Authentication and Authorization

Example File (Builder project Advanced Modeling/UI):

<your example path>\Advanced Modeling\UI\uml\uiAuthentication.xml
<your example path> \Advanced Modeling\UI\uml\uiLibSecurityServices.xml
<your example path> \Advanced Modeling\UI\uml\uiLibCommentServices.xml

On this Page:

- Authentication
- Authorization
- Advanced Controlling

Related Pages:

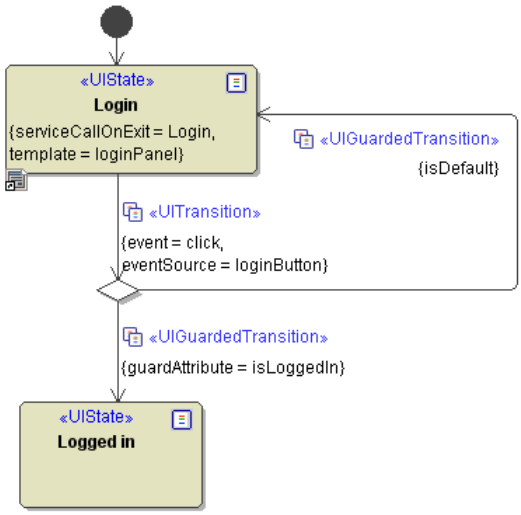
- Authentication and Authorization
- File Upload
- HTTPS
- History State
- Form and Form Validation
- Calling a UI from external Applications
- Usage of Choices
- Service Calls
- HTTP Proxy
- Controller States
- Back Button and Browser History
- Mock-Ups

Authentication

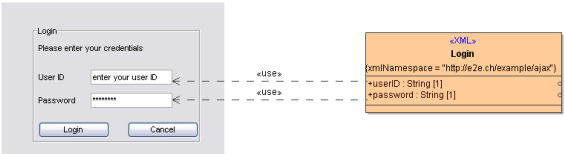
The xUML UI allows usage of existing authentication frameworks. In the presented example, a simple self-made authentication framework based on an SQLite database is used. To authenticate, the user types name and password into a form. Upon clicking the **Login** button, a service call is invoked. This service call verifies that the credentials exist and are correct.

The form, the binding and the behavior is defined within the usual diagrams:

- a UI state machine defining the login mechanism

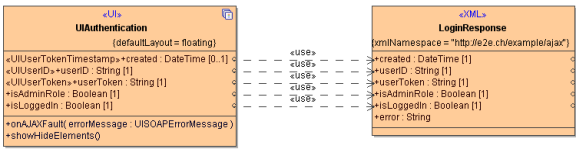


- a UI binding diagram

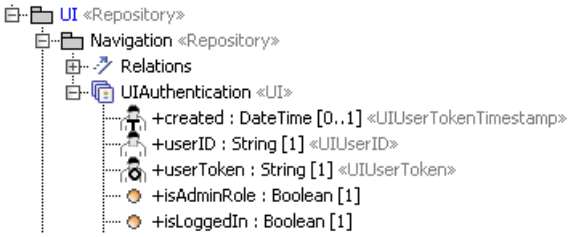


Transitions to a choice state are synchronous transitions by default, as the choice might depend on data delivered by preceding function calls. If there is no choice state, it might make sense to define the transition as synchronous by setting the tagged value **Is Asynchronous** to false.

In the above example, the choice whether the log in procedure was successful or not is based on the attribute **isLoggedIn**. The value for this attribute is defined in the **serviceCallOnExit = Login**, which uses User ID and Password as input parameter. The output parameter of the operation invoked at the service call is of type **LoginResponse**. As shown in the binding for the output parameter, the LoginRepsonse contains, besides others, this attribute **isLoggedIn**. In order to access this attribute for the choice within the state machine, the attribute needs to be stored as an attribute of the controller class. This is done by drawing further <<use>> dependencies, as shown in the figure below.



As the figure above shows, besides the described **isLoggedIn** attribute, there are further attributes bound to the controller. The first three of them are specially stereotyped, and are displayed using a distinct icon in the containment tree.



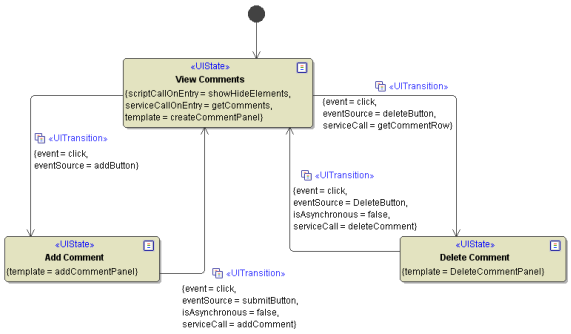
As the attribute **isLoggedIn,** the further controller attributes are populated after a successful authentication via the **Login** service call. The attributes stereotyped **<<UIUserTokenTimestamp>>**, **<<UI UserID>>** and **<<UIUserToken>>** will be used for authorization purposes. Details about authorization can be found in the following chapter Authorization.

The activity diagram defining the behaviour of the operation **Login** only assigns input and ouptut parameters, and catches potential exceptions. The concrete implementation of the Login mechanism, e. g. checking userID and password against a database, is done within a separate Builder model. This model is called uiLibSecurityServices and offers its functionality to the uiAuthentication example as an xUML library. This library represents the simple self-made authentication framework based on SQLite that was mentioned in the introduction. For more information regarding uiLibSecurityServices please also consider the chapter Authorization.

In case of a failed attempt to log in, the operation does not populate these attributes. The only attributes populated in the presented example would be the attribute **isLoggedIn** set to **false**, and the attribute **error** set to an individual text describing the error that occurred. In order to display this text to the user, the error String also has a binding to a UI element: The caption that was initially set to "Please enter your credentials" will display the error text in case of an unsuccessful log in attempt. The corresponding binding is not shown in the figure above but can be seen in the example.

Authorization

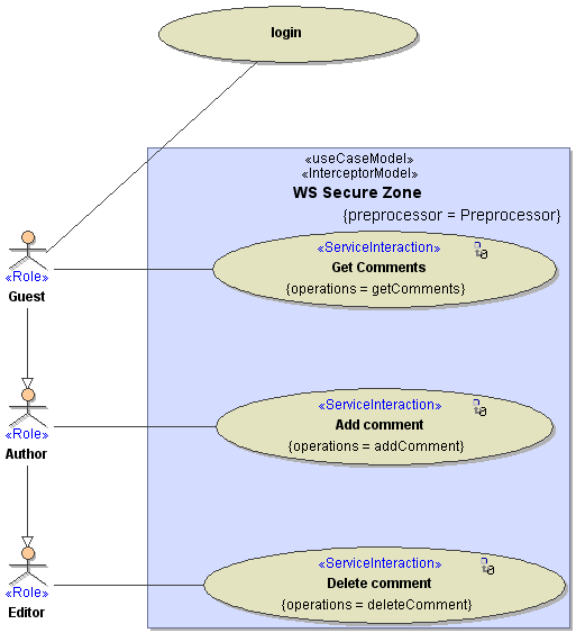
Authorization within xUML User Interfaces is based on the Security Model. The example represents a small web application showing comments posted by users. The figure below shows the UI state machine diagram defining the user interfaces and transitions for this web application.



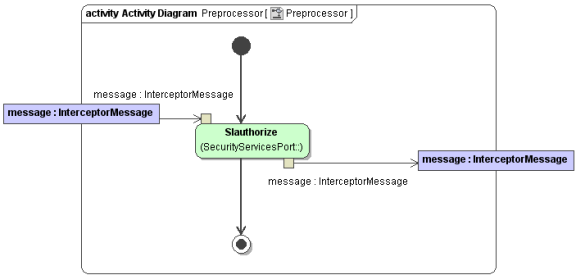
As illustrated in the state machine diagram above, users can view, add and delete comments. Navigation between the different states is implemented using buttons and corresponding click events. However, adding and deleting comments is reserved to certain user roles:

- All users need to authenticate themselves by userID and password, using the login functionality described in chapter Authentication.
- Depending on the user's role, there are further functions available:
    - Guests can only view comments
    - Authors can view and add comments
    - Editors can view, add, and delete comments.

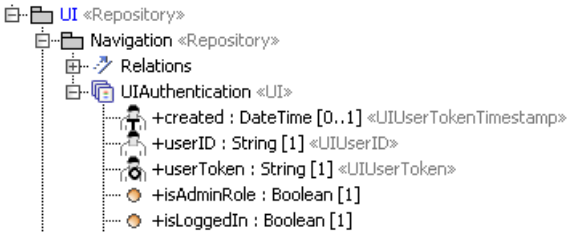
These rights and restrictions are defined within the following use case diagram. The operations used as **s erviceCall**s in the UI State Machine (**getComments**, **deleteComment**, **addComment**) are placed within a Secure Zone and assigned to user roles.



Following the interceptor pattern, whenever a user wants to access one of the operations within the Secure Zone, the preprocessor is called to verify his authorization. The figure below shows the activity diagram for the called preprocessor. As also described in the previous chapter about authentication, an operation from the uiLibSecurityServices library is invoked here again:



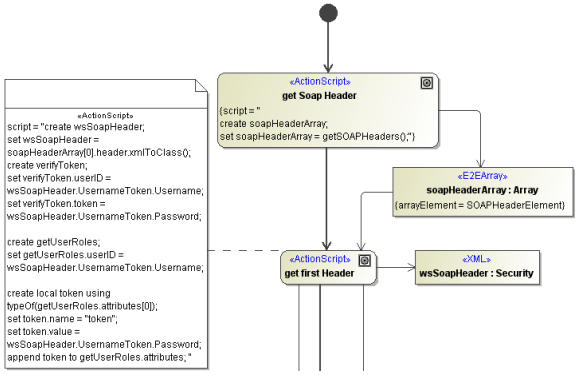
The **Slauthorize** operation verifies whether the user who is logged in is allowed to access the desired operation. Information about the logged in user originates from the UI state machine attributes that were initialized during the login procedure (see also chapter Authentication). The attributes **userID**, **userToken** and **created** are passed on to the preprocessor within SOAP headers. This corresponds to the WebServiceSecurity Standard wssecurity2004. In the figure below, the source of the information and an example for a resulting soap header is shown.



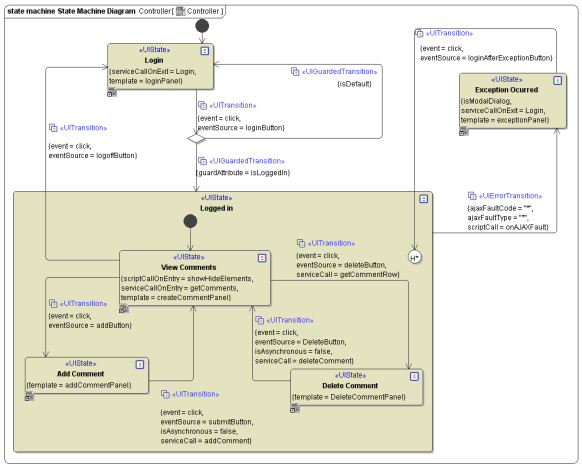

```
<soap:envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:header>
        <wsse:security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
        wsswssecurity-secext-1.0.xsd">
            <wsse:usernametoken>
                <wsse:username>Chris</wsse:username>
                <wsse:password>0000000cd58087b0000010e4000010f846984780</wsse:password>
                <wsu:created xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                wsswssecurity-utility-1.0.xsd">2010-07-15T09:47:37.02</wsu:created>
            </wsse:usernametoken>
        </wsse:security>
    </soap:header>
    <soap:body></soap:body>
</soap:envelope>
```

The naming of the WS Security header attributes does not completely correspond to the naming of the stereotyped UI state machine attributes. **<<UIUserID>>** corresponds to **<wsse:username>**; **<<UIUserToken>>** corresponds to **<wsse:password>** and **<<UIUserTokenTimestamp>>** corresponds to **<wsu:created>**.

Within the SIauthorize operation, first the SOAP header information is extracted, as shown in the figure below. Further on , the token is verified and roles assigned to the user are obtained. Please refer to the example file uiLibSecurityServices.xml for more details.



When a user tries to access an operation without permission - e.g. a Guest tries to delete a comment - the Security Service throws an exception. In the front end, this exception can be caught and shown to the user following the UI error handling mechanism described in chapter 8 Error Handling. In the presented example, an error message popup is shown to the user. Additionally the user has the option to provide other credentials that might suffice to execute the operation. When leaving the error popup, the user is transferred back to the UI state, where the exception had occurred, using the deep history state. The figure below shows the complete UI State Machine diagram, composed of the login procedure, the states to view, edit and delete comments, and the described error handling in the top right corner.



Due to a MagicDraw limitation, it is currently not possible to model a transition from a choice state to a history state.

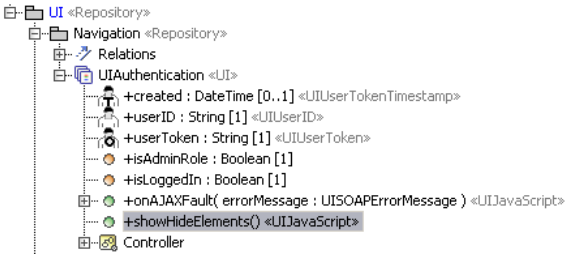
Advanced Controlling

Besides the authorization via the Security Interceptor model, in many cases it might make sense to adapt a user interface behavior or its contents depending on a user's role. This can be done in two ways:

Firstly it can be modeled in the state machine, using choice states that route different users to distinct UI states. The decisions within the guarded transitions can take any controller attribute into account.

Secondly it's also possible to adapt the template of a single user interface according to the logged in user. Such modifications of the appearance of a user interface can be done using JavaScript. In the presented example, a guest is not permitted to add comments. Besides the server side authorization that prevents him of adding comments in any case, the button on the user interface is hidden from the user if he is not an administrator. This is done by applying a script. The following figure shows the place where this script is called.

The script is contained in the operation **showHideElements** in the UI State Machine, and accesses the attribute **isAdminRole**. The operation is located in the UI Controller.



Based on the value of the controller attribute isAdminRole, the addButton is hidden.