Webertise Consulting

WBF Framework

Web Application Development for Open Text Delivery Server 11.0

2013-04-05

	EMAIL	TWITTER	WEB
Webertise Consulting GmbH, Warendorf	bernfried.howe@webertise.de	@BernfriedHowe	www.webertise.de

Webertise Consulting – WBF-Framework

Table of Contents

Webertise Consulting – WBF-Framework

Introduction

This document describes a Web Application Development Framework for the Open Text Web Site Management Delivery Server 11.0. It got the short name "WBF Framework" and development started end of 2012. The major objective of this project is to ease the development of highly integrated web applications for websites, intranets, and extranets using core Delivery Server functionality like XML (DynaMents) and XSL(T) for the presentation layer, Java (Open API and other Java APIs) as implementation language for the business layer, and the Delivery Server Content and User Repository as well as other data sources like relational databases as the data layer.

The WBF-Framework ensures a clear separation of responsibilities which means following tasks are handled by Delivery Server:

- Session handling is based on Delivery Server sessions
- User Authentication and Authorization uses Delivery Server User Repository and the Open API (User)
- Results are provided using XML Actions (DynaMents)
- HTML-Views are rendered using XSL(T) Stylesheets / Templates
- Properties are stored in Content Attributes of a Delivery Server Content
- Delivery Server functionality is used within the Java Actions using e.g. Open API DynaMent Requests to call a Target-DynaMent
- The implementation of individual business logic takes place in Java classes (called actions)

Typically you build individual functionality using Iolets, which provide the possibility to write own DynaMents. These Iolets are called using the Iolet-DynaMent within your content.

The main difference between Iolets and the WBF-Framework is, that Iolets are executed during the processing of the requested content. The WBF-Framework instead calls the requested action before any content is processed, which means, the result of the action can influence which content should be used to present the result. With Iolets you can only use redirect but then you have to take care that the request information is still available for the next request. Furthermore the WBF-Framework provides a clear procedure how to implement business logic and offers some standard functionality for:

- Request Handling
- Request Parameter Validation
- Forwarding (xchg, redirect, action)
- Response Creation (Result and Error Handling)

Installation

The WBF-Framework has been developed on an OT WSM Delivery Server 11.0 and the corresponding Open API, Tomcat 6, MySql 5.1 database, and runs on Cent OS 6.3 Server. Therefore a running Delivery Server 11.0 is required to install the WBF-Framework. Older versions of Delivery Server like v10.1 have not been tested but might work. It would require that the WBF Java Sources are compiled with the corresponding Delivery Server libraries.

Step 1: Create Eclipse Project

As Eclipse is a very common IDE for Java development we will use it as a basis for building the WBF classes. For development we used "Eclipse IDE for Java Developers – Indigo Service Release 2".

1. Download the directory “wbf-framework/dsframework” with all contents and sub-directories from GIT to a local directory
2. Create a new Java Project in your Eclipse Workspace
3. Copy to the content of the “wbf-framework/dsframework/src” directory to the “src” directory of your new Eclipse project.
4. Add all libraries of the “wbf-framework/dsframework/lib” directory to the Java Build Path (Project Properties/Java Build Path/Libraries/Add external Jars).
5. Refresh your Eclipse Project => There should be no errors and warnings in the “Problems” window.

Now you have an Eclipse project which builds you all Java Classes you need for deployment.

Step 2: Deploy WBF classes and Libraries

The Delivery Server needs some configuration in order to run the WBF-Framework. Classes and additional libraries need to be copied to the Delivery Server Web Application path and Hot Deployment needs to be configured.

1. Create xlets-wbf directory within the Delivery Server directory. It’s assumed that “cps” is the name of the Delivery Server Web Application under Tomcat. “xlets-wbf” directory should be created here: “<tomcat-dir>/webapps/cps/WEB-INF/xlets-wbf”.
2. Copy the build classes of the wbf-framework project in Eclipse (usually build into “bin” directory) to the “xlets-wbf” directory. You should see the directory “de/webertise/…” with some more sub-directories in the “xlets-wbf” directory.
3. Copy the library “freemarker.jar” from the “wbf-framework/dsframework/lib” directory to the “<tomcat-dir>/webapps/cps/WEB-INF/lib” directory.
4. Restart tomcat to make all changes available.
5. Log-in to Delivery Server and create a Hot Deployment
   a. Go to “Server Manager – Java-based Additions – Administer Hot Deployment Configuration”
   b. Click “New” to create a new configuration
   c. Set Name to “xlets-wbf” and Description to “Hot Deployment for WBF-Framework”
   d. Activate the hot deployment configuration
   e. Click in left tree on “Class Paths”, create a new item and set the value to “[#system:webappdir#].concat("/WEB-INF/xlets-wbf").toFileUri()”
   f. Click in the left tree on “Weblet Definitions”, create a new item and set following values:

            i. Alias: wbf
            ii. Full class name: de.webertise.wbf.weblet.MasterWeblet
            iii. Active: checked
    g. Commit all changes with “OK”.
    h. Open the Hot Deployment Configuration again and check, whether the 2 values in the “Hash Codes” column are identical. If not, click “deploy” link and if one line disappears the refresh button at the bottom. Now both hash codes should be identical and the weblet “wbf” and all WBF-Framework classes should work.

Step 3: Configure Delivery Server

If you want to run the wbf Weblet as default weblet, which is possible and would remove the “/wbf/” from all urls, change the System configuration as follows:

1. Go to “Server Manager – Configuration – Administer System Configuration”
2. Click in the left tree navigation on System
3. Change the System Parameter with key “reddot.idea.masterservlet.defaultweblet (Default Weblet)” to value “wbf”.
4. Commit your changes with “OK”.
5. Restart Tomcat to make this change available.

Furthermore some settings concerning the Session ID Cookie need to be done:

1. Go to “Server Manager – Configuration – Administer User Configuration”
2. Activate option “Write Delivery Server session ID as cookie”.
3. Set “Name of Session ID cookie” to “wbfSessionID”
4. Set “Path to Session ID cookie” to “/”
5. Commit all changes with “OK”

For the Demo Project “wbfcommunity” and the “wbfadmin” project the language “de” and “en” are required. Create these two languages as follows (if not exists already):

1. Go to “Server Manager – Configuration – Administer Project Configuration”
2. Click in the left tree navigation on “Languages” and check if “de” and “en” are available”.
3. Create missing languages with following values:
    a. for “de”: Name=German, Language (RFC 4646)=de, Language (ISO 639)=Deutsch (de [ger,deu]), Region (ISO 3166): leer, variant=leer
    b. for “de”: Name=English, Language (RFC 4646)=en, Language (ISO 639)=English (en [eng]), Region (ISO 3166): leer, variant=leer
4. Commit all changes with “OK”

Step 4: Import wbfadmin Project

The Delivery Server project “wbfadmin” is required as it holds all settings using content attributes of specific contents. An export of this project can be found in the git repository under “wbf-framework/dsframework/exports/wbfadmin.zip”. Import this file with following steps:

1. Copy the wbfadmin.zip file into the directory "<tomcat-dir>/webapps/cps/WEB-INF/share/projects/wbfadmin".
2. Create a new project called "wbfadmin" in Delivery Server.
3. Import the wbfadmin.zip file using "Project – Administer Project – Import" from within the wbfadmin project.

The "wbfadmin" project should be available now with all required content to start the admin dialog.

Step 5: Create Administrator and Demo User

The Administration dialogs require an authorization of an user, who owns certain groups and roles. Execute the following steps in order to create such an administration user and to prepare the authorization structure for the demo project:

1. Create three new user group called "wbfAdministrators", "wbfGuests", and "wbfMembers" under "Users – Administer".
2. Create a user group under wbfMembers called "wbfCommunity".
3. Create a new user called "wbfadmin" within the "wbfAdministrators" user group.
    a. Set the properties like password, name, etc like you want but activate the flag "Use project settings (ignore following cookie settings)".
    b. Assign new roles "wbfAdministrator" and "wbfAuthenticated" to this user.
    c. Commit all changes with "OK"
4. Assign the user group "wbfGuests" to the "anonymous" user.
5. Create a new demo user for the wbfcommunity project called "demouser" under user group "wbfMembers / wbfCommunity" with following settings:
    a. Set the properties like password, name, etc like you want but activate the flag "Use project settings (ignore following cookie settings)".
    b. Assign new roles "wbfCommunityUser" to this user.
    c. Commit all changes with "OK"

Step 6: Import wbfcommunity Demo Project

The Delivery Server project "wbfcommunity" is a demo project which uses the WBF-Framework. An export of this project can be found in the git repository under "wbf-framework/dsframework/exports/wbfcommunity.zip". Import this file with following steps:

1. Copy the wbfcommunity.zip file into the directory "<tomcat-dir>/webapps/cps/WEB-INF/share/projects/wbfcommunity".
2. Create a new project called "wbfcommunity" in Delivery Server.
3. Import the wbfcommunity.zip file using "Project – Administer Project – Import" from within the wbfcommunity project.

The "wbfcommunity" project should be available now with all required content to start the admin dialog.

Step 7: Test the installation

After all these steps have been done the WBF-Administration dialogs and the Community Project should work now. You can test this with the following URLs (the URL might differ dependent on your Delivery Server installation e.g. ".../cps/rde/wbf..."):

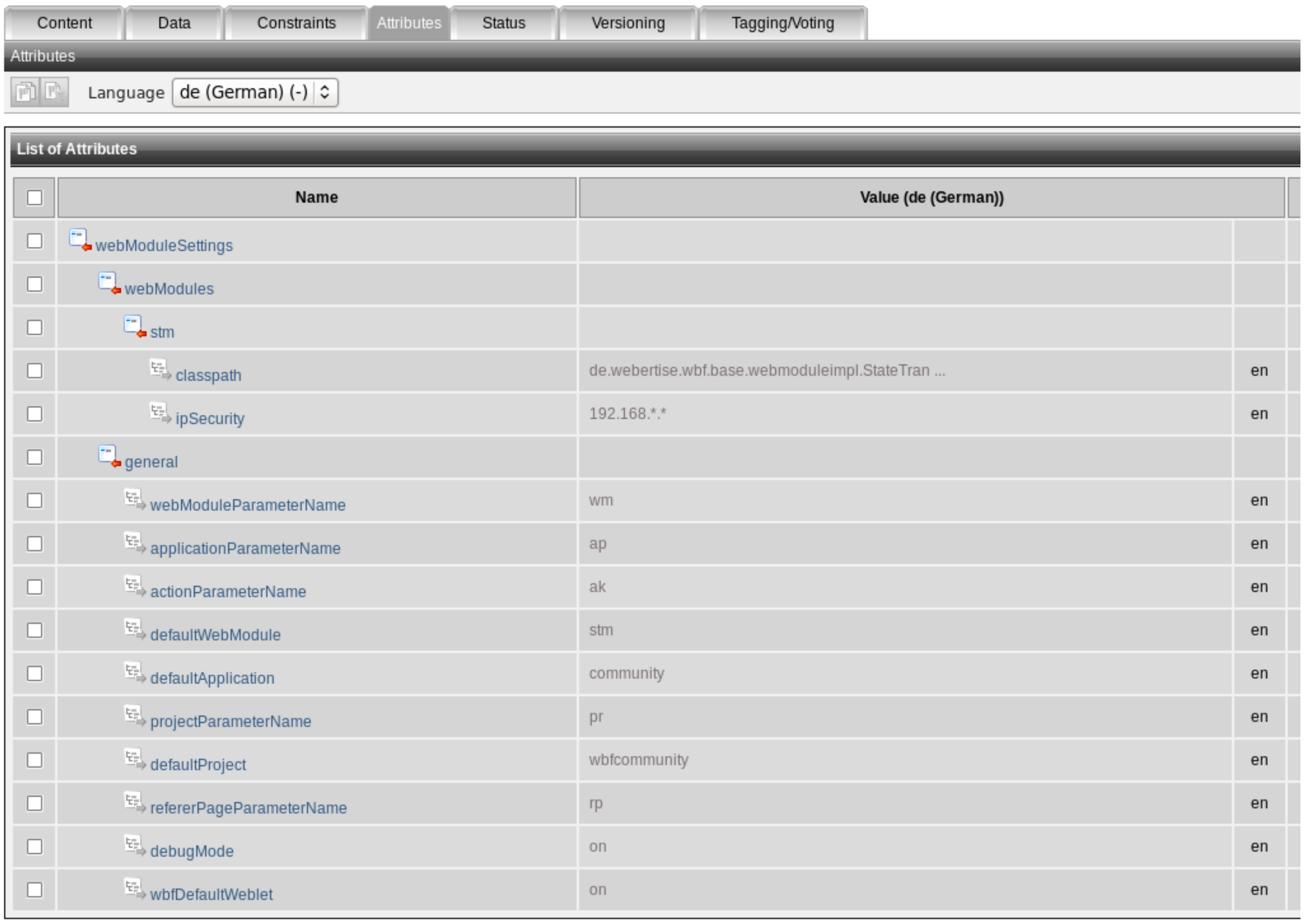
- WBF-Admin: http://<host>:<port>/cps/wbfadmin (Click "Home" and login with "wbfadmin" user)
- WBF-Community: http://<host>:<port>/cps/wbfcommunity (Click "Login" and login with "demouser" user)

Configuration

The configuration of the WBF-Framework basically takes place in two Delivery Server contents of the “wbfadmin” project based on content attributes.

Web Module Configuration

The “Web Module” is an abstraction layer which should give the framework the flexibility to implement different types of web application patterns. For now there is exactly one implementation of a web module which follows a “state-transition” model and is called “stm”. All settings for the “stm” module are maintained in the Delivery Server content “WebModuleSettings” of the “wbfadmin” project.



Language de (German) (-)

List of Attributes

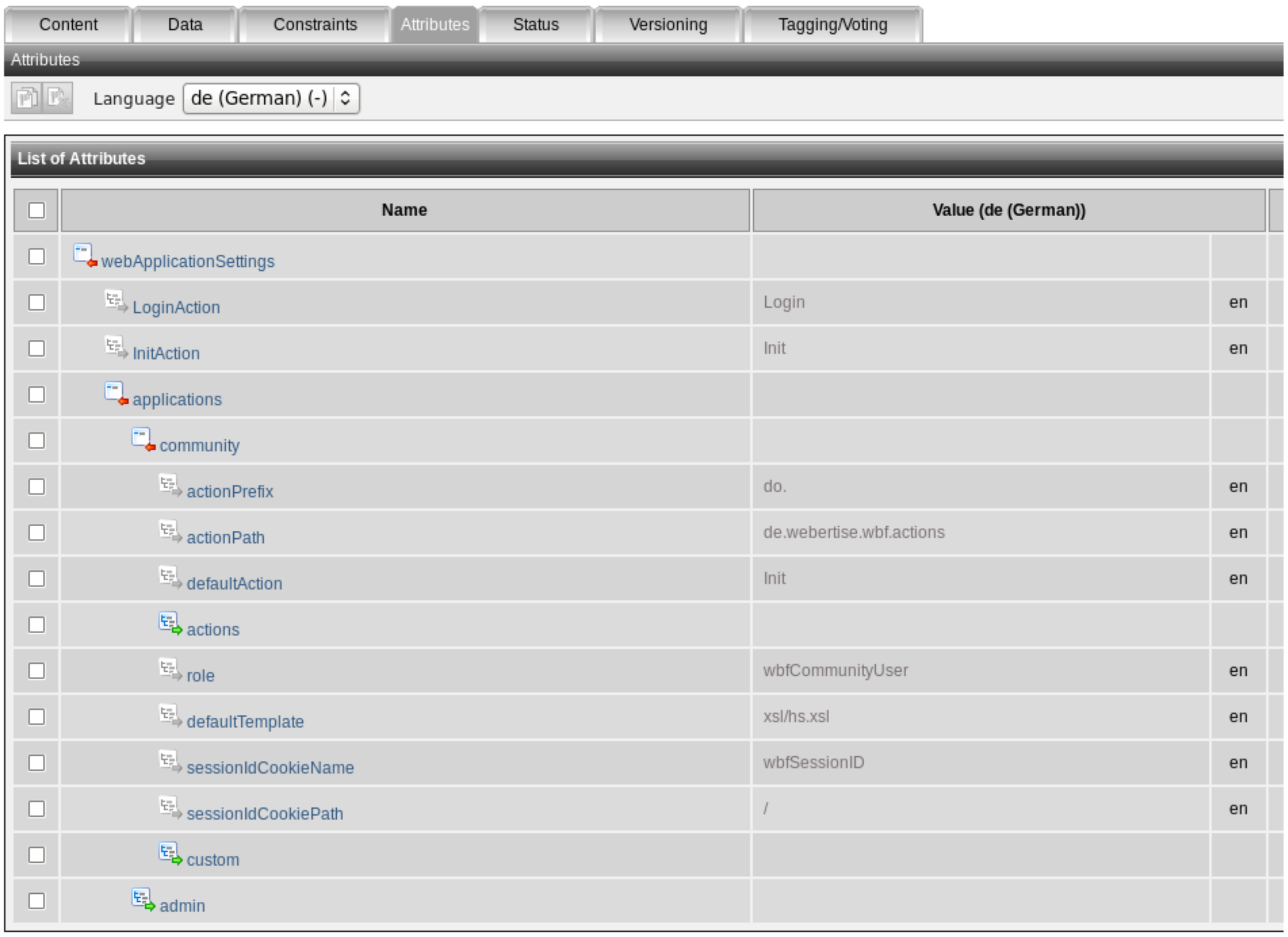
Name	Value (de (German))	
webModuleSettings		
webModules		
stm		
classpath	de.webertise.wbf.base.webmoduleimpl.StateTran ...	en
ipSecurity	192.168.*.*	en
general		
webModuleParameterName	wm	en
applicationParameterName	ap	en
actionParameterName	ak	en
defaultWebModule	stm	en
defaultApplication	community	en
projectParameterName	pr	en
defaultProject	wbfcommunity	en
refererPageParameterName	rp	en
debugMode	on	en
wbfDefaultWeblet	on	en

The following screenshot shows the list of settings, which all have to be defined in the “en” language variant:

- **webModulesSettings.webModules.stm.classpath:** Defines the Java classpath to the implementation of the “stm” web module. This value should not be changed.
- **webModulesSettings.webModules.stm.ipSecurity:** Defines a range of allowed IP addresses for the WBF-Framework. The notation supports wildcards (e.g. 192.168.*.*) and ranges (e.g. 192.168.1.10-20). You can add multiple values to this content attribute.
- **webModulesSettings.general.webModuleParameterName:** Defines the name of the URL request parameter which holds the name of the web module. Default: wm
- **webModulesSettings.general.applicationParameterName:** Defines the name of the URL request parameter which holds the name of the application. Default: ap
- **webModulesSettings.general.actionParameterName :** Defines the name of the URL request parameter which holds the name of the action parameter. Default: ak
- **webModulesSettings.general.projectParameterName :** Defines the name of the URL request parameter which holds the name of the project parameter. Default: pr
- **webModulesSettings.general.refererPageParameterName :** Defines the name of the URL request parameter which holds the name of the content to which should be forwarded the response to. Default: rp
- **webModulesSettings.general.defaultWebModule :** Defines the name of the default web module which is used if no web module name could be found within the URL. Default: stm
- **webModulesSettings.general.defaultApplication :** Defines the name of the default application which is used if no web module name could be found within the URL. Default: community
- **webModulesSettings.general.defaultApplication :** Defines the name of the default Delivery Server project which is used if no project name could be found within the URL. Default: wbfcommunity
- **webModulesSettings.general.debugMode :** Defines the debug mode. Allowed values are “on” => debugging activated and “off” => debugging deactivated.
- **webModulesSettings.general.wbfDefaultWeblet :** Defines whether the wbf weblet is defined as the default weblet of the Delivery Server instance (on/off).

Web Application Settings

Each “Web Application” which has been developed based on the WBF-Framework requires some settings to work. Theses settings are store beside global web application settings in the Delivery Server content “WebApplicationSettings” in the root directory of the “wbfadmin” project.



Name	Value (de (German))	
webApplicationSettings		
LoginAction	Login	en
InitAction	Init	en
applications		
community		
actionPrefix	do.	en
actionPath	de.webertise.wbf.actions	en
defaultAction	Init	en
actions		
role	wbfCommunityUser	en
defaultTemplate	xsl/hs.xsl	en
sessionIdCookieName	wbfSessionID	en
sessionIdCookiePath	/	en
custom		
admin		

General Settings

- **webApplicationSettings.LoginAction:** defines the name of the action which represents the “Show Login Dialog” functionality of the web application. The name of this action needs to be identical for all applications defined.
- **webApplicationSettings.InitAction:** defines the name of the action which represents the “Show Start Page” functionality of the web application. The name of this action needs to be identical for all applications defined.

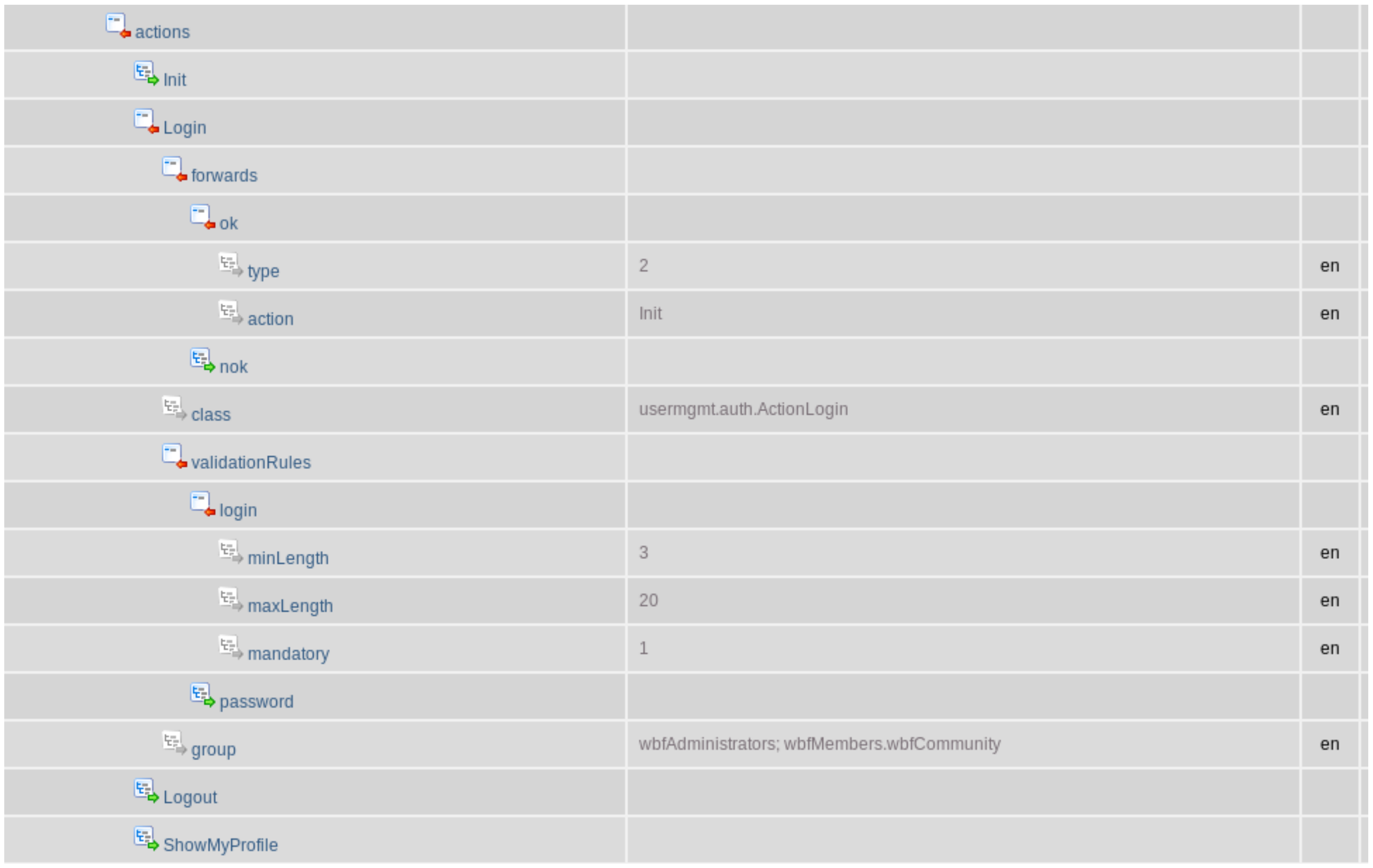
Web Application specific Settings

- **webApplicationSettings.applications.<application-name>:** Defines the application with all settings as child attributes.

- **webApplicationSettings.applications.<application-name>.actionPrefix:** Defines the prefix for an URL parameter name (default: “do.”), which helps to identify the name of the called action, e.g. do.login would call the “login” action. This feature is required for html forms, which have multiple buttons with different actions. Each button gets the name=”do.<action-name>”.
- **webApplicationSettings.applications.<application-name>.actionPath:** Defines the class path to the actions of the given application.
- **webApplicationSettings.applications.<application-name>.defaultAction:** Defines the name of the default action if no action name could be found in the request. Default: Init
- **webApplicationSettings.applications.<application-name>.defaultTemplate:** Defines the default XLS(T) Template which is used to render the action responses. Default: xsl/hs.xsl
- **webApplicationSettings.applications.<application-name>.role:** Defines the user roles which are allowed to use this application.
- **webApplicationSettings.applications.<application-name>.sessionIdCookieName:** Defines the name of the cookie, which is used to identify the Delivery Server Session ID.
- **webApplicationSettings.applications.<application-name>.sessionIdCookiePath:** Defines the path of the cookie, which holds the Delivery Server Session ID.
- **webApplicationSettings.applications.<application-name>.custom.<parameter-name>:** Defines custom settings for an application like an email address or name of the smpt-connector. All these settings will be read on startup and provided as list in the ApplicationSettings object.

Web Application Action Settings

Each application has an content attribute “webApplicationSettings.applications.<application-name>.actions” where all actions are defined as child elements.



actions		
Init		
Login		
forwards		
ok		
type	2	en
action	Init	en
nok		
class	usermgmt.auth.ActionLogin	en
validationRules		
login		
minLength	3	en
maxLength	20	en
mandatory	1	en
password		
group	wbfAdministrators; wbfMembers.wbfCommunity	en
Logout		
ShowMyProfile		

General

- **webApplicationSettings.applications.<application-name>.actions.<action-name>:** Defines the name of an action. No value is required as the name of the content attribute is used.
- **…actions.<action-name>.class:** Defines the class name including sub-packages which provides the implementation of the action. The final class path of an action is put together based on the value of “webApplicationSettings.applications.<application-name>.actionPath” + “/” + value of “…actions.<action-name>.class”.
- **…actions.<action-name>.group:** Defines the user groups allowed to execute this action. Multiple values are supported.

Action Forwards

- **…actions.<action-name>.forwards.<forward-name>:** Defines the name of an action forward. Value is not required for this content attribute.

- **…actions.<action-name>.forwards.<forward-name>.type:** Defines the type of the action forward. Following types are allowed:
  - 1: the xchg weblet is called using the *view* and *template* settings of this action forward item
  - 2: a redirect to another action is called based on the *action* setting of the action forward item
  - 3: a redirect to another page is called based using the *view* and *template* settings of this action forward item
  - 4: the xchg weblet is called using the page given in the refererPage Url Parameter.
  - 5: the action is called based on the *action* set in the action forward settings. The called action may not have an action forward of type 5 in order to avoid recursion.
- **…actions.<action-name>.forwards.<forward-name>.action:** Defines the name of the action to be forwarded or redirected to. Only required for action forward types 2 and 5.
- **…actions.<action-name>.forwards.<forward-name>.view:** Defines the name of the html page which includes the result of the action response. E.g. html/login.html. Only relevant for type 1,3, and 4.
- **…actions.<action-name>.forwards.<forward-name>.template:** Defines the name of the xsl content which renders the result of the action response. E.g. xsl/hs.xsl. Only relevant for type 1,3, and 4.

Validation Rules

- **…actions.<action-name>.validationRules.<parameter-name>:** Defines the name of a request parameter which should be validated for this action.
- **…actions.<action-name>.validationRules.<parameter-name>.minLength:** Defines the minimal length for the parameter. Should be a value between 1-9999.
- **…actions.<action-name>.validationRules.<parameter-name>.maxLength:** Defines the maximum length for the parameter. Should be a value between 1-9999.
- **…actions.<action-name>.validationRules.<parameter-name>.mandatory:** Defines whether this parameter is mandatory or not. (1=mandatory; 2=not mandatory)
- **…actions.<action-name>.validationRules.<parameter-name>.pattern:** Defines a pattern for a regular expression which is used for validation.

Getting started

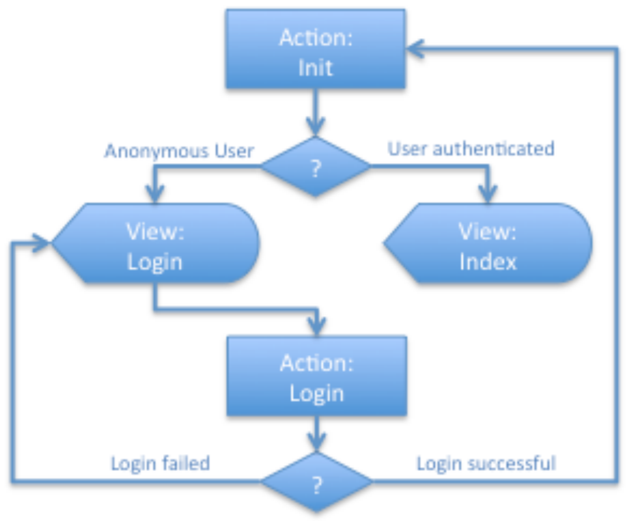
After the WBF-Framework has been installed and configured correctly we will explain how to develop a simple action step by step. Let’s start with the classic login dialog and process. What are the requirements?

- Need a dialog with username and password as input fields and a button to send the login request.
- Validation of the input values against validation rules
- Validation against existing data of the Delivery Server User Repository
- Execute the action (here: login to Delivery Server)
- Return the result (error or success) on the appropriate page (login dialog for errors and index page for success)

This scenario already covers all features of the WBF-Framework, which will now be explained in detail. We use the wbfcommunity project as basis. You will find all configurations and files within that project.

Step 1: Configuration

We assume that we have already an application defined in the “WebApplicationSettings” and all general settings have been made. The best way to understand what needs to be configured is to draw a State Transition Diagram for the login process.



There are 2 actions and 2 views required for a basic login process. The “Init” action is the default action and will be called if nothing else was specified. Dependent on the User (anonymous or authenticated) the Login Page or the Index Pages is delivered. The “Login” action will be forwarded to two different pages. An successful login will forward to the action “Init” which will lead to the view “Index”. If the login fails the login view will be shown.

Configure the actions

First we will define the “Init” action in the “WebApplicationSettings” content of the “wbfadmin” project. The action will be defined as child content attribute of “webApplicationSettings.applications.community.actions” with name “Init”. Set following child attributes for this new content attribute:

```
…actions.Init
        class: ActionInit
        forwards
                startPage
                        type: 1
                        view: html/index.html
                        template: xsl/hs.xsl
                loginPage
                        type: 1
                        view: html/login.html
                        template: xsl/hs.xsl
```

Validation rules are not required as no request parameters are sent to this action. The second action is the “Login” action configured as follows:

```
…actions.Login
        class: usermgmt.auth.ActionLogin
        group: wbfAdministrators; wbfMembers.wbfCommunity
        validationRules
                login
                        minLength: 2
                        maxLength: 20
                        mandatory: 1
                password
                        minLength: 5
                        maxLength: 20
                        mandatory: 1
        forwards
                ok
                        type: 2
                        action: Init
                nok
                        type: 1
                        view: html/login.html
                        template: xsl/hs.xsl
```

For the “Login” Action two input parameters (Request Parameters) are required. Therefore we define validation rules concerning min/max Length and mandatory. The forward “ok” makes a redirect to the “Init” action, the forward “nok” returns back to the Login page using a forward (xchg).

If we finally compare the configuration result with the State Transition Diagram, we see, that each action requires an action configuration and each arrow from an action represents a forward definition of an action.

Step 2: Implement the Actions

Now it’s time to build the business logic which always has it’s base in an action class. The classes, we describe here can be found in the Java package “de.webertise.wbf.actions”. Each action needs a execute(…) and validate(…) method implemented with specific code for the action.

Init Action

The Init Action basically checks the authorization status of the session user and decides which page should be shown.

```
package de.webertise.wbf.actions;
import de.reddot.api.common.session.CoaSession;
import de.reddot.api.web.io.WebletRequest;
import de.webertise.wbf.base.action.ActionResponseItem;
import de.webertise.wbf.weblet.MasterWeblet;

public class ActionInit extends de.webertise.wbf.base.action.AbstractAction {

   public ActionInit() {}

   public boolean execute(CoaSession session, WebletRequest request) {
      String login = session.getCoaUser().getLogin();
      if (login.equals("anonymous")) {
         this.setActionForwardName("loginPage");
      } else {
         this.setActionForwardName("startPage");
       }
      return true;
   }

   public boolean validate(CoaSession session, WebletRequest request) {
      return true;
   }
}
```

This action is very simple as it just reads the session user object and checks the login name. If the name equals “anonymous” the current user is not authenticated yet. Dependent on this result the action forward name of the action is set to “loginPage” or “startPage”. Both names are defined forwards of the action “Init”. The validate method just returns true as nothing needs to be validated.

Login Action

The “Login” Action needs a bit more functionality, as form values need to be read and validated.

```
package de.webertise.wbf.actions.usermgmt.auth;
import java.util.Enumeration;
import de.reddot.api.ObjectNotFoundException;
import de.reddot.api.common.session.CoaSession;
import de.reddot.api.common.user.CoaUser;
import de.reddot.api.common.user.CoaUserController;
import de.reddot.api.common.user.CoaUserGroup;
import de.reddot.api.web.io.WebletRequest;
import de.webertise.wbf.actions.ActionConstants;
import de.webertise.wbf.base.action.ActionResponseItem;
import de.webertise.wbf.weblet.MasterWeblet;

public class ActionLogin extends de.webertise.wbf.base.action.AbstractAction {

   public ActionLogin() {}

   public boolean execute(CoaSession session, WebletRequest request) {

     String login = (String) this.getRequestParameter("login");
     String password = (String) this.getRequestParameter("password");
     String applName = request.getParameter("wbf.requestctrldata.application");

     CoaUser user = null;
     if (login != null && !login.equals("")) {
        try {
           user = CoaUserController.selectUser(login);
        } catch (ObjectNotFoundException e) {
           log.debug(LOG_CATEGORY, "ActionLogin - execute: ObjectNotFoundException for
              Login = '" + login + "'");
        }
     }

     if (user != null) {
        String roles = "";
        @SuppressWarnings("unchecked")
        Enumeration<String> roleEnum = user.getRolesEnumeration();
        while(roleEnum.hasMoreElements()) {
           String role = roleEnum.nextElement();
           roles = roles + "[" + role + "]";
        }
        if (!MasterWeblet.applicationSetting.getApplication(applName).checkUserRoles(roles)) {
           this.setActionForwardName("nok");
           this.setErrorCode(ActionConstants.ACTION_CUSTOM_VALIDATION_ERROR_11);
        } else {
           int result = session.assignUser(login, password);
           if (result == CoaSession.USERAUTH_OK) {
              String userName = session.getCoaUser().getName();
              this.setResponseParameter(ActionResponseItem.TARGET_SESSION,
                    "userName", userName, false, false);
              this.setErrorCode(Integer.toString(ACTION_EXECUTE_RESULT_OK));
              this.setActionForwardName("ok");
           } else {
              this.setActionForwardName("nok");
              this.setErrorCode(ActionConstants.ACTION_CUSTOM_VALIDATION_ERROR_10);
        }
      }
      this.setResponseParameter(ActionResponseItem.TARGET_SESSION_TRANSIENT,
           "login", login, false, false);
    } else {
      // an error occured: lets check 'result'
      this.setActionForwardName("nok");
      this.setErrorCode(ActionConstants.ACTION_CUSTOM_VALIDATION_ERROR_10);
```

```
    }
    return true;
  }

  public boolean validate(CoaSession session, WebletRequest request) {

   if (!validateAllRequestParameters(session, request)) {
      this.removeRequestParameter("password");
      this.setResponseParameter(ActionResponseItem.TARGET_SESSION_TRANSIENT, "login", (String),
          this.getRequestParameter("login"), false, false);
      this.setErrorCode(Integer.toString(ACTION_VALIDATION_RESULT_NOK));
      super.setActionForwardName("nok");
      return false;
    } else {
      this.setErrorCode(Integer.toString(ACTION_VALIDATION_RESULT_OK));
      return true;
    }
  }
}
```

The "Login" action requires the validate method, as form input values need to be validated. The action provides a standard method "validateAllRequestParameters", which iterates through the list of defined validationRules and reads all corresponding request parameters, validates them and returns error codes. It's also standard that all request parameter are automatically transferred to the action response. If certain values like the password should not be considered, you can remove it using the removeResponseParameter(…) method.

If the validation failes, the execute method does not need to be called. This will be reached by returning a false. If that is the case, the validate method needs to take care of the response concerning, which forward should be used (=> setActionForwardName("name")) and what response parameter should be set. The method setResponseParameter(…) is used to set specific values for the action response including the scope. This will be described in more detail later. An general error code should be set with setErrorCode(Integer.toString(ACTION_VALIDATION_RESULT_NOK)).

If the validate() method returns true, the execute() method is called. With getRequestParameter("name") all values of validated request parameters (defined in the action) can be read. The action implements all individual business logic and returns the result using the setResponseParameter(…) and all other methods for errors as mentioned in the validate() method description.

Of course there are more details to know but for the "Getting Started" part it should be enough.

Step 3: Create the View

The presentation of the result is done using Delivery Server core functionality. For actions like the “Login” action, which requires a view you will find following contents in Delivery Server:

HTML Page

The HTML page covers the page html code which is required to render the page. With an Include DynaMent the result of an action is included into the html page:

```
<rde-dm:include     content="modules/modAuth/actions/show.xml"
                    stylesheet="modules/modAuth/templates/show.xsl"/>
```

You can see that an action requires an XML content, which reads the relevant action response XML and an XSL content to render the XML response into final html code.

Action XML

The action xml is responsible for the provision of all data from action response but also from other sources for the XSL(T) template. For the “Login” action following xml is used:

```
<show>
  <!-- text resources -->
  <rde-dm:include content="modules/modAuth/resources/resources.xml"/>
  <!-- error code of the action -->
  <rde-dm:attribute attribute="session:transient.stm.admin.Init.last.action_errors"
                    mode="read" op="xml" tag="notag"/>
  <!-- form data from last request -->
  <rde-dm:attribute mode="read" attribute="session:transient.stm.admin.Init.last.action_request"
                    op="xml" tag="notag"/>
</show>
```

The first Include-DynaMent includes the text resources from separate xml content. The two Attribute-DynaMents read XML from the action response. We use the op=”xml” as the response writes xml as Strings to reserved session attributes (all attributes starting with session:transient will be deleted before the next requests is executed.

This xml content can be found in the “wbfadmin” project in content “modules/modAuth/actions/show.xml

Action XSL(T) Template

The xsl(t) template renders the xml provided by the Action XML. In this case the text resources, all information about the errors and the input values (except password) can be found and rendered using standard xsl(t) / xpath functionality. The sample code can be found in the “wbfadmin” project in content “modules/modAuth/templates/show.xsl”.

WBF-Framework in Detail

tbd