
    }
  }
```

The client service `BahBahNotificationConsumerService` handles the client notification events. Update the methods `handleRefreshBuddies()` and `getUserNodePageFromDesktop()` as shown below.

```
  private void handleRefreshBuddies() {
    UserNodePage userPage = getUserNodePageFromDesktop();

    if (userPage != null) {
      try {
        Logger.info("refreshing buddies on client");
        userPage.updateBuddyPages();
      }
      catch (Throwable t) {
        Logger.error("handling of remote message failed.", t);
      }
    }
  }

  public UserNodePage getUserNodePageFromDesktop() {
    Desktop d = Desktop.get();
    if (d != null) {
      return d.getUserNodePage();
    }
    return null;
  }
```

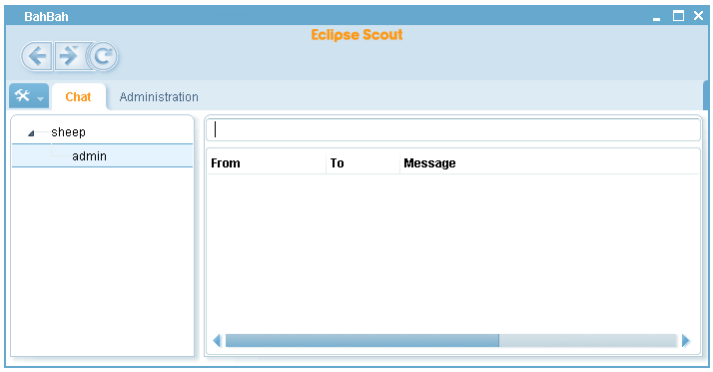
Finally add `getUserNodePage()` and `getChatOutline()` to the Desktop class:

```
  public UserNodePage getUserNodePage() {
    IPage invisibleRootPage = getChatOutline().getRootPage();
    if (invisibleRootPage != null && invisibleRootPage.getChildNodeCount() > 0) {
      IPage p = invisibleRootPage.getChildPage(0);
      if (p instanceof UserNodePage) {
        return (UserNodePage) p;
      }
    }
    return null;
  }

  private IOutline getChatOutline() {
    for (IOutline o : getAvailableOutlines()) {
      if (o.getClass().equals(ChatOutline.class)) {
        return o;
      }
    }
    return null;
  }
```

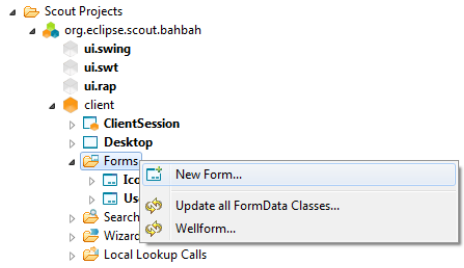
3 CHAT FORM

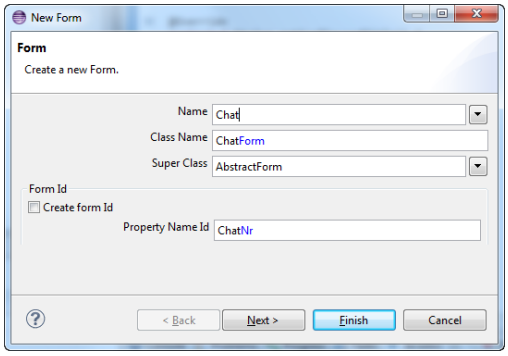
The next goal is to create a form to enter the chat message and display the history of the conversation.



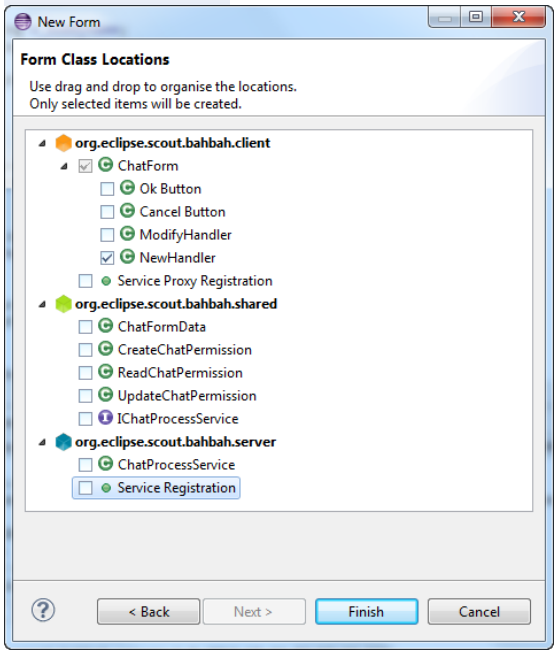
3.1 CREATING A NEW FORM

Select *New Form…* in the Scout Explorer and enter *Chat* as name. You can uncheck the check box *Create form Id*, because the default form id property is not needed. Click Next to proceed.

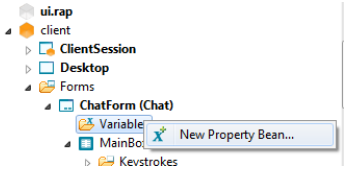




Unlike most common forms the chat form is not used to enter data to be persisted on the backend or display stored data. It is only used to display messages received by client notifications and send messages to other users. We therefore do not want to create the default process services and handlers. Uncheck everything except *New Handler* and finish the wizard.



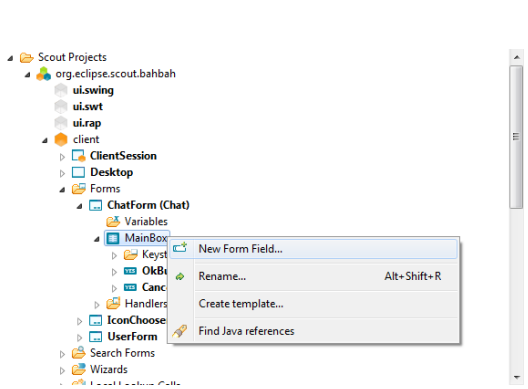
In this form we will need the buddy name and user name to send and display messages: Create the bean properties *userName* and *buddyName* (both of type String) for the *ChatForm* using the menu *New Property Bean…* in the *Scout Explorer*.

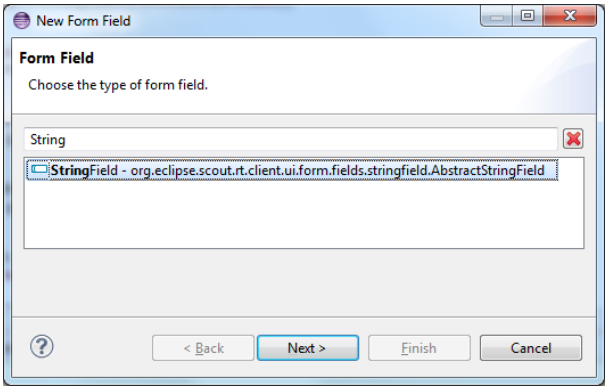


3.1.1 Adding Fields to the Form

Now let's add a field to enter the message to the form:

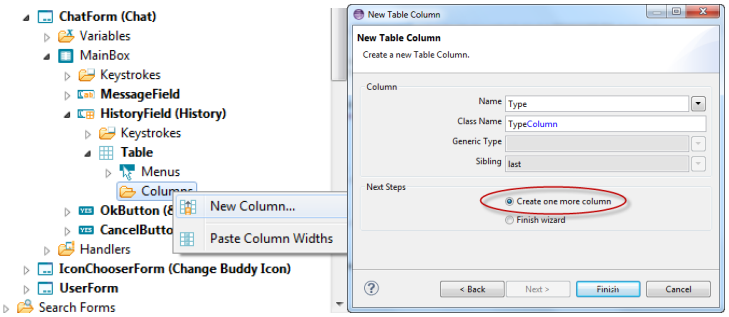
Select *New Form Field… on Main Box*, choose the type *StringField* and click next. Enter the class name MessageField (leave the Name empty) and Finish the Wizard.





No label is needed for the message field. Click on the new node *MessageField* in the Scout Explorer View and uncheck the property *Label Visible* in the section *Advanced Properties*.

We add another field for the history of the conversation: Select *New FormField…* on *MainBox* and create a field with type *TableField* and class name *HistoryField*. Uncheck the property *Label Visible* for the *HistoryField* as well.

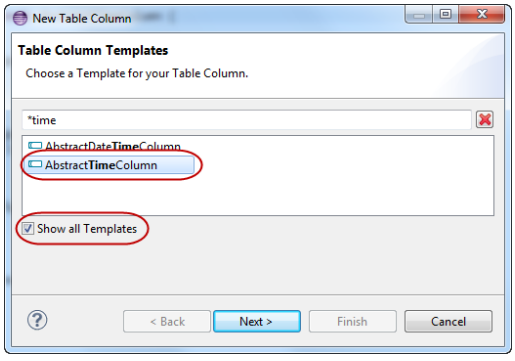


To add columns to the history table expand the node HistoryField in the Scout Explorer and select *New column…* Select *Integer Column* as template and Type as column name. To add additional columns you can select *Create one more column* as the next step to the column wizard.

Add the following columns to the table:

Name	**Type**	**Remarks**
Type	Integer	
Sender	String	
Receiver	String	
Message	String	
Time	Time	To find this type, tick the box as shown below.

Check *Show all Templates* to add the time column.

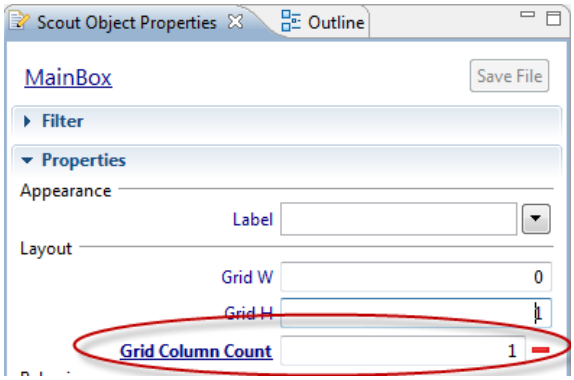


We set the some more properties for the columns in the Scout Object Properties View:

- Type Column: Uncheck displayable, because we only use this columns to hold data and do not actually display it

- Time Column: We want to sort the history in descending order, such that the last message appears on top: Uncheck *Sort Ascending* and set the *Sort Index* to 0 (both in the *Advanced Properties* section).
- Time Column: Set the *Format* to HH:mm:ss
- Message Column: Tick *Text Wrap*
- Table: Tick the property *Multiline Text*

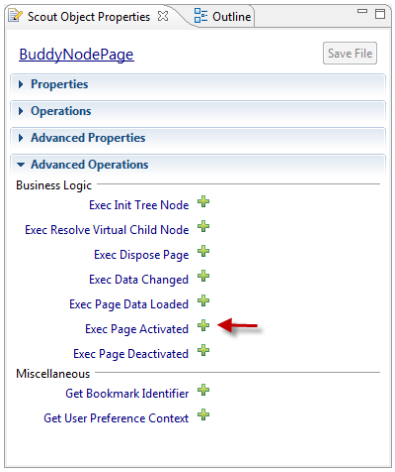
Finally, we change the layout of the form fields. By default the fields are organized in a grid with two columns. We change the *Grid Column Count* to 1 on the *MainBox*.



3.1.2 Opening the Chat Form

The chat form should be displayed on a click on the buddy node, i.e. when the *BuddyNodePage* is activated.

Open the *BuddyNodePage* in the ScoutExplorer and add *Exec Page Activated*.



A form can be added to a page by calling *setDetailForm*.

FormHandlers are used to start a form in different ways, e.g. for creating new data vs. for editing existing data. The chat form has only one handler (*NewHandler*), which is started by calling startNew().

The following code snippet contains a basic implementation of *execPageActivated*.

```
public ChatForm getChatForm() throws ProcessingException {
    if (m_form == null) {
      m_form = new ChatForm();
      m_form.setAutoAddRemoveOnDesktop(false);
      m_form.setUserName(ClientSession.get().getUserId());
      m_form.setBuddyName(getName());
      m_form.startNew();
    }
    return m_form;
  }


  @Override
  protected void execPageActivated() throws ProcessingException {
    super.execPageActivated();

    // after buddy page activation the buddy's chat history is displayed on the right side
    ChatForm chatForm = getChatForm();
    setDetailForm(chatForm);
  }
```

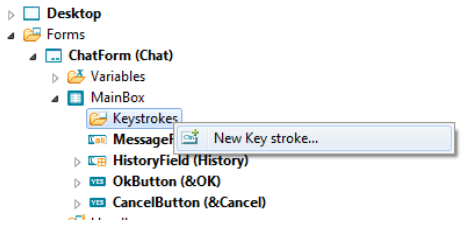
3.2 SENDING AND RECEIVING MESSAGES

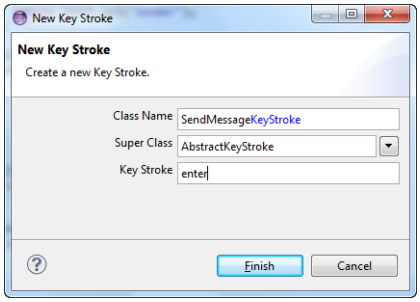
In this section we cover the actual sending and receiving of the chat messages.

3.2.1 Sending a Message

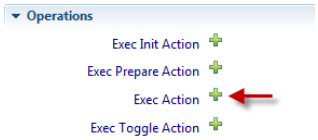
In order to send a message, a user types something in the message field and hits enter. To implement this functionality, we add a key stroke action to the form:

Open the *ChatForm* in the *Scout Explorer* and right-click on *Keystrokes* below the *MainBox*. Select *New Key stroke…* and enter *SendMessageKeyStroke* for the *Class Name* and *enter* for the Key Stroke.





Select the *SendMessageKeyStroke* in the Scout Explorer and add the operation *Exec Action* in the Scout Objects Properties View and implement the `execAction` such that the current value of the message field is sent.




```
@Order(10.0)
public class SendMessageKeyStroke extends AbstractKeyStroke {

  @Override
  protected String getConfiguredKeyStroke() {
```

```
      return "enter";
    }

    @Override
    protected void execAction() throws ProcessingException {
      // TODO send the current value of the message field to the server
    }
  }
```

Hints:

- Call `sendMessage` in `INotificationProcessService` to send the message
- `getMessageField().getValue(); //gets the current value of the message field`

3.2.2 Appending a Message to the History field

We create a new method to the history field to add a message to the table:

```
public class HistoryField extends AbstractTableField<HistoryField.Table> {

private final Integer MESSAGE_TYPE_LOCAL = 1;
private final Integer MESSAGE_TYPE_REMOTE = 2;

    public void addMessage(boolean local, String sender, String receiver, Date date, String message)
throws ProcessingException {
    getTable().addRowByArray(new Object[]{(local ? MESSAGE_TYPE_LOCAL : MESSAGE_TYPE_REMOTE), sender,
receiver, message, date});
     }
…
```

Now you can easily change the *execAction* Method in *SendMessageKeyStroke* such that the message is added to the history field. Restart the client to verify that the message is added to the history field.

3.2.3 Handling Received Messages

The sending and receiving of messages can be implemented using client notifications. The server sends a client notification containing the message to the receiving client. The client then displays the message.

Note that in contrast to the refreshing of the *RefreshBuddiesNotification* a *SingleUserFilter* is applied, such that only the receiving user needs to handle the message. (see *NotificationProcessService.sendMessage*).

Add the following code snippet to the *UserNodePage* to find the chat form by buddy name:

```
public ChatForm getChatForm(String buddy) throws ProcessingException {
  if (StringUtility.isNullOrEmpty(buddy)) {
    return null;
  }

  for (ITreeNode node : getChildNodes()) {
    BuddyNodePage buddyNode = (BuddyNodePage) node;
    if (buddyNode.getName().equals(buddy)) {
      return buddyNode.getChatForm();
    }
  }
  return null;
```

```
  }
```

Open *BahBahNotificationConsumerService* and implement *handleMessage* to display received messages in the correct chat form:

```
private void handleMessage(MessageNotification notification) {
    UserNodePage userPage = getUserNodePageFromDesktop();
    String buddy = notification.getSenderName();

    if (userPage != null) {
      try {
        ChatForm form = userPage.getChatForm(buddy);
        if (form != null) {
          form.getHistoryField().addMessage(false, buddy, form.getUserName(),
                                            new Date(), notification.getMessage());
        }
      }
      catch (Throwable t) {
        logger.error("handling of remote message failed.", t);
      }
    }
  }
```

4 BUDDY ICON & ROW DECORATION

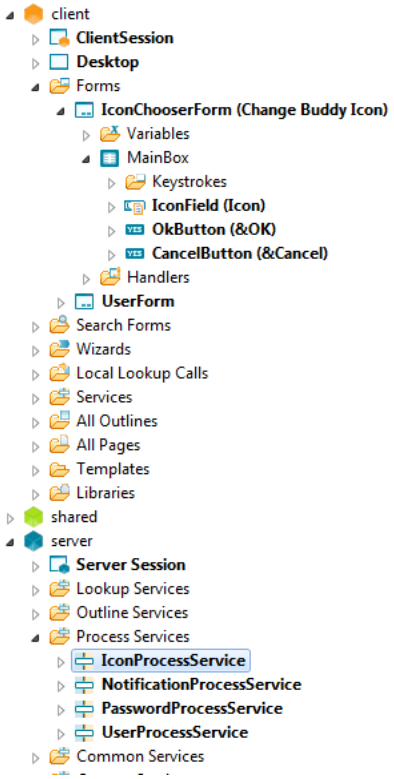
This chapter describes how to extend the Chat outline so that each user can change its own buddy icon. Furthermore we will improve the outline and chat history table to show these buddy icons. We will add some colors as well to better distinguish the two persons involved in a conversation.

4.1 EXPLANATION OF PREPARED ELEMENTS

Each user must be able to change its buddy icon. Therefore we need a possibility to store icons per user. The Derby database included in the workspace is already prepared for that.

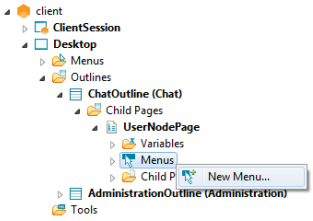
Press *Ctrl+Shift+T* to open the *Open Type* dialog. In the search field enter *dbsetup* and open the type by pressing *Ok*. In the attribute list of the create-table-statement you can see that there exists a *BLOB* (binary large object) type column named *icon*. So the database is ready to store the buddy icon.

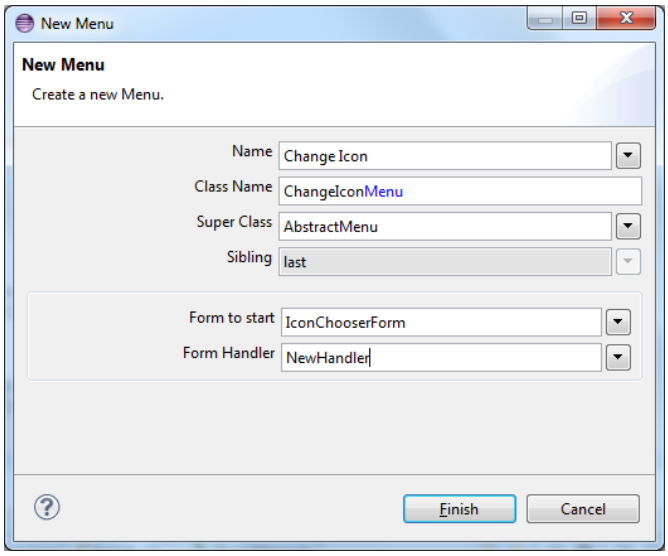
The workspace also contains a form capable of browsing locally to select an image file. You can find the form in the *Scout Explorer*: *client/Forms/IconChooserForm*. When saved, this form calls the *IconProcessService* which scales down the image and stores it in the BLOB column in the database.



4.2 CHANGING THE BUDDY ICON

So that we can make use of these prepared pieces we must add them to our chat application. In this tutorial we create a new context menu on the top-level node of the chat buddies tree on the left hand side of the application. To do so we navigate to the *client/Desktop/Outlines/ChatOutline/Child Pages/UserNodePage/Menus* node in the Scout Explorer and right-click on it. From the context we choose the entry *New Menu…* Fill the dialog as shown in the screenshot:





Press finish to create the menu entry. Below the *Menus* node in the *Scout Explorer* the new menu is shown. Select it and navigate to the *Scout Object Properties* view below. There remove the configuration for *Single Selection Action* and *Empty Space Action* by clicking on the red minus icon right to the properties. Below the *Operations* Section in the *Scout Object Properties* view press the green plus icon next to *Exec Prepare Action*. When the operation is created, it is displayed as a hyperlink. By clicking on it we will jump to the just created operation in the Java code. There replace the TODO comment with the following code snippet:

```
setVisible(UserAgentUtility.isDesktopDevice() && ACCESS.check(new UpdateIconPermission()));
```

This hides the menu entry if the user has no permissions to update icons or if we are running on a mobile or tablet client.

Now we have already added the change icon feature to our application. By activating the new context menu in the buddies tree a user can now update its icon. But because Scout caches icons for performance reasons, we will only see the new icon after we have restarted the client application.

4.3 DISPLAYING BUDDY ICONS IN PAGES & FORMS

As every user can now update its icon we would also like to show these icons in the buddies tree and the chat history table. To understand how this works, we will have a look at how icons are loaded in Scout.

Icons always have a unique name. If an icon with a certain name should be shown, Scout calls an *IconProviderService*. This service must then provide the necessary data for the requested icon name.

The initial workspace already contains an *IconProviderService* that can load icons from our Derby database. The default Scout implementation of that service loads icons from plugin resource folders. But because we would like to do both (load buddy icons from the database and load all other application icons from the plugin resources) we need to distinguish icon names that belong to a buddy from all other icons.

In this tutorial this is accomplished by a special prefix that is added to all buddy icons: If our *IconProviderService* gets a request for an icon with that prefix, it will return the icon stored in the database. Otherwise it will just delegate to the default implementation of Scout. This means that at everywhere we would like to show buddy icons we must remember to add this prefix!

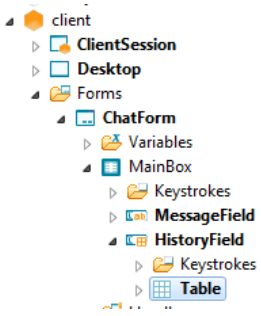
Now let's start showing the icons: Open the *client/All Pages/Buddy Node Page*. In the Java editor add the following code inside the class:

```
@Override
protected String getConfiguredIconId() {
  return BuddyIconProviderService.BUDDY_ICON_PREFIX + getName();
}
```

Now open client/All Pages/UserNodePage and add the following code:

```
@Override
protected String getConfiguredIconId() {
  return BuddyIconProviderService.BUDDY_ICON_PREFIX + ClientSession.get().getUserId();
}
```

Finally we navigate to *client/Forms/ChatForm/MainBox/HistoryField/Table*. In the Ad*vanced Operations* section of the *Scout Object Properties* view add the *Exec Decorate Cell* operation and jump to the code by clicking on the hyperlink after the creation. Inside the method add the following code:




```
// text color
row.setForegroundColor("000000");

// set icon id of the sender of the message (user icons have a specific prefix)
row.setIconId(BuddyIconProviderService.BUDDY_ICON_PREFIX + getSenderColumn().getValue(row));

// set font style and background color depending whether the message is from myself or not
boolean isMessageFromMe = MESSAGE_TYPE_LOCAL.equals(getTypeColumn().getValue(row));
if (!isMessageFromMe) {
```

```
  view.setFont(FontSpec.parse("BOLD"));
  row.setBackgroundColor("ddebf4");
}
```

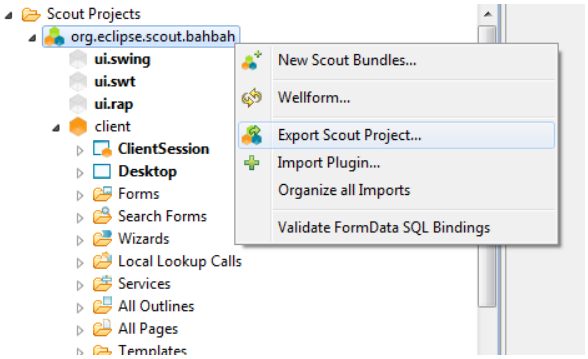
Now we can start the application and change our icon. Afterwards a second client can be started (choose a different user name). The new icon is then shown in the buddies tree on the left hand side and inside the chat window.

5 DEPLOYING THE APPLICATION TO TOMCAT

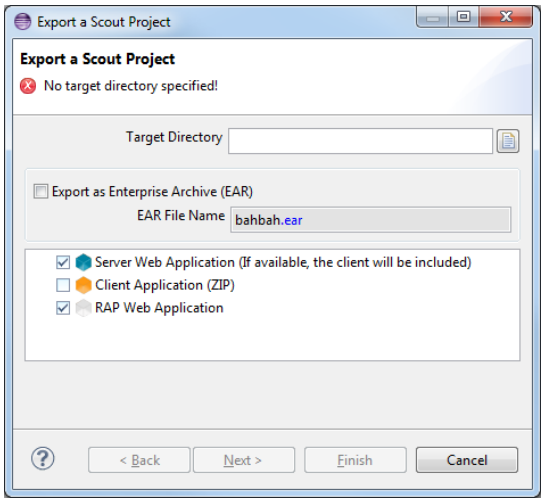
In this chapter we will deploy the created application to an Apache Tomcat servlet container. For this we will export it to two .war files:

1. The first containing the BahBah Server and a downloadable rich client.
2. The second containing the RAP UI used for browser, mobile & tablet access.

First navigate to the *org.eclipse.scout.bahbah* project node in the *Scout Explorer* and select *Export Scout Project…*.



In the dialog specify a target directory. In this folder the exported .war files will be saved. All other settings can be kept at its default as shown in the example.



Press *Next* and verify that the *War File* for the BahBah server is named *bahbah.war*. Press *Next* again.

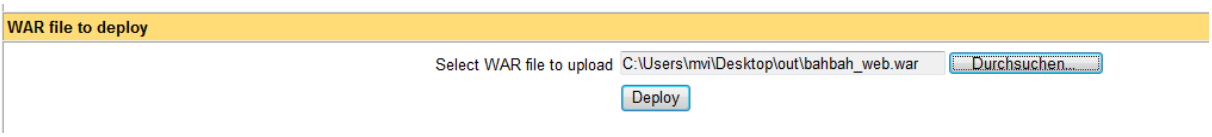
On this wizard page you can specify which rich client should be available to download when navigating to the BahBah web page. From the *Client Product to Include* choose one of the products marked with (production). You can choose the SWT or Swing client. Keep the default settings for *Client Download Location* and press *Next* one last time.

In the last wizard page ensure that the *WAR File* is named *bahbah_web.war* and press *Finish*. The selected products are now exported and packaged into two .war files that will be stored at the target directory specified in the first step.

On the USB sticks provided you can also find a Tomcat 6 servlet container. Copy it to your hard drive and start the server using *<Tomcat Root>/bin/startup.bat* or *<Tomcat Root>/bin/startup.sh*. Open a

web browser and navigate to the following page: http://localhost:8080/manager/html. Enter *admin* for username and password.

On the manager web application scroll down to the *Select WAR file to upload* field and press the *Browse* button. Navigate to target directory specified in the first step of the Scout Project Export wizard and select the *bahbah.war* file. Press Deploy. Repeat this step also for the *bahbah_web.war* file exported at the same location.



Then navigate to the following URL: http://localhost:8080/bahbah/. The BahBah Server is now running! From the web page you can download the selected rich client to start chatting or you can now also use the web version at http://localhost:8080/bahbah_web. This is also the URL to use for the mobile and tablet access.