Workshop 5: Designing a Real Time Web application with ICEFaces

Objective

In this workshop we will design a simple chat application. To accomplish this, we will need to use the following:

- The GlassFish application server.
- Java Server Faces (JSF).
- The ICEFaces component suite together with the ICEPush push server.
- Context and Dependency Injection (CDI).
- Bean validation.

Final date: See course page

1 Introduction

This workshop will take you a little deeper into the intricacies of Java Server Faces and teach you how you can properly utilize the provided functionality of a modern JSF component suite such as ICEFaces.

ICEFaces provides automatic Ajax functionality[1], direct-to-DOM rendering[2] and server push[3]. With these technologies, ICEFaces can partially update pages and only trigger submits on page components that it deems need updating; something called "Single Submit"[4] in the documentation of ICEFaces. In practice; this means that less navigation (and reloading of pages) is needed when updating components on a page. Compared to other component suites it is easier to get working Ajax behavior and requires very little effort from the programmer.

We will use the functionality of ICEFaces to create a single-paged chat without navigation or submit buttons (ICEFaces can submit component data without actual button clicks). We will also discuss how you should think when designing your application in order to get a clean design and preserve the MVC (Model-View-Controller) pattern in your web application.

2 Preparation

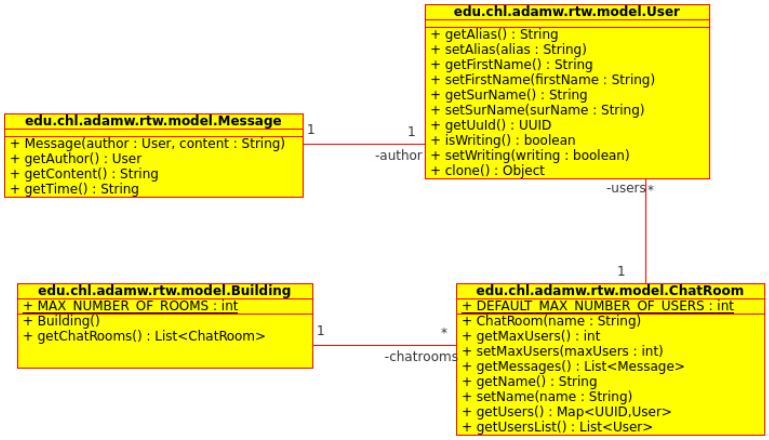
- Download the skeleton code from the course site and import it into NetBeans.

[1] Automatic Ajax, ICEFaces: http://www.icesoft.org/wiki/display/ICE/Automatic+Ajax
[2] D2D Rendering, ICEFaces: http://www.icesoft.org/wiki/display/ICE/Direct-to-DOM+Rendering
[3] Ajax Push Overview, ICEFaces: http://www.icesoft.org/wiki/display/ICE/Ajax+Push+-+Overview
[4] Single Submit, ICEFaces: http://www.icesoft.org/wiki/display/ICE/Single+Submit

- Inspect the skeleton code. Notice that the whole chat model has been provided for you. We have also provided you with an initial layout in the index.xhtml page.
- Your task will eventually be to implement all the managed beans and the JSF page (index.xhtml) holding the view description. If you look in the project view; suitable packages to place the Java classes in have already been provided for you.
- Read through this whole workshop *before* starting any actual work (especially the troubleshooting section at the end).

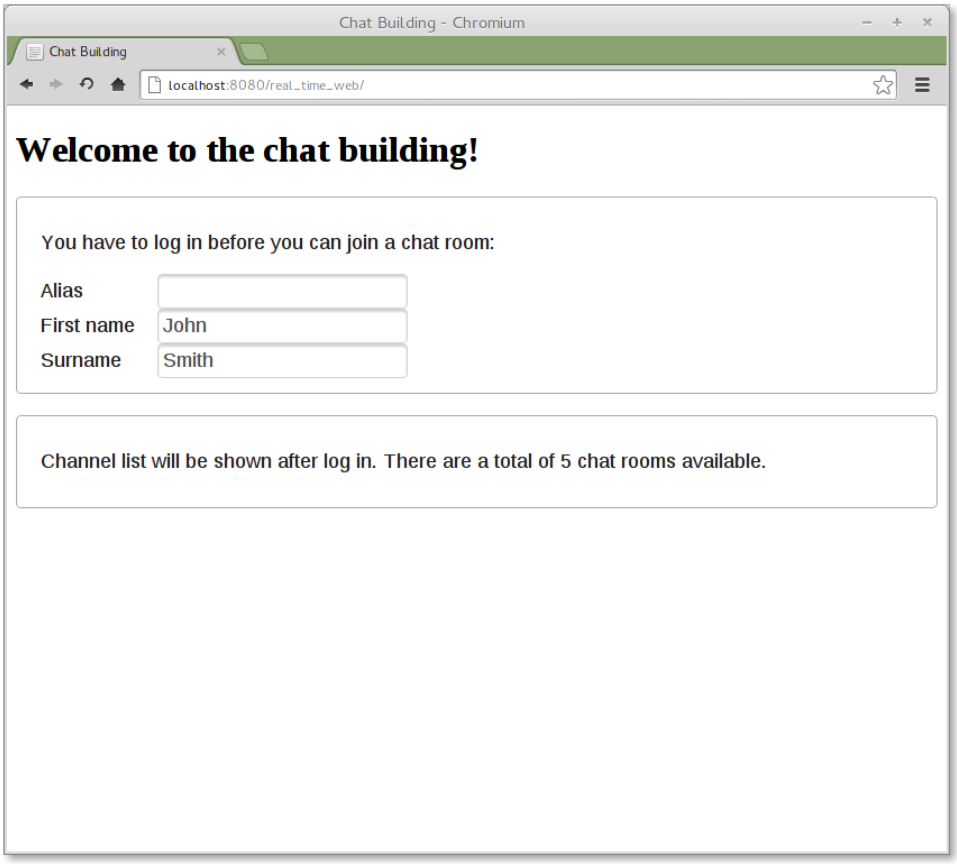
3 The chat model

While very simple and by no means complete, the chat model provided in the skeleton code consists of four separate classes. No changes should be needed to the actual model to implement the chat application.



- The Building
  - The main class and entry point of the application.
  - Each building has a number of chat rooms.
- Chat Rooms
  - Holds a list of users currently logged into the chat room.
  - Holds a list of messages entered by users of the chat room.
- Users
- Messages
  - Each message is associated with the user that posted it.

4 Creating a log in page



Lets create the ”log in” page for the chat. We want to create something similar to the picture above. You will need to do several steps for this:

1. Create a BuildingBean holding the building. Consider what scope it should be placed in and why. You should get the property *#{buildingBean.numRooms}* used inside index.xhtml to work.
2. Create a UserBean holding the user data. Consider what scope it should be placed in and why.
3. Add the following under **SECTION#1** in *index.xhtml*:
   a) Create form elements for entering alias, first name and surname. You can use the *<ace:textEntry>* component for this purpose.
   b) To accomplish form submission without a submit button you can (use the *<icecore:defaultAction>* component together with a *key=”Enter”* attribute.
4. Add validation and reject empty fields.
5. Does the section need a separate backing bean?

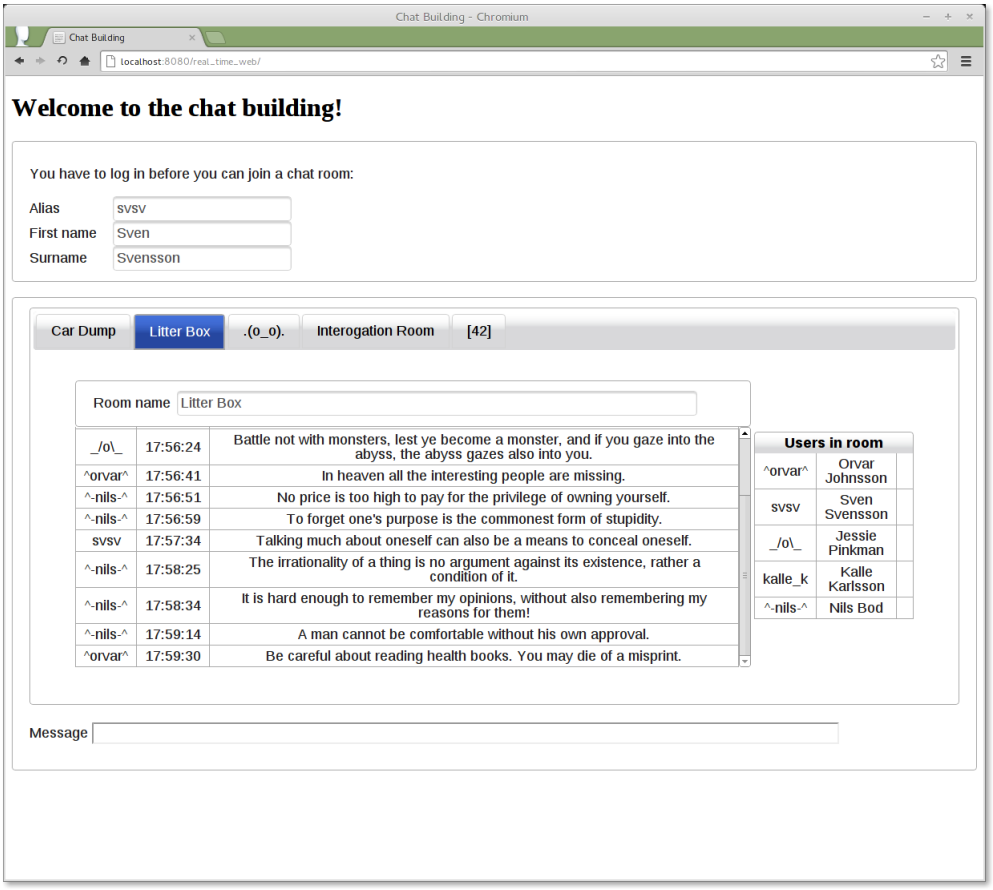
5 Mind how you toot those beans

When designing a JSF application it is important to structure your beans in a manner that preserves the MVC[5] pattern. This will help in creating an application with clean separation and coherent design. There are also many other considerations to take into account. As a thumb of rule, you should always try to follow these simple rules:

- Add at least one backing bean to each page. Often, pages have a single backing bean. However, multiple backing beans are allowed and is especially useful if a component, or set of components, are used in several pages. It can also be a valuable help when dividing the logic of the application.
- **Never** add actions, actionListeners or valueChangeListeners to a backing bean. Backing beans are part of the view and should not receive events from user input. Instead, create a special controller bean for this purpose.
- Keep the model separated from the beans. Try to resist the urge to add Java EE annotations directly into the model package. Instead, you should wrap model classes by extending them from a bean class. If you look in the code skeleton, there is a *model.bean* package intended for exactly this purpose.
- Always keep your beans in the shortest possible scope. Every time you put a bean in the wrong scope you are consuming unnecessary memory and breaking the application.
- *@ApplicationScoped* and *@Singleton* CDI beans are not thread-safe. If they have modifiable fields you have to make sure they are atomic. Otherwise your application will break when multiple instances are accessing it. There are many ways of solving this problem; one is to back up the model inside an EJB (which supplies thread safety with the right annotations), another is to handle the problem using the many synchronization options that Java itself supplies.

[5] MVC, Wikipedia: http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

6 Creating the chat view



Lets create the chat view. We want to create something similar to the picture above:

1. Add the following under **SECTION#2** in *index.xhtml*:
   a) Create a *<ace:tabSet>* that holds an empty *<ace:tabPane>* for each chat room defined. You can use the *<c:forEach>* tag to programmatically create multiple *<ace:tabPane>* entries from the list of rooms in the building.
   b) Inside each tab; add a *<ace:textEntry>* that displays the name of the room. Entering a new room name should also change the name in the tab. If done correctly, this should happen transparently without any effort on your part.
   c) Each tab should also hold two *<ace:dataTable>* components; one for the messages posted to the chat room and one for the list of users currently inside the chat room. A data table isn't the most optimal component for a chat view but it keeps the complexity down so we will use it anyway.
      i. The message entries in the left table should display the alias of the poster, time of the post and the message itself.

ii. The user entries in the right table should display the alias of the user, full name and an indicator symbol which is shown whenever that particular user is writing something in the bottom message field.

2. Make sure your beans are structured in a way that preserves MVC.

3. Add the following under **SECTION**#3 in *index.xhtml*:

   a) Create a *<h:inputText>* where the user can enter messages to the currently selected chat room. To get this to work, you need to be able to store the currently selected tab (or tab index) inside a backing bean.

   b) Again, to accomplish form submission without a submit button you can use the *<icecore:defaultAction>* component.

   c) To get the writing indicator symbol to show up in the user list of the chat room we need to update the *user.writing* property whenever a user is writing in the message field. This can be accomplished by adding the following components to the message field:

      i. An *<f:ajax>* component that listens to the *keyup* event and sets *user.writing* to *true*.

      ii. An *<f:ajax>* component that listens to the *blur* event and sets *user.writing* to *false*.

4. Make sure your beans are structured in a way that preserves MVC.

7 Ways of testing the chat application

Your application should now work just fine in single-user mode. It should even work with multiple users. However, users will need to manually reload to see each other's changes. To test the chat with multiple users you have two options:

1. Start two separate browsers (two different windows of the same will not work as that will use the same session).

2. Use Google Chrome and test the application with multiple users by using the user manager available in Chrome. The user manager can be found under *Menu* > *Settings* > *Users*. Create a few users and try it out.

To get the chat working without requiring separate reloads from users on changes we need to add push functionality.

8 Adding Ajax push

ICEFaces provides a simple way of managing push notifications between clients with it's PushRenderer interface. It works in the following manner:

- PushRenderer.addCurrentSession(GROUPNAME); registers the user's session to the specified group name.
- PushRenderer.removeCurrentSession(GROUPNAME); removes the user's session from the specified group name.
- PushRenderer.render(GROUPNAME); tell all members of GROUPNAME to update their view.

Using this, it is relatively simple to update the chat view for all users when new content arrives. You could, for example, try the following solution (these calls should all be done inside an actionListener, valueChangeListener or similar):

- Whenever a tab is switched, make the following calls:
  - *PushRenderer.removeCurrentSession("CHATROOM" + oldChatRoomIndex)*
  - *PushRenderer.addCurrentSession("CHATROOM" + newChatRoomIndex)*
  - *PushRenderer.render("CHATROOM" + oldChatRoomIndex)*
  - *PushRenderer.render("CHATROOM" + newChatRoomIndex)*
- Whenever a new message is entered, make the following calls:
  - *PushRenderer.render("CHATROOM" + chatRoomIndex)*
- Whenever a user logs in, make the following calls:
  - *PushRenderer.addCurrentSession("CHAT")*
  - *PushRenderer.addCurrentSession("CHATROOM" + chatRoomIndex)*
  - *PushRenderer.render("CHAT")*
- Whenever a user is writing something in the message field or stops, make the following calls:
  - *PushRenderer.render("CHATROOM" + chatRoomIndex)*
- Whenever the name of a chat room is changed, make the following calls:
  - *PushRenderer.render("CHAT")*

9 Clean up the graphical interface on session end

Add a session listener that cleans up the interface. Do you remember how to do this from the servlet workshop? Whenever a session expires the user should be removed from any chat room that he or she is part of. Ajax push should be used to notify every client of the change in the user list. How do you best accomplish this?

Troubleshooting

- Some of the beans of this workshop need to be put in session scope simply because CDI in Java EE 6 does not provide a view scope. This has been fixed in Java EE 7.
- Some validation errors might pop up in NetBeans while working with the *defaultAction* component (complaining about a missing action attribute). These can safely be ignored.
- Some versions of NetBeans will complain with *"CDI artifact is found but there is no beans.xml file."* on managed CDI beans. This is harmless and no longer relevant.
- When grouping components together on your page, use *<h:panelGroup>* and *<ace:panel>*.