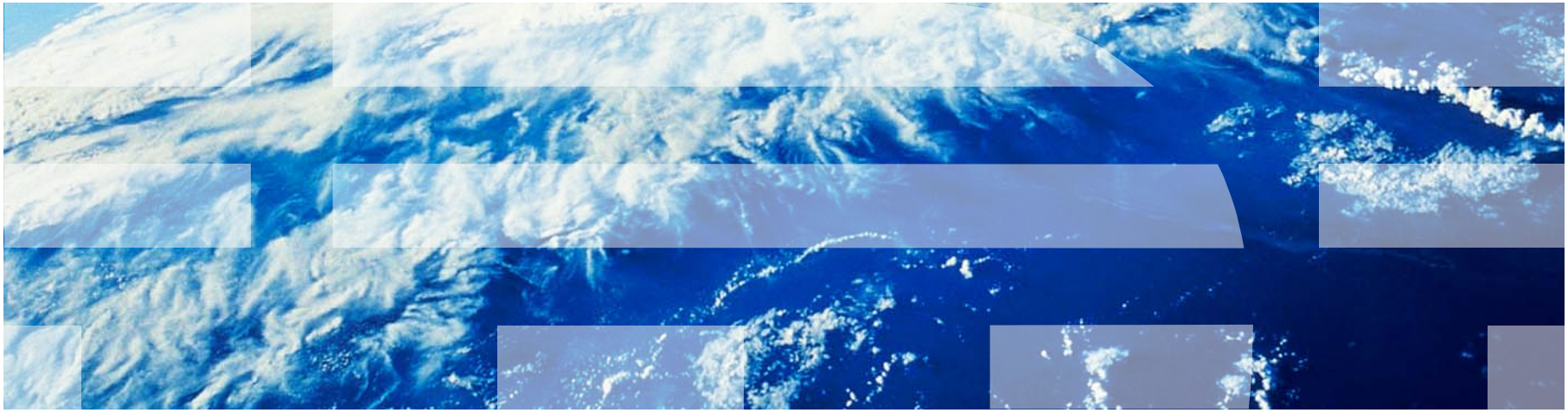
IBM Worklight V6.1.0
Getting Started

Custom authenticator and login module



Trademarks

- IBM, the IBM logo, and ibm.com, are trademarks or registered trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Worklight is a trademark or registered trademark of Worklight, an IBM Company. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.
- Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.
- Other company products or service names may be trademarks or service marks of others.
- This document may not be reproduced in whole or in part without the prior written permission of IBM.

About IBM®

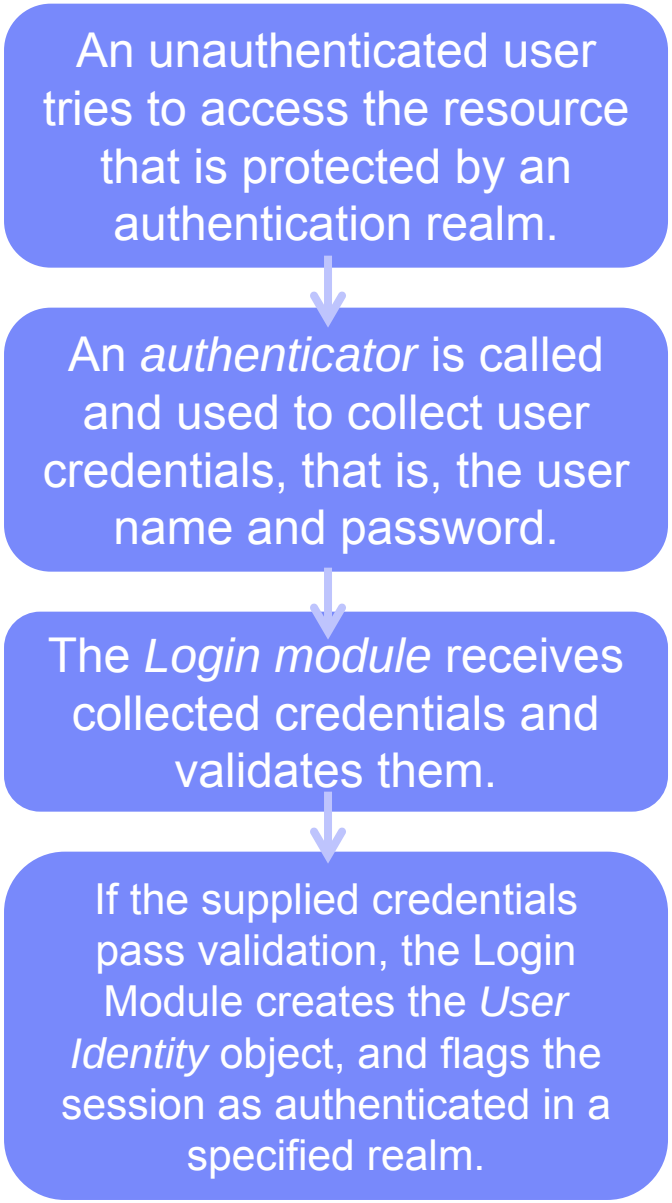
- See http://www.ibm.com/ibm/us/en/

Agenda

- Authentication introduction
- Configuring authenticationConfig.xml
- Creating a custom Java authenticator
- Creating a custom Java login module
- Creating client-side authentication components
- Examining the result

Authentication introduction (1 of 3)

- The authentication process can be interactive (for example, user name and password) or non-interactive (for example, header-based authentication).
- It can involve a single step (for example, a simple user name/password form) or multiple steps (for example, it might have to add a challenge after it issued the first password).
- The definition of the authentication realm includes the class name of an authenticator and a reference to a login module.
- An authenticator is an entity that collects user information.
  - For example: a login form
- A login module is a server entity that validates the retrieved user credentials and builds the user identity.
- You configure authentication settings such as realms, authenticators, and login modules, in the `authenticationConfig.xml` file that is on the Worklight Server.



Authentication introduction (2 of 3)

- The authenticator, login module, and user identity instances are stored in a session scope, therefore they exist while the session is alive.
- You can write custom login modules and authenticators when the default ones do not match your requirements.
- In previous modules:
  - You implemented a form-based authentication and used a non-validating login module.
  - You implemented an adapter-based authentication without having to add login modules, and ran credentials validation manually.
- In some cases, when the credential validation cannot be ran on the adapter level and requires more complex code, an extra login module can be implemented.
  - For example: when an enterprise-custom credentials validation is required; or when more information must be retrieved from each client request, such as cookie, header, and user-agent.

Authentication Introduction (3 of 3)

- This module explains how to create a custom authenticator and a login module:
  - You learn how to implement a custom authenticator that collects the user name and password by using a request to a predefined URL.
  - You learn how to implement a custom login module that checks credentials that are received from the authenticator.
  - You learn how to define a realm that uses your custom authenticator and login module.
  - You learn how to use this realm to protect resources.
- For more information about Worklight® authentication concepts, see the IBM Worklight user documentation.

Agenda

- Authentication introduction
- Configuring authenticationConfig.xml
- Creating a custom Java authenticator
- Creating a custom Java Login Module
- Creating client-side authentication components
- Examining the result

Configuring authenticationConfig.xml (1 of 2)

- Add authentication information to the **authenticationConfig.xml** file.
- In the realms section, define a realm that is called **CustomAuthenticatorRealm**.
  - Make sure that it uses **CustomLoginModule**.
- Specify **MyCustomAuthenticator** as the className. You implement it in later slides.

```
<realm name="CustomAuthenticatorRealm" loginModule="CustomLoginModule">
    <className>com.mypackage.MyCustomAuthenticator</className>
</realm>
```

- In the loginModules section, add a **loginModule** called **CustomLoginModule**.

```
<loginModule name="CustomLoginModule">
    <className>com.mypackage.MyCustomLoginModule</className>
</loginModule>
```

- Specify **MyCustomLoginModule** as the className. You implement it in later slides.

Configuring authenticationConfig.xml (2 of 2)

- Add a security test to the <securityTests> section of the **authenticationConfig.xml** file.
- You will use this security test later to protect the adapter procedure, so make it a <customSecurityTest>

```
<securityTests>
    <customSecurityTest name="CustomAuthSecurityTest">
        <test isInternalUserID="true" realm="CustomAuthenticatorRealm"/>
    </customSecurityTest>
</securityTests>
```

- Remember the security test name, to use in following slides

Agenda

- Authentication introduction
- Configuring authenticationConfig.xml
- Creating a custom Java authenticator
- Creating a custom Java login module
- Creating client-side authentication components
- Examining the result

Creating a custom Java™ authenticator (1 of 21)

- The Authenticator API is:
  - `void init(Map<String, String> options)`
  - `AuthenticationResult processRequest(HttpServletRequest request, HttpServletResponse response, boolean isAccessToProtectedResource)`
  - `AuthenticationResult processAuthenticationFailure(HttpServletRequest request, HttpServletResponse respo`
  - `AuthenticationResult processRequestAlreadyAuth request, HttpServletRespo`
  - `Map<String, Object> getAu`
  - `HttpServletRequest getReq request, HttpServletRespo userIdentity, LoginExtens`
  - `Boolean changeResponseOnS request, HttpServletResponse response)`
  - `WorkLightAuthenticator clone()`

The `init()` method of the authenticator is called when the Authenticator instance is created. It receives the options that are specified in the realm definition in the **authenticationConfig.xml**.

Creating a custom Java authenticator (2 of 21)

- The Authenticator API is:
  - `void init(Map<String, String> options)`
  - `AuthenticationResult processRequest(HttpServletRequest request, HttpServletResponse response, boolean isAccessToProtectedResource)`
  - `AuthenticationResult processAuthenticationFailure(HttpServletRequest request, HttpServletResponse respo`
  - `AuthenticationResult processRequestAlreadyAuth request, HttpServletRespo`
  - `Map<String, Object> getAu`
  - `HttpServletRequest getReq request, HttpServletRespo userIdentity, LoginExtens`
  - `Boolean changeResponseOnS request, HttpServletResponse response)`
  - `WorkLightAuthenticator clone()`

The `processRequest()` method is called for each request from an unauthenticated session.

Creating a custom Java authenticator (3 of 21)

- The Authenticator API is:
  - `void **init**(Map<String, String> options)`
  - `AuthenticationResult **processRequest**(HttpServletRequest request, HttpServletResponse response, boolean isAccessToProtectedResource)`
  - `AuthenticationResult **processAuthenticationFailure**(HttpServletRequest request, HttpServletResponse response, String errorMessage)`
  - `AuthenticationResult **processRequestAlreadyAuth**(HttpServletRequest request, HttpServletResp`
  - `Map<String, Object> **getAu**`
  - `HttpServletRequest **getRe**(HttpServletRequest request, HttpServletResp, userIdentity, LoginExtens`
  - `Boolean **changeResponseOns**(HttpServletRequest request, HttpServletResp`
  - `WorkLightAuthenticator **cl**`

The `processAuthenticationFailure()` method is called if the login module returns a credentials validation failure.

Creating a custom Java authenticator (4 of 21)

- The Authenticator API is:
  - `void **init**(Map<String, String> options)`
  - `AuthenticationResult **processRequest**(HttpServletRequest request, HttpServletResponse response, boolean isAccessToProtectedResource)`
  - `AuthenticationResult **processAuthenticationFailure**(HttpServletRequest request, HttpServletResponse response, String errorMessage)`
  - `AuthenticationResult **processRequestAlreadyAuthenticated**(HttpServletRequest request, HttpServletResponse response)`
  - `Map<String, Object> **ge**`
  - `HttpServletRequest **get** request, HttpServletRe userIdentity, LoginExt`
  - `Boolean **changeResponse** request, HttpServletRe`
  - `WorkLightAuthenticator`

The `processRequestAlreadyAuthenticated()` method is called for each request from an already authenticated session.

Creating a custom Java authenticator (5 of 21)

- The Authenticator API is:
  - `void init(Map<String, Stri`
  - `AuthenticationResult proce`
    `request, HttpServletRespon`
    `isAccessToProtectedResourc`
  - `AuthenticationResult`
    `processAuthenticationFailu`
    `HttpServletResponse respon`
  - `AuthenticationResult`
    `processRequestAlreadyAuthe`
    `request, HttpServletRespons`
  - `Map<String, Object> getAuthenticationData()`
  - `HttpServletRequest getRequestToProceed(HttpServletRequest request, HttpServletResponse response, UserIdentity userIdentity, LoginExtension... loginExtension)`
  - `Boolean changeResponseOnSuccess (HttpServletRequest request, HttpServletResponse response)`
  - `WorkLightAuthenticator clone()`

The `getAuthenticationData()` method is used by a login module to get the credentials that are collected by an authenticator.

Creating a custom Java authenticator (6 of 21)

- The Authenticator API is:
  - `void init(Map<String, Stri`
  - `AuthenticationResult proce`
    `request, HttpServletRespon`
    `isAccessToProtectedResourc`
  - `AuthenticationResult`
    `processAuthenticationFailu`
    `HttpServletResponse respon`
  - `AuthenticationResult`
    `processRequestAlreadyAuthe`
    `request, HttpServletRespons`
  - `Map<String, Object> getAuthenticationData()`
  - `HttpServletRequest getRequestToProceed(HttpServletRequest request, HttpServletResponse response, UserIdentity userIdentity, LoginExtension... loginExtension)`
  - `Boolean changeResponseOnSuccess (HttpServletRequest request, HttpServletResponse response)`
  - `WorkLightAuthenticator clone()`

The `getRequestToProceed()` method is called only after the login module successfully validates the credentials that were collected by an authenticator.

The `getRequestToProceed()` method is **deprecated** since IBM Worklight V5.0.0.3.

Creating a custom Java authenticator (7 of 21)

- The Authenticator API is:
  - `void init(Map<String, Stri`
  - `AuthenticationResult proce`
    `request, HttpServletRespon`
    `isAccessToProtectedResourc`
  - `AuthenticationResult`
    `processAuthenticationFailu`
    `HttpServletResponse respon`
  - `AuthenticationResult`
    `processRequestAlreadyAuthe`
    `request, HttpServletRespons`
  - `Map<String, Object> getAuthenticationData()`
  - `HttpServletRequest getRequestToProceed(HttpServletRequest request, HttpServletResponse response, UserIdentity userIdentity, LoginExtension... loginExtension)`
  - `Boolean changeResponseOnSuccess (HttpServletRequest request, HttpServletResponse response)`
  - `WorkLightAuthenticator clone()`

The `changeResponseOnSuccess()` method is called after authentication success. It is used to add data to the response after the authentication is successful.

Creating a custom Java authenticator (8 of 21)

- The Authenticator API is:
  - `void init(Map<String, Stri`
  - `AuthenticationResult proce`
    `request, HttpServletRespon`
    `isAccessToProtectedResourc`
  - `AuthenticationResult`
    `processAuthenticationFailu`
    `HttpServletResponse respon`
  - `AuthenticationResult`
    `processRequestAlreadyAuthe`
    `request, HttpServletRespons`
  - `Map<String, Object> getAuthenticationData()`
  - `HttpServletRequest getRequestToProceed(HttpServletRequest request, HttpServletResponse response, UserIdentity userIdentity, LoginExtension... loginExtension)`
  - `Boolean changeResponseOnSuccess (HttpServletRequest request, HttpServletResponse response)`
  - `WorkLightAuthenticator clone()`

The `clone()` method is used to create a deep copy of class members.

Creating a custom Java authenticator (9 of 21)

- Create a **MyCustomAuthenticator** class in the **server\java** folder
- Make sure that this class implements the **WorkLightAuthenticator** interface

```
public class MyCustomAuthenticator implements WorkLightAuthenticator {
```

- Add the **authenticationData** map to your authenticator to hold the credentials information
  - This object is retrieved and used by a login module

```
private Map<String, Object> authenticationData = null;
```

Creating a custom Java authenticator (10 of 21)

- You must add a Server runtime library dependency to use server-related classes, for example, **HttpServletRequest**.
- Right-click your Worklight project and select **Properties**.
- Select **Java Build Path** → **Libraries** and click **Add Library**.
- Select **Server Runtime** and click **Next**.
- You see a list of Server Runtimes that are installed in your Eclipse.
- Select one and click **Finish**.
- Click **OK**.

Creating a custom Java authenticator (11 of 21)

- The `init()` method is called when the authenticator is created.
- It receives options map specified in a realm definition in the **authenticationConfig.xml.**

```
@Override
public void init(Map<String, String> options) throws MissingConfigurationOptionException {
    logger.info("init");
}
```

- The `clone()` method of the authenticator creates a deep copy of the object members.

```
@Override
public WorkLightAuthenticator clone() throws CloneNotSupportedException {
    MyCustomAuthenticator otherAuthenticator = (MyCustomAuthenticator) super.clone();
    otherAuthenticator.authenticationData = new HashMap<String, Object>(authenticationData);
    return otherAuthenticator;
}
```

Creating a custom Java authenticator (12 of 21)

- The `processRequest()` method is called for each unauthenticated request to collect credentials.

```
@Override
public AuthenticationResult processRequest(HttpServletRequest request, HttpServletResponse response, boolean isAccessToProtectedR
    logger.info("MyCustomAuthenticator :: proces        uest");
    if (request.getRequestURI().contains("my_cus        auth_request_url")){
        String username = request.getParameter("use        e");
        String password = request.getParameter("

        if (null != username && null != password
            authenticationData = new HashMap<Str
            authenticationData.put("username", u
            authenticationData.put("password", p
            return AuthenticationResult.createFr
        } else {
            response.setContentType("application
            response.setHeader("Cache-Control",
            response.getWriter().print("{\"authS                                                    sword\"}");
            return AuthenticationResult.createFr
        }
    }

    if (!isAccessToProtectedResource)
        return AuthenticationResult.createFrom(A

    response.setContentType("application/json; c
    response.setHeader("Cache-Control", "no-cach
    response.getWriter().print("{\"authStatus\":
    return AuthenticationResult.createFrom(Authe
}
```

The `processRequest()` method receives the `request`, `response`, and `isAccessToProtectedResource` arguments. The method might retrieve data from a request and write data to a response, and must return a specific `AuthenticationResult` status as described in subsequent slides. Reminder: the authenticator collects the credentials for a login module; it **does not** validate them.

Creating a custom Java authenticator (13 of 21)

- The `processRequest()` method is called for each unauthenticated request to collect credentials.

```
@Override
public AuthenticationResult processRequest(HttpServletRequest request, HttpServletResponse response, boolean isAccessToProtectedR
    Logger.info("MyCustomAuthenticator :: processRequest");
    if (request.getRequestURI().contains("my_custom_auth_request_url")){
        String username = request.getParameter("username");
        String password = request.getParameter("password");

        if (null != username && null != password && username.length() > 0 && password.length() > 0){
            authenticationData = new HashMap<String, Object>();
            authenticationData.put("username", username);
            authenticationData.put("password", password);
            return AuthenticationResult.createFrom(Authentica
        } else {
            response.setContentType("application/json; charse
            response.setHeader("Cache-Control", "no-cache, mu
            response.getWriter().print("{\"authStatus\":\"req
            return AuthenticationResult.createFrom(Authentica
        }
    }

    if (!isAccessToProtectedResource)
        return AuthenticationResult.createFrom(Authentication

    response.setContentType("application/json; charset=UTF-8"
    response.setHeader("Cache-Control", "no-cache, must-reval
    response.getWriter().print("{\"authStatus\":\"required\"}
    return AuthenticationResult.createFrom(AuthenticationStat
}
```

The application sends an authentication request to a specific URL. This request URL contains **my_custom_auth_request_url** component, which is used by the authenticator to make sure that this request is an authentication request. It is advised to have a different URL component in every authenticator.

Creating a custom Java authenticator (14 of 21)

- The `processRequest()` method is called for each unauthenticated request to collect credentials.

```
@Override
public AuthenticationResult processRequest(HttpServletRequest request, HttpServletResponse response, boolean isAccessToProtectedR
    logger.info("MyCustomAuthenticator :: processRequest");
    if (request.getRequestURI().contains("my_custom_auth_request_url")){
        String username = request.getParameter("username");
        String password = request.getParameter("password");

        if (null != username && null != password && username.length() > 0 && password.length() > 0){
            authenticationData = new HashMap<String, Object>();
            authenticationData.put("username", username);
            authenticationData.put("password", password);
            return AuthenticationResult.createFrom(AuthenticationStatus.SUCCESS);
        } else {
            response.setContentType("application/json; charset=UTF-8"
            response.setHeader("Cache-Control", "no-cache, must-reval
            response.getWriter().print("{\"authStatus\":\"required\",
            return AuthenticationResult.createFrom(AuthenticationStat
        }
    }

    if (!isAccessToProtectedResource)
        return AuthenticationResult.createFrom(AuthenticationStatus.

    response.setContentType("application/json; charset=UTF-8");
    response.setHeader("Cache-Control", "no-cache, must-revalidate");
    response.getWriter().print("{\"authStatus\":\"required\"}");
    return AuthenticationResult.createFrom(AuthenticationStatus.CLIEN
}
```

The authenticator retrieves the user name and password credentials that are passed as request parameters.

Creating a custom Java authenticator (15 of 21)

- The `processRequest()` method is called for each unauthenticated request to collect credentials.

```
@Override
public AuthenticationResult processRequest(HttpServletRequest request, HttpServletResponse response, boolean isAccessToProtectedR
    logger.info("MyCustomAuthenticator :: processRequest");
    if (request.getRequestURI().contains("my_custom_auth_request_url")){
        String username = request.getParameter("username");
        String password = request.getParameter("password");

        if (null != username && null != password && username.length() > 0 && password.length() > 0){
            authenticationData = new HashMap<String, Object>();
            authenticationData.put("username", username);
            authenticationData.put("password", password);
            return AuthenticationResult.createFrom(AuthenticationStatus.SUCCESS);
        } else {
            response.setContentType("application/json; charset=UTF-8");
            response.setHeader("Cache-Control", "no-cache, must
            response.getWriter().print("{\"authStatus\":\"requi
            return AuthenticationResult.createFrom(Authenticati
        }
    }

    if (!isAccessToProtectedResource)
        return AuthenticationResult.createFrom(AuthenticationSt

    response.setContentType("application/json; charset=UTF-8");
    response.setHeader("Cache-Control", "no-cache, must-revali
    response.getWriter().print("{\"authStatus\":\"required\"}");
    return AuthenticationResult.createFrom(AuthenticationStatus
}
```

"}");

The authenticator checks the credentials for basic validity, creates an `authenticationData` object, and returns SUCCESS. SUCCESS means only credentials collection success; the login module is called after that to validate the credentials.

Creating a custom Java authenticator (16 of 21)

- The `processRequest()` method is called for each unauthenticated request to collect credentials

If there is a problem with the received credentials, the authenticator adds an errorMessage to the response and returns CLIENT_INTERACTION_REQUIRED. The client must still supply authentication data.

```
@Override
public AuthenticationResult processRequest(HttpServ
    logger.info("MyCustomAuthenticator :: processRe
    if (request.getRequestURI().contains("my_custom
        String username = request.getParameter("use
        String password = request.getParameter("pas

        if (null != username && null != password &&
            authenticationData = new HashMap<String
            authenticationData.put("username", user
            authenticationData.put("password", pass
            return AuthenticationResult.createFrom(
        } else {
            response.setContentType("application/json; charset=UTF-8");
            response.setHeader("Cache-Control", "no-cache, must-revalidate");
            response.getWriter().print("{\"authStatus\":\"required\", \"errorMessage\":\"Please enter username and password\"}");
            return AuthenticationResult.createFrom(AuthenticationStatus.CLIENT_INTERACTION_REQUIRED);
        }
    }

    if (!isAccessToProtectedResource)
        return AuthenticationResult.createFrom(AuthenticationStatus.REQUEST_NOT_RECOGNIZED);

    response.setContentType("application/json; charset=UTF-8");
    response.setHeader("Cache-Control", "no-cache, must-revalidate");
    response.getWriter().print("{\"authStatus\":\"required\"}");
    return AuthenticationResult.createFrom(AuthenticationStatus.CLIENT_INTERACTION_REQUIRED);
}
```

Creating a custom Java authenticator (17 of 21)

- The `processRequest()` method is called for each unauthenticated request to collect credentials.

```
@Override
public AuthenticationResult processRequest(HttpServletRequest re
    logger.info("MyCustomAuthenticator :: processRequest");
    if (request.getRequestURI().contains("my_custom_auth_request
        String username = request.getParameter("username");
        String password = request.getParameter("password");

        if (null != username && null != password && username.len
            authenticationData = new HashMap<String, Object>();
            authenticationData.put("username", username);
            authenticationData.put("password", password);
            return AuthenticationResult.createFrom(Authenticatio
        } else {
            response.setContentType("application/json; charset=U
            response.setHeader("Cache-Control", "no-cache, must-
            response.getWriter().print("{\"authStatus\":\"requir
            return AuthenticationResult.createFrom(Authenticatio
        }
    }

    if (!isAccessToProtectedResource)
        return AuthenticationResult.createFrom(AuthenticationStatus.REQUEST_NOT_RECOGNIZED);

    response.setContentType("application/json; charset=UTF-8");
    response.setHeader("Cache-Control", "no-cache, must-revalidate");
    response.getWriter().print("{\"authStatus\":\"required\"}");
    return AuthenticationResult.createFrom(AuthenticationStatus.CLIENT_INTERACTION_REQUIRED);
}
```

The `isAccessToProtectedResource` argument specifies whether an access attempt was made to a protected resource. If not, the method returns REQUEST_NOT_RECOGNIZED, which means that the authenticator treatment is not required, and proceed with the request as is.

Creating a custom Java authenticator (18 of 21)

- The `processRequest()` method is called for each unauthenticated request to collect credentials.

```
@Override
public AuthenticationResult processRequest(HttpServletRequest request, HttpServletResponse response, boolean isAccessToProtectedR
    logger.info("MyCustomAuthenticator :: processRequest");
    if (request.getRequestURI().contains("my_custom_auth_request_url")){
        String username = request.getParameter("username");
        String password = request.getParameter("password");

        if (null != username && null != password && username.leng
            authenticationData = new HashMap<String, Object>();
            authenticationData.put("username", username);
            authenticationData.put("password", password);
            return AuthenticationResult.createFrom(Authentication
        } else {
            response.setContentType("application/json; charset=UT
            response.setHeader("Cache-Control", "no-cache, must-r
            response.getWriter().print("{\"authStatus\":\"require
            return AuthenticationResult.createFrom(Authentication
        }
    }

    if (!isAccessToProtectedResource)
        return AuthenticationResult.createFrom(AuthenticationStat

    response.setContentType("application/json; charset=UTF-8");
    response.setHeader("Cache-Control", "no-cache, must-revalidate");
    response.getWriter().print("{\"authStatus\":\"required\"}");
    return AuthenticationResult.createFrom(AuthenticationStatus.CLIENT_INTERACTION_REQUIRED);
}
```

If the request made to a protected resource does not contain authentication data, the authenticator adds an `authStatus:required` property to the response, and also returns a CLIENT_INTERACTION_REQUIRED status.

Creating a custom Java authenticator (19 of 21)

- The authenticator `getAuthenticationData()` method is used by a login module to get collected credentials.

```
@Override
public Map<String, Object> getAuthenticationData() {
    logger.info("getAuthenticationData");
    return authenticationData;
}
```

- After the authenticated session is established, all requests are transported through the **`changeResponseOnSuccess()`** and **`processRequestAlreadyAuthenticated()`** methods.
- You can use those methods to retrieve data from requests and to update responses.

Creating a custom Java authenticator (20 of 21)

- The `changeResponseOnSuccess()` method is called after credentials are successfully validated by the login module.
- You can use this method to modify the response before you return it to the client.
- This method must return true if the response was modified, false otherwise.
- Use it to notify a client application about the authentication success.

```
@Override
public boolean changeResponseOnSuccess(HttpServletRequest request, HttpServletResponse response) throws IOException {
    logger.info("MyCustomAuthenticator :: changeResponseOnSuccess");
    if (request.getRequestURI().contains("my_custom_auth_request_url")){
        response.setContentType("application/json; charset=UTF-8");
        response.setHeader("Cache-Control", "no-cache, must-revalidate");
        response.getWriter().print("{\"authStatus\":\"complete\"}");
        return true;
    }
    return false;
}
```

Creating a custom Java authenticator (21 of 21)

- The `processRequestAlreadyAuthenticated()` method returns AuthenticationResult for authenticated requests.

```
@Override
public AuthenticationResult processRequestAlreadyAuthenticated(HttpServletRequest request,
    logger.info("processRequestAlreadyAuthenticated");
    return AuthenticationResult.REQUEST_NOT_RECOGNIZED;
}
```

- If the login module returns an authentication failure, `processAuthenticationFailure()` is called. This method writes an error message to a response body, and returns CLIENT_INTERACTION_REQUIRED status.

```
@Override
public AuthenticationResult processAuthenticationFailure(HttpServletRequest request, HttpServletResponse response,
        String errorMessage) throws IOException, ServletException {

    logger.info("processAuthenticationFailure");
    response.setContentType("application/json; charset=UTF-8");
    response.setHeader("Cache-Control", "no-cache, must-revalidate");
    response.getWriter().print("{\"authRequired\":true, \"errorMessage\":\"" + errorMessage + "\"}");
    return AuthenticationResult.CLIENT_INTERACTION_REQUIRED;
}
```

Agenda

- Authentication introduction
- Configuring authenticationConfig.xml
- Creating a custom Java authenticator
- Creating a custom Java login module
- Creating client-side authentication components
- Examining the result

Creating a custom Java login module (1 of 20)

- The login module API is:
  - `void init(Map<String, String> options)`
  - `boolean login(Map<String, Object> authenticationData)`
  - `UserIdentity createIdentity(String loginModule)`
  - `void logout()`
  - `void abort()`
  - `WorkLightAuthLoginModu`

The `init()` method of the login module is called when the login module instance is created. This method receives the options that are specified in the login module definition of the **authenticationConfig.xml** file.

Creating a custom Java login module (2 of 20)

- The login module API is:
  - `void init(Map<String, String> options)`
  - `boolean login(Map<String, Object> authenticationData)`
  - `UserIdentity createIdentity(String loginModule)`
  - `void logout()`
  - `void abort()`
  - `WorkLightAuthLoginModu`

The `login()` method of the login module is used to validate the credentials that are collected by the authenticator.

Creating a custom Java login module (3 of 20)

- The login module API is:
  - `void **init**(Map<String, String> options)`
  - `boolean **login**(Map<String, Object> authenticationData)`
  - `UserIdentity **createIdentity**(String loginModule)`
  - `void **logout**()`
  - `void **abort**()`
  - `WorkLightAuthLoginModu`

The `createIdentity()` method of the login module is used to create a `userIdentity` object after the credentials validation succeeds.

Creating a custom Java login module (4 of 20)

- The login module API is:
  - `void init(Map<String, String> options)`
  - `boolean login(Map<String, Object> authenticationData)`
  - `UserIdentity createIdentity(String loginModule)`
  - `void logout()`
  - `void abort()`
  - `WorkLightAuthLoginModu`

The `logout()` and `abort()` methods are used to clean up cached data after a logout or an authentication abort occurs.

Creating a custom Java login module (5 of 20)

- The login module API is:
  - `void init(Map<String, S`
  - `boolean login(Map<Stri authenticationData)`
  - `UserIdentity createIde`
  - `void logout()`
  - `void abort()`
  - `WorkLightLoginModule clone()`

The `clone()` method is used to create a deep copy of the class members.

Creating a custom Java login module (6 of 20)

- Create a **MyCustomLoginModule** class in the **server\java** folder.
- Make sure that this class implements the **WorkLightAuthLoginModule** interface.

```
public class MyCustomLoginModule implements WorkLightAuthLoginModule {
```

- Add two private class members, USERNAME and PASSWORD, to hold the user credentials

```
private String USERNAME;
private String PASSWORD;
```

Creating a custom Java login module (7 of 20)

- The `init()` method is called when the login module instance is created. It receives a map of options that are specified in a login module definition in the **authenticationConfig.xml** file.

```
@Override
public void init(Map<String, String> options) throws MissingConfigurationOptionException {
    logger.info("init");
}
```

- The `clone()` method of the login module creates a deep copy of the object members.

```
@Override
public MyCustomLoginModule clone() throws CloneNotSupportedException {
    return (MyCustomLoginModule) super.clone();
}
```

Creating a custom Java login module (8 of 20)

- The `login()` method is called after the authenticator returns SUCCESS status.

```
@Override
public boolean login(Map<String, Object> authenticationData) {
    logger.info("MyCustomLoginModule :: login");
    USERNAME = (String) authenticationData.get("username");
    PASSWORD = (String) authenticationData.get("password");

    if (USERNAME.equals("wluser") &
        return true;
    else
        throw new RuntimeException(
}
```

When called, the `login()` method gets an `authenticationData` object from the authenticator.

Creating a custom Java login module (9 of 20)

- The `login()` method is called after the authenticator returns SUCCESS status.

```
@Override
public boolean login(Map<String, Object> authenticationData) {
    Logger.info("MyCustomLoginModule :: login");
    USERNAME = (String) authenticationData.get("username");
    PASSWORD = (String) authenticationData.get("password");

    if (USERNAME.equals("wluser") && PASSWORD.equals("12345"))
        return true;
    else
        throw new RuntimeException("
}
```

The `login()` method retrieves the user name and password credentials that the authenticator previously stored.

Creating a custom Java login module (10 of 20)

- The `login()` method is called after the authenticator returns SUCCESS status.

```
@Override
public boolean login(Map<String, Object> authenticationData) {
    logger.info("MyCustomLoginModule :: login");
    USERNAME = (String) authenticationData.get("username");
    PASSWORD = (String) authenticationData.get("password");

    if (USERNAME.equals("wluser") && PASSWORD.equals("12345"))
        return true;
    else
        throw new RuntimeException(
}
```

In this example, the login module validates the credentials against hardcoded values. You can implement your own validation rules. The `login()` method returns **true** if the credentials are valid.

Creating a custom Java login module (11 of 20)

- The `login()` method is called after the authenticator returns SUCCESS status.

```
@Override
public boolean login(Map<String, Object> authenticationData) {
    logger.info("MyCustomLoginModule :: login");
    USERNAME = (String) authenticationData.get("username");
    PASSWORD = (String) authenticationData.get("password");

    if (USERNAME.equals("wluser") && PASSWORD.equals("12345"))
        return true;
    else
        throw new RuntimeException("Invalid credentials");
}
```

If the credential validation fails, the `login()` method can either return **false** or throw a `RuntimeException` with a text that is returned to the authenticator as an `errorMessage` parameter.

Creating a custom Java login module (12 of 20)

- The `createIdentity()` method is called when the login() method returned **true**. It is used to create an authenticated user identity object.

```
@Override
public UserIdentity createIdentity(String loginModule) {
    logger.info("MyCustomLoginModule :: createIdentity");

    HashMap<String, Object> customAttributes = new HashMap<String, Object>();
    customAttributes.put("AuthenticationDate", new Date());

    UserIdentity identity = new UserIdentity(loginModule, USERNAME, null, null, customAttributes, PASSWORD);
    return identity;
}
```

After the `login()` method returns **true**, the `createIdentity()` method is called. It is used to create a `UserIdentity` object. You can store your own custom attributes in it to use later in Java or adapter code.

Creating a custom Java login module (13 of 20)

- The `createIdentity()` method is called when the `login()` method returned **true**. It is used to create an authenticated user identity object.

```
@Override
public UserIdentity createIdentity(String loginModule) {
    logger.info("MyCustomLoginModule :: createIdentity");

    HashMap<String, Object> customAttributes = new HashMap<String, Object>();
    customAttributes.put("AuthenticationDate", new Date());

    UserIdentity identity = new UserIdentity(loginModule, USERNAME, null, null, customAttributes, PASSWORD);
    return identity;
}
```

The `UserIdentity` object contains user information. Its constructor is:

```
public UserIdentity(String loginModule,
                    String name,
                    String displayName,
                    Set<String> roles,
                    Map<String, Object> attributes,
                    Object credentials)
```

Creating a custom Java login module (14 of 20)

- The `createIdentity()` method is called when the `login()` method returned **true**. It is used to create an authenticated user identity object.

Login module name to set user for

```
@Override
public UserIdentity createIdentity(
    logger.info("MyCustomLoginModule :: createIden ity");

    HashMap<String, Object> customAttributes = new HashMap<String, Object>();
    customAttributes.put("AuthenticationDate", new Date());

    UserIdentity identity = new UserIdentity(loginModule, USERNAME, null, null, customAttributes, PASSWORD);
    return identity;
}
```

The UserIdentity object contains user information. Its constructor is:

```
public UserIdentity(String loginModule,
                    String name,
                    String displayName,
                    Set<String> roles,
                    Map<String, Object> attributes,
                    Object credentials)
```

Creating a custom Java login module (15 of 20)

- The `createIdentity()` method is called when the `login()` method returned **true**. It is used to create an authenticated user identity object.

A unique user identifier

```
@Override
public UserIdentity createIdentity(String logi
    logger.info("MyCustomLoginModule :: createIdentity");

    HashMap<String, Object> customAttributes = new HashMap<S cing, Object>();
    customAttributes.put("AuthenticationDate", new Date());

    UserIdentity identity = new UserIdentity(loginModule, USERNAME, null, null, customAttributes, PASSWORD);
    return identity;
}
```

The `UserIdentity` object contains user information. Its constructor is:

```
public UserIdentity(String loginModule,
                    String name,
                    String displayName,
                    Set<String> roles,
                    Map<String, Object> attributes,
                    Object credentials)
```

Creating a custom Java login module (16 of 20)

- The `createIdentity()` method is called when the `login()` method returned **true**. It is used to create an authenticated user identity object.

The `UserIdentity` object contains user information. Its constructor is:

```
public UserIdentity(String loginModule,
                    String name,
                    String displayName,
                    Set<String> roles,
                    Map<String, Object> attributes,
                    Object credentials)
```

Creating a custom Java login module (17 of 20)

- The `createIdentity()` method is called when the `login()` method returned **true**. It is used to create an authenticated user identity object.

User Java security roles

```
@Override
public UserIdentity createIdentity(String loginModule) {
    logger.info("MyCustomLoginModule :: createIdentity");

    HashMap<String, Object> customAttributes = new HashMap<String, Object>( );
    customAttributes.put("AuthenticationDate", new Date());

    UserIdentity identity = new UserIdentity(loginModule, USERNAME, null, null, customAttributes, PASSWORD);
    return identity;
}
```

The `UserIdentity` object contains user information. Its constructor is:

```
public UserIdentity(String loginModule,
                    String name,
                    String displayName,
                    Set<String> roles,
                    Map<String, Object> attributes,
                    Object credentials)
```

Creating a custom Java login module (18 of 20)

- The `createIdentity()` method is called when the `login()` method returned **true**. It is used to create an authenticated user identity object.

```
@Override
public UserIdentity createIdentity(String loginModule) {
    logger.info("MyCustomLoginModule :: createIdentity");

    HashMap<String, Object> customAttributes = new HashMap<String, Object>();
    customAttributes.put("AuthenticationDate", new Date());

    UserIdentity identity = new UserIdentity(loginModule, USERNAME, null, null, customAttributes, PASSWORD);
    return identity;
}
```

Custom user attributes

The `UserIdentity` object contains user information. Its constructor is:

```
public UserIdentity(String loginModule,
                    String name,
                    String displayName,
                    Set<String> roles,
                    Map<String, Object> attributes,
                    Object credentials)
```

Creating a custom Java login module (19 of 20)

- The `createIdentity()` method is called when the `login()` method returned **true**. It is used to create an authenticated user identity object.

```
@Override
public UserIdentity createIdentity(String loginModule) {
    logger.info("MyCustomLoginModule :: createIdentity");

    HashMap<String, Object> customAttributes = new HashMap<String, Object>();
    customAttributes.put("AuthenticationDate", new Date());

    UserIdentity identity = new UserIdentity(loginModule, USERNAME, null, null, customAttributes, PASSWORD);
    return identity;
}
```

Sensitive user credentials that are not to be persisted

The `UserIdentity` object contains user information. Its constructor is:

```
public UserIdentity(String loginModule,
                    String name,
                    String displayName,
                    Set<String> roles,
                    Map<String, Object> attributes,
                    Object credentials)
```

Creating a custom Java login module (20 of 20)

- The `logout()` and `abort()` methods are used to clean up class members after the user logs out or aborts the authentication flow.

```
@Override
public void logout() {
    logger.info("MyCustomLoginModule :: logout");
    USERNAME = null;
    PASSWORD = null;
}

@Override
public void abort() {
    logger.info("MyCustomLoginModule :: abort");
    USERNAME = null;
    PASSWORD = null;
}
```

Agenda

- Authentication introduction
- Configuring authenticationConfig.xml
- Creating a custom Java authenticator
- Creating a custom Java login module
- Creating client-side authentication components
- Examining the result

Creating client-side authentication components (1 of 13)

- Create a Worklight application.
- The application consists of two main elements:
  - `<div id=“AppBody”>` element is used to display the application content.
  - `<div id=“AuthBody”>` element is used for authentication form purposes.
- When authentication is required, the application hides the AppBody and shows the AuthBody. When authentication is complete, it does the opposite.

Creating client-side authentication components (2 of 13)

- Start by creating an AppBody.
- It has a basic structure and functions.

```
<div id="AppBody">
    <div class="header">
        <h1>Custom Login Module</h1>
    </div>
    <div class="wrapper">
        <input type="button" value="Call protected adapter proc" onclick="getSecretData()" />
        <input type="button" value="Logout"
               onclick="WL.Client.logout('CustomAuthenticatorRealm',{onSuccess: WL.Client.reloadApp})" />
    </div>
</div>
```

- Buttons are used to invoke the `getSecretData` procedure and to log out.

Creating client-side authentication components (3 of 13)

- AuthBody contains the following elements:

```
<div id="AuthBody" style="display: none">
    <div id="loginForm">
        Username:<br/>
        <input type="text" id="usernameInputField"  /><br />
        Password:<br/>
        <input type="password" id="passwordInputField" /><br/>
        <input type="button" id="loginButton" value="Login" />
        <input type="button" id="cancelButton" value="Cancel" />
    </div>
</div>
```

  - Username and Password input fields
  - Login and Cancel buttons
- AuthBody is styled as `display:none` because it must not be displayed before the server requests the authentication.

Creating client-side authentication components (4 of 13)

- The following API describes how to create the challenge handler and implement its functionality:

```
var myChallengeHandler = WL.Client.createChallengeHandler("realm-name");

myChallengeHandler.isCustomResponse = function (response){
    return false;
};

myChallengeHandler.handleChallenge = function (response){
};
```

Use `WL.Client.createChallengeHandler()` to create a challenge handler object. Supply a realm name as a parameter.

Create a challenge handler to define a customized authentication flow. In your challenge handler, do not add code that modifies the user interface when this modification is not related to the authentication flow.

Creating client-side authentication components (5 of 13)

- The following API describes how to create the challenge handler and implement its functionality:

```
var myChallengeHandler = WL.Client.createChallengeHandler("realm-name");

myChallengeHandler.isCustomResponse = function (response){
    return false;
};

myChallengeHandler.handleChallenge = function (response){
};
```

The `isCustomResponse` function of the challenge handler is called each time that a response is received from the server.
It is used to detect whether the response contains data that is related to this challenge handler. It must return **true** or **false**.

Creating client-side authentication components (6 of 13)

- The following API describes how to create the challenge handler and implement its functionality.

```
var myChallengeHandler = WL.Client.createChallengeHandler("realm-name");

myChallengeHandler.isCustomResponse = function (response){
    return false;
};

myChallengeHandler.handleChallenge = function (response){
};
```

If `isCustomResponse` returns **true**, the framework invokes the `handleChallenge()` function. This function is used to perform required actions, such as hide application screen and show login screen.

Creating client-side authentication components (7 of 13)

- In addition to the methods that the developer must implement, the challenge handler contains functionality that the developer might want to use:
  - `submitLoginForm()` is used to send collected credentials to a specific URL. The developer can also specify request parameters, headers, and callback.
  - `submitSuccess()` notifies the Worklight framework that the authentication successfully finished. The Worklight framework then automatically issues the original request that triggered the authentication
  - `submitFailure()` notifies the Worklight framework that the authentication completed with a failure. The Worklight framework then disposes of the original request that triggered the authentication

  **** Note that each one of these functions should be attached to its object. For example:*** *`myChallengeHandler.submitSucces()`*

- You use those functions during the implementation of the challenge handler in the next slides.

Creating client-side authentication components (8 of 13)

- Create a challenge handler.

```
var customAuthenticatorRealmChallengeHandler = WL.Client.createChallengeHandler("CustomAuthenticatorRealm");

customAuthenticatorRealmChallengeHandler.isCustomResponse = function(response) {
    if (!response || !response.responseJSON) {
        return false;
    }

    if (response.responseJSON.authStatus)
        return true;
    else
        return false;
};

customAuthenticatorRealmChallengeHandler.handleChallenge = fu
    var authStatus = response.responseJSON.authStatus;

    if (authStatus == "required"){
        $('#AppBody').hide();
        $('#AuthBody').show();
        $('#passwordInputField').val('');
        if (response.responseJSON.errorMessage){
            alert(response.responseJSON.errorMessage);
        }
    } else if (authStatus == "complete"){
        $('#AppBody').show();
        $('#AuthBody').hide();
        customAuthenticatorRealmChallengeHandler.submitSu
    }
};
```

If the challenge JSON contains `authStatus` property, return **true**, otherwise return **false.**

Creating client-side authentication components (9 of 13)

- Create a challenge handler.

```
var customAuthenticatorRealmChallengeHandler =

customAuthenticatorRealmChallengeHandler.isCust
    if (!response || !response.responseJSON) {
        return false;
    }

    if (response.responseJSON.authStatus)
        return true;
    else
        return false;
};

customAuthenticatorRealmChallengeHandler.handleChallenge = function(response){
    var authStatus = response.responseJSON.authStatus;

    if (authStatus == "required"){
        $('#AppBody').hide();
        $('#AuthBody').show();
        $('#passwordInputField').val('');
        if (response.responseJSON.errorMessage){
            alert(response.responseJSON.errorMessage);
        }
    } else if (authStatus == "complete"){
        $('#AppBody').show();
        $('#AuthBody').hide();
        customAuthenticatorRealmChallengeHandler.submitSuccess();
    }
};
```

If the `authStatus` property equals "required", show login form, clean up password input field, and display the error message if it exists.

Creating client-side authentication components (10 of 13)

- Create a challenge handler.

```
var customAuthenticatorRealmChallengeHandler = WL.Client.createChallengeHandler("CustomAuthenticatorRealm");

customAuthenticatorRealmChallengeHandler.isCust
    if (!response || !response.responseJSON) {
        return false;
    }

    if (response.responseJSON.authStatus)
        return true;
    else
        return false;
};

customAuthenticatorRealmChallengeHandler.handl
    var authStatus = response.responseJSON.auth

    if (authStatus == "required"){
        $('#AppBody').hide();
        $('#AuthBody').show();
        $('#passwordInputField').val('');
        if (response.responseJSON.errorMessage){
            alert(response.responseJSON.errorMessage);
        }
    } else if (authStatus == "complete"){
        $('#AppBody').show();
        $('#AuthBody').hide();
        customAuthenticatorRealmChallengeHandler.submitSuccess();
    }
};
```

if `authStatus` equals “complete”, hide the login screen, return to the application, and notify Worklight framework that authentication is successfully complete.

Creating client-side authentication components (11 of 13)

- Create a challenge handler.

```
$('#loginButton').bind('click', function () {
    var reqURL = '/my_custom_auth_request_url';
    var options = {};
    options.parameters = {
        username : $('#usernameInputField').val(),
        password : $('#passwordInputField').val()
    };
    options.headers = {};
    customAuthenticatorRealmChallengeHandler.submitLoginForm(reqURL, options,
            customAuthenticatorRealmChallengeHandler.submitLoginFormCallback);
});

$('#cancelButton').bind('click', funct
    $('#AppBody').show();
    $('#AuthBody').hide();
    customAuthenticatorRealmChallengeH
});
```

Clicking a **login** button triggers the function that collects the user name and password from HTML input fields, and submits them to server. You can set request headers here and specify callback functions.

Creating client-side authentication components (12 of 13)

- Create a challenge handler.

```
$('#loginButton').bind('click', function
    var reqURL = '/my_custom_auth_request
    var options = {};
    options.parameters = {
        username : $('#usernameInputFiel
        password : $('#passwordInputFiel
    };
    options.headers = {};
    customAuthenticatorRealmChallengeHandl
            customAuthenticatorRealmChallengeHandler.submitLoginFormCallback);
});

$('#cancelButton').bind('click', function () {
    $('#AppBody').show();
    $('#AuthBody').hide();
    customAuthenticatorRealmChallengeHandler.submitFailure();
});
```

Clicking a **cancel** button hides authBody, shows appBody, and notifies the Worklight framework that authentication failed.

Creating client-side authentication components (13 of 13)

- Create a challenge handler.

```
customAuthenticatorRealmChallengeHandler.submitLoginFormCallback = function(response) {
    var isLoginFormResponse = customAuthenticatorRealmChallengeHandler.isCustomResponse(response);
    if (isLoginFormResponse){
        customAuthenticatorRealmChallengeHandler.handleChallenge(response);
    }
};
```

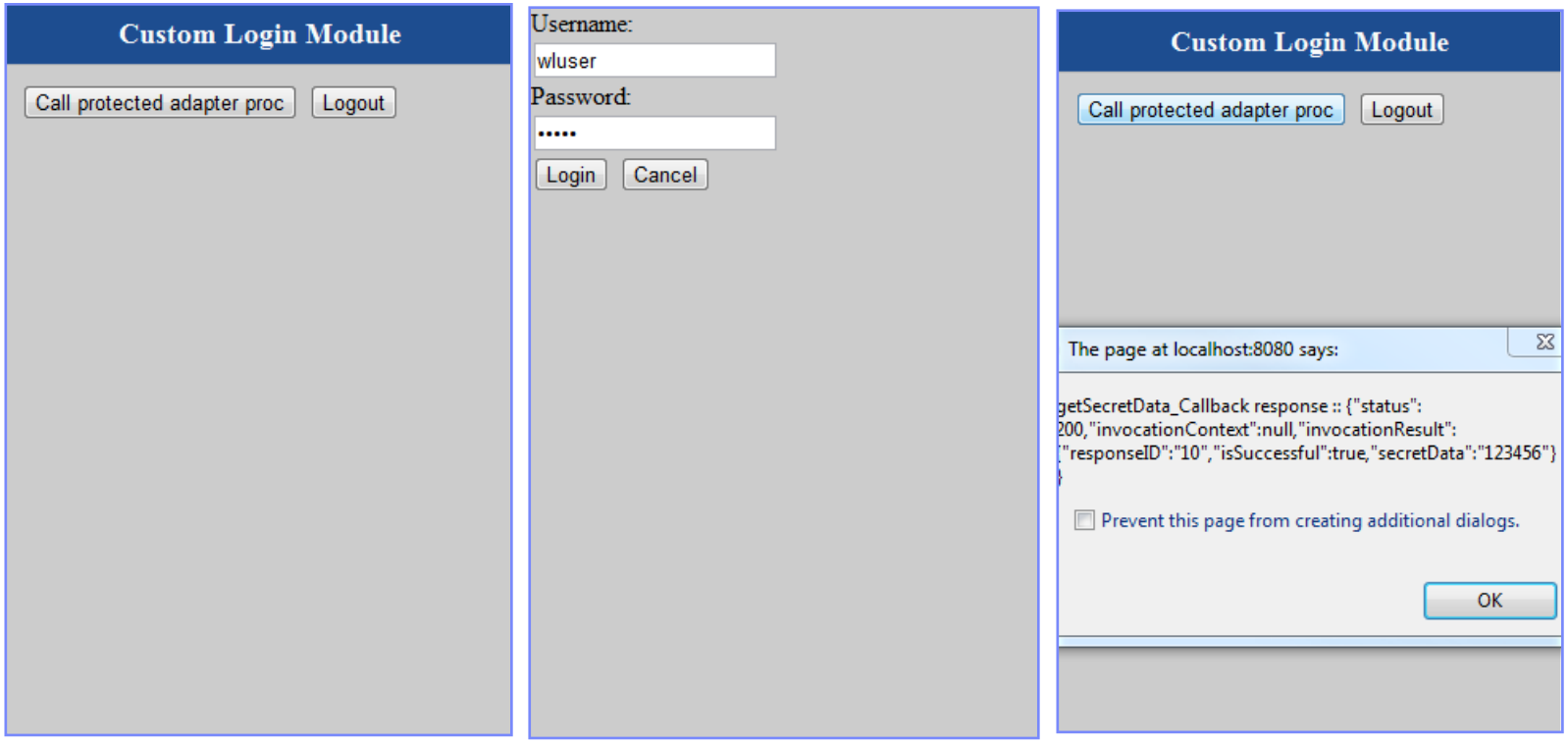
The callback function checks the response for the containing server challenge once again. If the challenge is found, the `handleChallenge()` function is called again.

Agenda

- Authentication introduction
- Configuring authenticationConfig.xml
- Creating a custom Java authenticator
- Creating a custom Java login module
- Creating client-side authentication components
- Examining the result

Examining the Result

- The sample for this training module can be found in the Getting Started page of the IBM Worklight documentation website at http://www.ibm.com/mobile-docs
- Enter *wluser and 12345* as the user credentials



Notices

- Permission for the use of these publications is granted subject to these terms and conditions.
- This information was developed for products and services offered in the U.S.A.
- IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.
- IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:
  - IBM Director of Licensing
    IBM Corporation
    North Castle Drive
    Armonk, NY 10504-1785
    U.S.A.
- For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:
  - Intellectual Property Licensing
    Legal and Intellectual Property Law
    IBM Japan Ltd.
    1623-14, Shimotsuruma, Yamato-shi
    Kanagawa 242-8502 Japan
- **The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.
- This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.
- Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.
- IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.
- Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:
  - IBM Corporation
    Dept F6, Bldg 1
    294 Route 100
    Somers NY 10589-3216
    USA
- Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.
- The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.
- Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

COPYRIGHT LICENSE:

- This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.
- Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:
  - © (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

Privacy Policy Considerations

- IBM Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.
- Depending upon the configurations deployed, this Software Offering may use session cookies that collect session information (generated by the application server). These cookies contain no personally identifiable information and are required for session management. Additionally, persistent cookies may be randomly generated to recognize and manage anonymous users. These cookies also contain no personally identifiable information and are required.
- If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent. For more information about the use of various technologies, including cookies, for these purposes, see IBM’s Privacy Policy at http://www.ibm.com/privacy and IBM’s Online Privacy Statement at http://www.ibm.com/privacy/details the sections entitled “Cookies, Web Beacons and Other Technologies” and the “IBM Software Products and Software-as-a-Service Privacy Statement” at http://www.ibm.com/software/info/product-privacy.

Support and comments

- For the entire IBM Worklight documentation set, training material and online forums where you can post questions, see the IBM website at:
  - http://www.ibm.com/mobile-docs
- **Support**
  - Software Subscription and Support (also referred to as Software Maintenance) is included with licenses purchased through Passport Advantage and Passport Advantage Express. For additional information about the International Passport Advantage Agreement and the IBM International Passport Advantage Express Agreement, visit the Passport Advantage website at:
    - http://www.ibm.com/software/passportadvantage
  - If you have a Software Subscription and Support in effect, IBM provides you assistance for your routine, short duration installation and usage (how-to) questions, and code-related questions. For additional details, consult your IBM Software Support Handbook at:
    - http://www.ibm.com/support/handbook
- **Comments**
  - We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this document. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.
  - For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.
  - When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state.
  - Thank you for your support.
  - Submit your comments in the IBM Worklight Developer Edition support community at:
    - https://www.ibm.com/developerworks/mobile/worklight/connect.html
  - If you would like a response from IBM, please provide the following information:
    - Name
    - Address
    - Company or Organization
    - Phone No.
    - Email address

Thank You