Form-based authentication

Overview

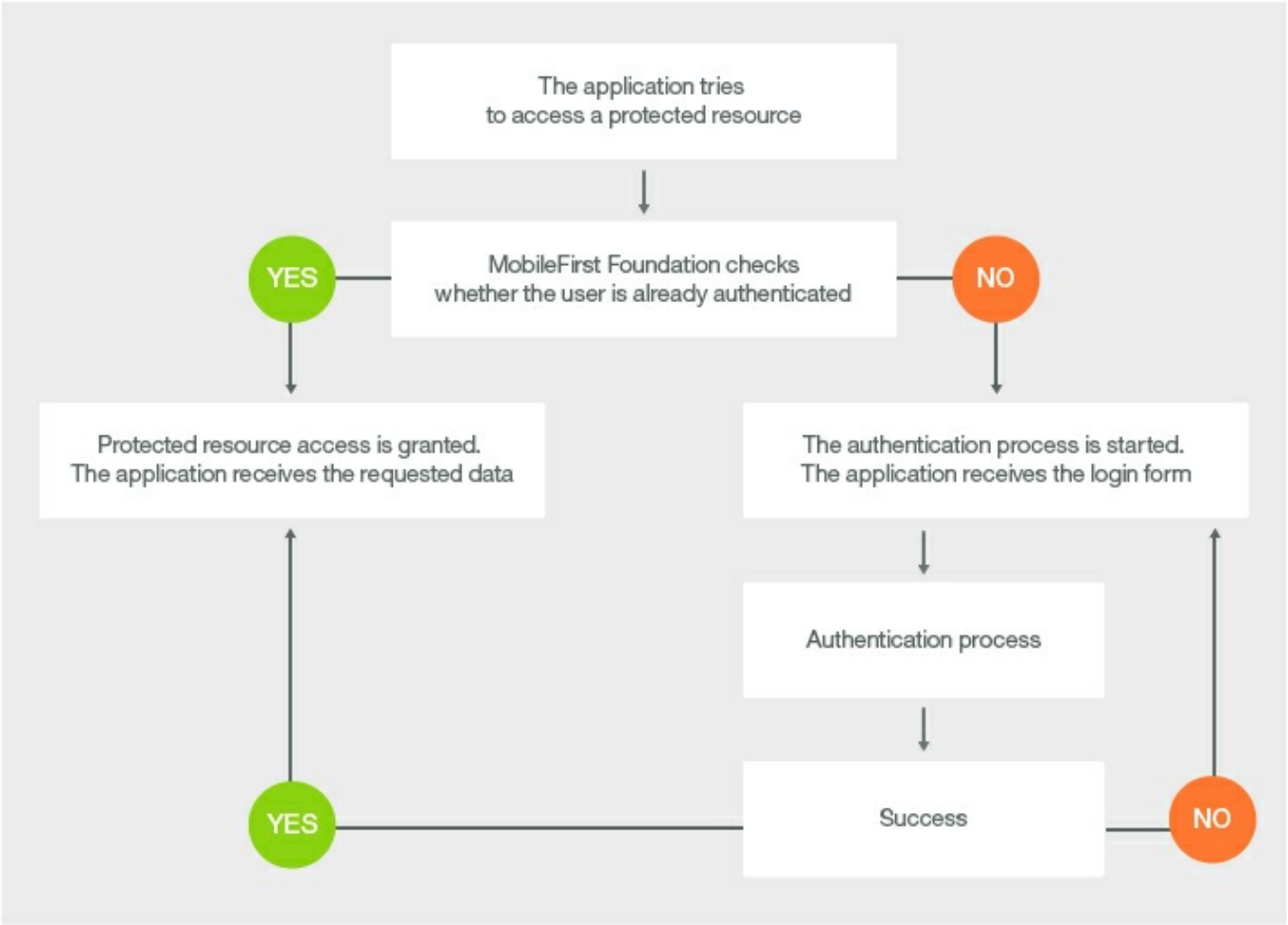
In form-based authentication, the HTML code of a login form is returned in the server response when the application tries to access a protected resource.
Although form-based authentication is best suited for desktop and web environments, where you actually display and use the returned login form, you can also use this authentication mode in mobile applications.

To use form-based authentication, you must use a login module to validate the received credentials.
In this tutorial, you implement a simple form-based authentication mechanism that is based on a user name and a password.

This tutorial covers the following topics:

- Configuring the authenticationConfig.xml file
- Protecting a JavaScript adapter
- Protecting a Java adapter
- Creating client-side authentication components

The following diagram illustrates the form-based authentication process.



Configuring the authenticationConfig.xml file

The default `authenticationConfig.xml` file already contains a sample realm that is configured to use a form-based authenticator.

```
<realm name="SampleAppRealm" loginModule="StrongDummy">
  <className>com.worklight.core.auth.ext.FormBasedAuthenticator</className
>
</realm>
```

Notice the `StrongDummy` login module that is used for this realm.

```
<loginModule name="StrongDummy">
  <className>com.worklight.core.auth.ext.NonValidatingLoginModule</className
>
</loginModule>
```

The `NonValidatingLoginModule` parameter means that the user credentials are not validated. In other words: any combination of user name and password is valid.

Define a security test that uses the `SampleAppRealm`. Remember the security test name, to use it in the subsequent steps.
Note: If you use Java adapters (and not JavaScript adapters), this step is not necessary.

```
<br />
<customSecurityTest name="AuthSecurityTest">
  <test isInternalUserID="true" realm="SampleAppRealm" /
>
</customSecurityTest>
```

Protecting a JavaScript adapter

1. Create an adapter and name it `AuthAdapter`.
2. Add a `getSecretData` procedure and protect it with the security test that you created previously.

   ```
   <procedure name="getSecretData" securityTest="AuthSecurityTest"/>
   ```

   In this module, the `getSecretData` procedure returns some hardcoded value:

   ```
   function getSecretData(){
      return {
         secretData: '123456'
      };
   }
   ```

Protecting a Java adapter

1. Create a Java adapter.

2. Add a `getSecretData` method and protect it with the realm that you created previously. In this module, the `getSecretData` procedure returns some hardcoded value:

```
@GET
@Produces("application/json")
@OAuthSecurity(scope="SampleAppRealm")
public JSONObject getSecretData(){
 JSONObject result = new JSONObject();
 result.put("secretData", "123456");
 return result;
}
```

3. To set your new realm as the default user identity for the application, add this option to the application descriptor:

```
<userIdentityRealms>SampleAppRealm</userIdentityRealms>
```

> To learn more about application descriptor properties, see the user documentation.

> To learn more about authenticators, see the topic about implementing form-based authenticators, in the user documentation.