INTERACTIVE FORMS
USER PROFILES LOGIC
DEVELOPER GUIDE

XPERIENCENTRAL

Date: November 25, 2013
Target Audience: Software Developer
Document intended for XperienCentral version: 10.2 and higher
Document ID and version: GXD0129_en, version 1.0

SUMMARY

Interactive Forms User Profiles logic makes it possible for you to get User Profile forms up-and-running with minimal effort. When using User Profiles within a website, Forms needed to login, signup etc are needed to make this work within IAF.

This document is mainly aimed at the Developer: it explains how to extend the default profile with extra data and expose that via IAF

The latest version of this document is available at:
https://connect.gxsoftware.com/WCM/Documentation.htm

The Javadoc for the XperienCentral API is available at:
https://connect.gxsoftware.com/javadoc/XperienCentral9/.

PREREQUISITES

To use the Interactive Forms, the programmer should have the following:

- Experience with Java programming language.
- Experience with creating WCBs (XperienCentral Component Bundles).

The programmer should also have completed the XperienCentral basic and advanced training modules for editors.

VERSIONING

Version	Date	Description
1.0	November 25, 2013	Initial version

CONTENTS

1. INTRODUCTION

This section introduces the interactive forms user profiles logic both from a configuration and a developer point of view.

1.1 Installation

The user profiles logic is implemented in one WCB that needs to be deployed. Apart from that, the following manual operations need to be executed:

- Import the IAF forms for webusers by importing the export.zip file
- Create a personalization expression:
    - Name: WebUsers orgurl
    - Tag name: wm-orgurl
    - Type: xsl
    - XSLT:<xsl:value-of select="/root/system/requestparameters/parameter[name='orgurl']/value" />
- Create the following pages and place a IAF form element on it with the respective form selected
    - Login (assign as special page of type 'Login'
    - Other pages for change profile, logout, forgot password, reset password etc
    - Several special pages to which is redirected after a confirm link has been received depending on the type of confirm and the correctness of the link. The values listed below are the keys to the special pages. These can be created using the Web Initiative Configuration Panel on the Special Pages tab.
        - confirm_registration_successful
        - confirm_registration_failed
        - confirm_registration_too_old
        - confirm_deregistration_successful
        - confirm_deregistration_failed
        - confirm_deregistration_too_old
        - confirm_resetpassword_successful
        - confirm_ resetpassword _failed
        - confirm_ resetpassword _too_old

1.2 Simple Configuration

The user profiles form logic WCB has a set of features which can be used depending on the requirements of a certain site. These features can be configured from within the edit environment of IAF, by either settings specific properties to a handler, or adding/removing handlers in certain forms.

2. FUNCTIONAL OVERVIEW

2.1 Register

Being able to register yourself with the data stored in the default profile of a webuser. By default the user is routed to a second step that displays a confirmation text.

Confirmation mail
By default a confirmation mail is sent to the email address entered during the registration. The subject- and body text of this email can be manipulated. The following placeholders in these texts will be replaced by actual data:

- #site-url#
- #confirmation-link#

It's currently not possible to send confirmation mail in different languages.

2.2 Confirm subscription

The newly registered user will receive a confirmation mail containing a confirmation link. Clicking or pasting this link will result in confirming the account encrypted in the link. The generated confirmation link is valid for a number of days that can be defined wihin the configuration management setting
'wmpuserprofilesformlogic_set.confirmation_code_valid_for_number_of_days'.

2.3 Autoconfirm subscription

When a webuser doesn't have to go through the confirmation mail cycle, the AutoConfirm handler can be used to confirm the account in the registration flow. Don't forget to remove the 'Send Confirmation Mail'- handler and to update the text on the completion step in the 'Register' form.

2.4 Log in

At login, the user credentials are verified against those stored within the default profile. If another user was logged in, this user is first logged out. Furthermore at login, that user may not have exceeded the maximum number of incorrect logins. When all these conditions have been validated, the user profile is loaded.

2.5 Maximum number of faulty logins

It's possible to make sure a user can only have a certain number of faulty logins. This is activated within the Login hander configuration. There you can also define how many faulty logins are allowed. When the maximum number is reached, the user has to go to the 'Forgot Password' functionality that will reset the count after a new password has been generated.

2.6 Delete account

The registered user that wants to sign out will receive a confirmation mail containing a confirmation link. Clicking or pasting this link will result in deleting the complete user profile for that user.

3. ARCHITECTURAL OVERVIEW

3.1 Basic setup

The basic setup of the interactive forms user profiles forms is to make use of a central service (WebUserService) which provides for elementary logic for each handler. A handlers' responsibility is to get fragment values, call upon the WebUserService and transfer fine grained exceptions to ContainerErrors.

3.2 Abstract User Profile API

Within WebUserService, the User Profile API is used and only there where needed exposed to the developer.

3.3 HttpSession

Any user profile data that needs to exist in the HttpSession for personalization purposes is only there for personalization purposes. This means that:

- A form logic component may not put user profile data in the HttpSession directly.
- A form logic component may not get the value of an attribute (presumably XML) and parse that XML to get user profile data from it

All user profile data present in the HttpSession is managed by the WebUserService.

3.4 User Profile Extension

Any extra user profile data that's not already present in the DefaultProfile already needed to be managed via a custom User Profile Extension. With the introduction of Interactive Forms, there also needs to be form logic to use this Extended Profile data within both forms and the personalization engine. Depending on the complexity of the Profile Extension data the largest part of this logic shall – or better said; can - be generically handled.

4. DEALING WITH PROFILE EXTENSIONS

Typically, the data managed via a user profile extension is also needed for personalization purposes. If that's the case then this data also needs to be made available to the personalization engine after a specific webuser has logic in. Depending on the complexity of the data managed within the profile extension, a developer needs to develop form logic or not to make the data available to the personalization engine.

4.1 What is a profile extension

A profile extension manages specific data associated with a User. XperienCentral provides via User Profile Management:

- User Profile Manager
- Default Profile
- Profile Extension interfaces

So, if you want to store the eye color and length of a user which is not naturally not already done via the DefaultProfile, you would need to create your own ProfileExtension like for instance your BiometricProfile:

```
interface BiometricProfile {

        static enum EyeColor {
         BLUE, BROWN, GREY, UNKNOWN
        }

        int getLength();

        void setLength(int length);

        EyeColor getEyeColour();

        void setEyeColour(EyeColor eyeColor);
}
```

You then would implement the rest of the profile extension as defined in …(doc userprofile)

4.2 Generic or not?

Whenever webuser logs in, the login logic loads the webusers' DefaultProfile and also all profile extensions available within the installation. This could include the BiometricProfile. Such a profile extension is processed through reflection and will end up generating the following xml for the personalization engine:

```
<biometric>
	<eyecolor><![CDATA[BLUE]]></eyecolor>
	<length>185</length>
</biometric>
```

Let's now assume that our BiometricProfile would be changed to look like this:

```
interface BiometricProfile {

	static enum EyeColor {
	 BLUE, BROWN, GRAY, UNKNOWN
	}

	int getLength();

	void setLength(int length);

	EyeColor getEyeColour();

	void setEyeColour(EyeColor eyeColor);

	String getFingerprintHash();

	void setFingerprintHash(String fingerprintHash);
}
```

We've now added the fingerprint hash data to the BiometricProfile. It's highly unlikely that you'd want to expose this data to the personalization engine from a security point of view. If you do nothing the personalization engine will 'see' this:

```
<userprofile-biometric>
        <eyecolor><![CDATA[BLUE]]></eyecolor>
        <length>185</length>
        <fingerprinthash><![CDATA[GSHHJa&K8*6$SxJHJ@S^KHHSsa24]]></fingerprint
hash>
</userprofile-biometric>
```

In order to prevent this from happening, or if the default generated xml doesn't match your purpose, you need to exclude your profile extension from generic handling and do part of the work yourself.

4.3 Non generic handling

When you've chooses to go for non generic handling of your profile extension, you need to take the following steps:

- Tell the login code to exclude your profile extension from generic handling.
- Write form logic (a handler) to create your own xml representation that needs to be exposed to the personalization engine.
- Add this custom handler to the login form.

4.3.1 Exclude profile extension from generic handling

Go to /web/setup and look for this entry:
wmpuserprofilesformlogic_set. exclude_profile_provider_from_generic_processing

Add the name of the class of ProfileProvides to this list as a new value. In our example this would be
'BiometricProfileProvider'.

4.3.2 Write form logic to create own xml representation

You need to create a handler form logic component that basically gets the custom profile, calls upon it's getters for data to form xml which is the offered to the personalization engine. Here the example code for such a handler.

```
public class ExposeBiometricProfileHandler extends ComponentBase implements
FormLogicProviderComponent, FormLogicProviderService {

    private WebUserService myWebUserService;
    private SessionManager mySessionManager;

    public static String UNABLE_TO_RETRIEVE_PROFILE = "UNABLE_TO_RETRIEVE_PROFILE";
    public static String NO_LOGGED_IN_USER = "NO_LOGGED_IN_USER";
    private static final Logger LOG =
Logger.getLogger(ExposeBiometricProfileHandler.class.getName());

    public RoutingResult run(FormScope scope, Map<String, Object> parameters, Map<String,
Object> languageLabels) {

        Session activeSession = mySessionManager.getActiveSession();
        HttpServletRequest httpServletRequest =
activeSession.getContext().getHttpServletRequest();
        Website website = activeSession.getContext().getWebsite();

        User activeUser = null;
        try {
            activeUser = myWebUserService.getActiveUser(httpServletRequest, website);
        } catch (WebUserException e) {
            LOG.log(Level.SEVERE, "Can't get active user ", e);
            scope.addContainerError("ERROR", (String) languageLabels.get("ERROR"));
        }

        if (activeUser == null || "".equals(activeUser)) {
            LOG.log(Level.SEVERE, "Can't get active user we're about to expose with new
biometric profile data");
            scope.addContainerError("ERROR", (String) languageLabels.get("ERROR"));
        } else {
            BiometricProfileProvider biometricProvider =
                    (BiometricProfileProvider)
myWebUserService.getProfileExtensionProvider(website,
                            BiometricProfileProvider.class);
            BiometricProfile biometricProfile = null;
            try {
```

```
                biometricProfile = (BiometricProfile)
biometricProvider.getProfileFor(activeUser);
            } catch (UserManagementException e) {
                LOG.log(Level.SEVERE, "Can't get BiometricProfile", e);
                scope.addContainerError(UNABLE_TO_RETRIEVE_PROFILE, (String)
languageLabels
                        .get(UNABLE_TO_RETRIEVE_PROFILE));
            }

            // Get profile extension data needed for personalization engine
            EyeColor eyeColor = biometricProfile.getEyeColor();
            int length = biometricProfile.getLength();

            // Create profile extension xml for personalization engine
            String xml = "<eyecolor><![CDATA[" + eyeColor.toString() + "]]></eyecolor>/n";
            if (length>0) {
                xml += "<length>" + length + "</length>/n";
            } else {
                xml += "<length/>"/n";
            }

            // Update the personalization representation for the biometric profile
            myWebUserService.exposeProfileExtensionRepresentationToPersonalizationEngine(
                httpServletRequest, BiometricProfileProviderImpl.class, xml);
            } catch (NoLoggedInUserException e) {
                scope.addContainerError(NO_LOGGED_IN_USER, (String)
        languageLabels.get(NO_LOGGED_IN_USER));
                LOG.log(Level.SEVERE, "Can't expose biometric Profile since user isn´t
logged in", e);
            } catch (UnableToRetrieveProfileException e) {
                scope.addContainerError(UNABLE_TO_RETRIEVE_PROFILE, (String)
languageLabels
                        .get(UNABLE_TO_RETRIEVE_PROFILE));
                LOG.log(Level.SEVERE, "Can't expose biometric Profile since profile can't
be retrieved", e);
            }
        }
        return null;
    }
}
```

4.3.3 Add custom handler to the login form

The ExposeBioMetricProfileHandler you just created or of course you equivalent implementation needs to added to the places where the DefaultProfile is also exposed to the personalization engine. The most obvious place is of course the Login, but please don't forget the 'update profile' form or any other custom form in which you update data within you custom profile extension. Make sure your handler is places behind the LoginHandler.

4.4 Updating your profile

When you developed your profile extension, you provided an API to get and set your profile data. What you haven't done yet is provide logic to update your profile from within a form. For that purpose you need to provide handlers to get and save your profile data to and from your form:

- A PreHandler that loads your profile extension data and puts it in the form scope to be picked up by form field you put on your form
- A Handler that saves your profile extension data after the form is submitted and also exposes the new profile data to the personalization engine.

4.4.1 Load profile data to form scope

A Prehandler is a normal handler that's configured to be used as prehandler so here we go with an example for our BiometricProfile:

```
public class LoadBiometricProfilePreHandler extends ComponentBase implements
    FormLogicProviderComponent, FormLogicProviderService {

    /** The Constant NO_LOGGED_IN_USER. */
    public static final String NO_LOGGED_IN_USER = "NO_LOGGED_IN_USER";

    /** The Constant UNABLE_TO_RETRIEVE_PROFILE. */
    public static final String UNABLE_TO_RETRIEVE_PROFILE = "UNABLE_TO_RETRIEVE_PROFILE";

    /** The Constant LOG. */
    private static final Logger LOG =
        Logger.getLogger(LoadBiometricProfilePreHandler.class.getName());

    /** The my session manager. */
    private SessionManager mySessionManager;

    /** The my web user service. */
    private WebUserService myWebUserService;

    /*
     * (non-Javadoc)
     * @see nl.gx.forms.wmpformapi.wcb.formlogicprovidertype.FormLogicProviderService
     * #run(nl.gx.forms.wmpformapi.engine.FormScope, java.util.Map, java.util.Map)
```

```
     */
    /**
     * {@inheritDoc}
     */
    public RoutingResult run(FormScope scope, Map<String, Object> parameters,
        Map<String, Object> languageLabels) {

        Session activeSession = mySessionManager.getActiveSession();
        HttpServletRequest httpServletRequest =
activeSession.getContext().getHttpServletRequest();
        Website website = activeSession.getContext().getWebsite();

        User user;
        try {
            // 1. Get the logged in user
            user = myWebUserService.getActiveUser(httpServletRequest, website);
            // 2. Get the specific profile for this user
            BiometricProfileProvider provider =
                (BiometricProfileProvider)
myWebUserService.getProfileExtensionProvider(website,
                 BiometricProfileProvider.class);
            BiometricProfile profile = provider.getProfileFor(user);

            // 3. Add the profile data to the form scope
            scope.setFragmentValue(WCBConstants.BIOMETRICPROFILE_PARAM_LENGTH,
                profile.getLength());
            scope.setFragmentValue(WCBConstants.BIOMETRICPROFILE_PARAM_EYECOLOR,
                profile.getEyeColor());
            // Assuming the fingerprint hash doesn't need to be shown on the form..
        } catch (UserManagementException e) {
            scope.addContainerError(UNABLE_TO_RETRIEVE_PROFILE,
                (String) languageLabels.get(UNABLE_TO_RETRIEVE_PROFILE));
            LOG.log(Level.SEVERE, "Biometric profile can't be retrieved", e);
        } catch (NoLoggedInUserException e) {
            // A pre-handler runs multiple times, sometimes without the user logged in.
            scope.addContainerError(NO_LOGGED_IN_USER,
               (String) languageLabels.get(NO_LOGGED_IN_USER));
            LOG.log(Level.FINEST, "Biometric profile can't be retrieved."
                    + " The user is not logged in for this session (pre-handler dummy
session?)");
        }
        return null;
    }
}
```