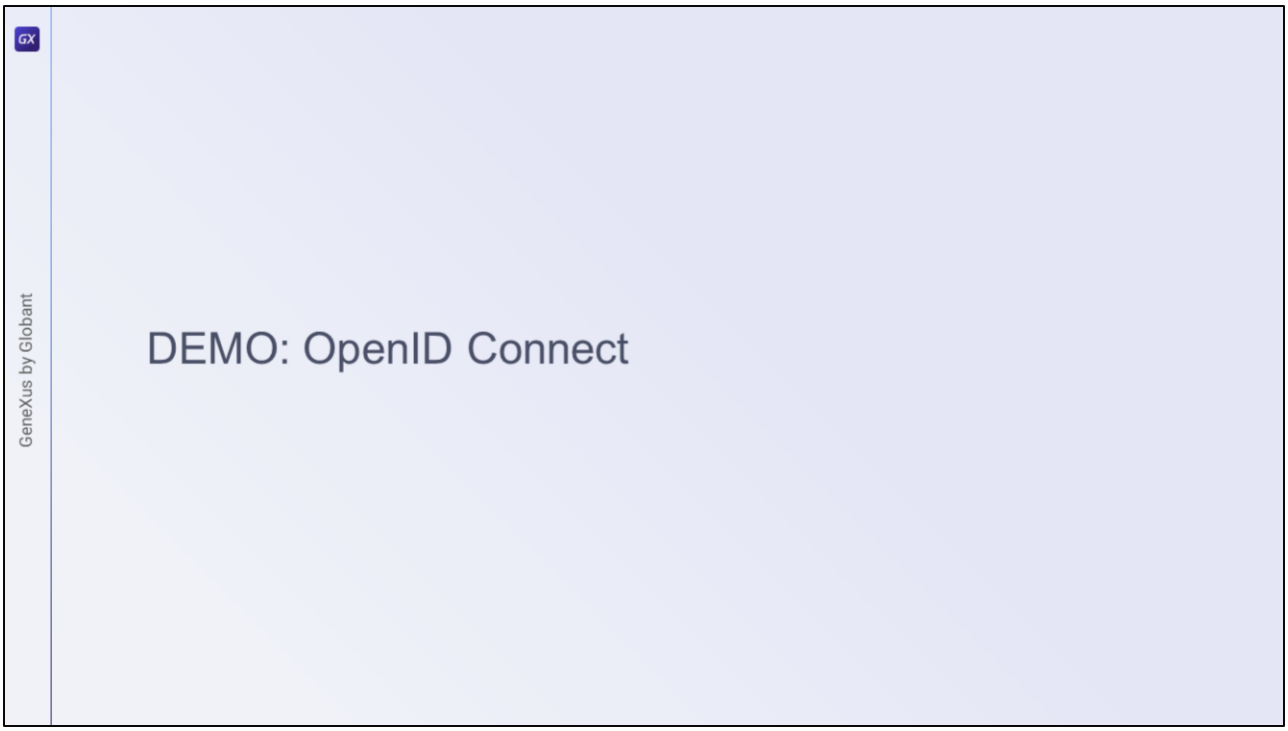
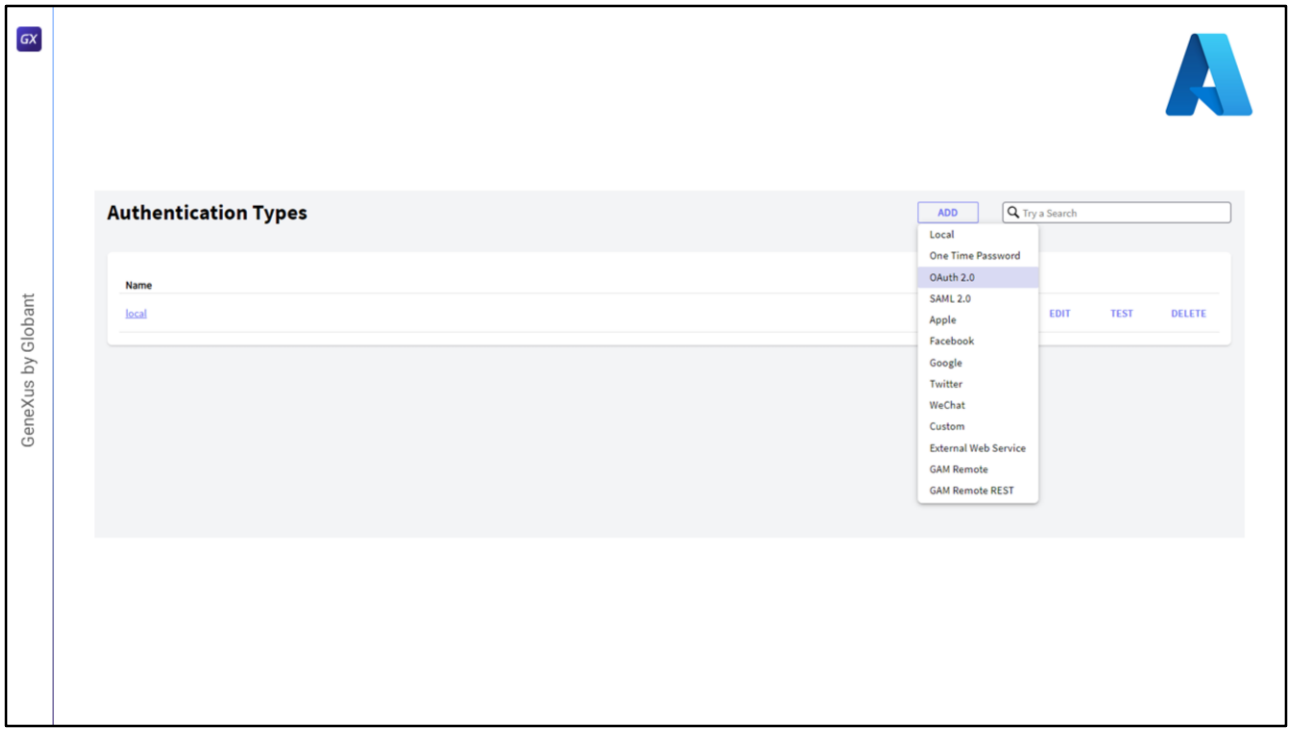


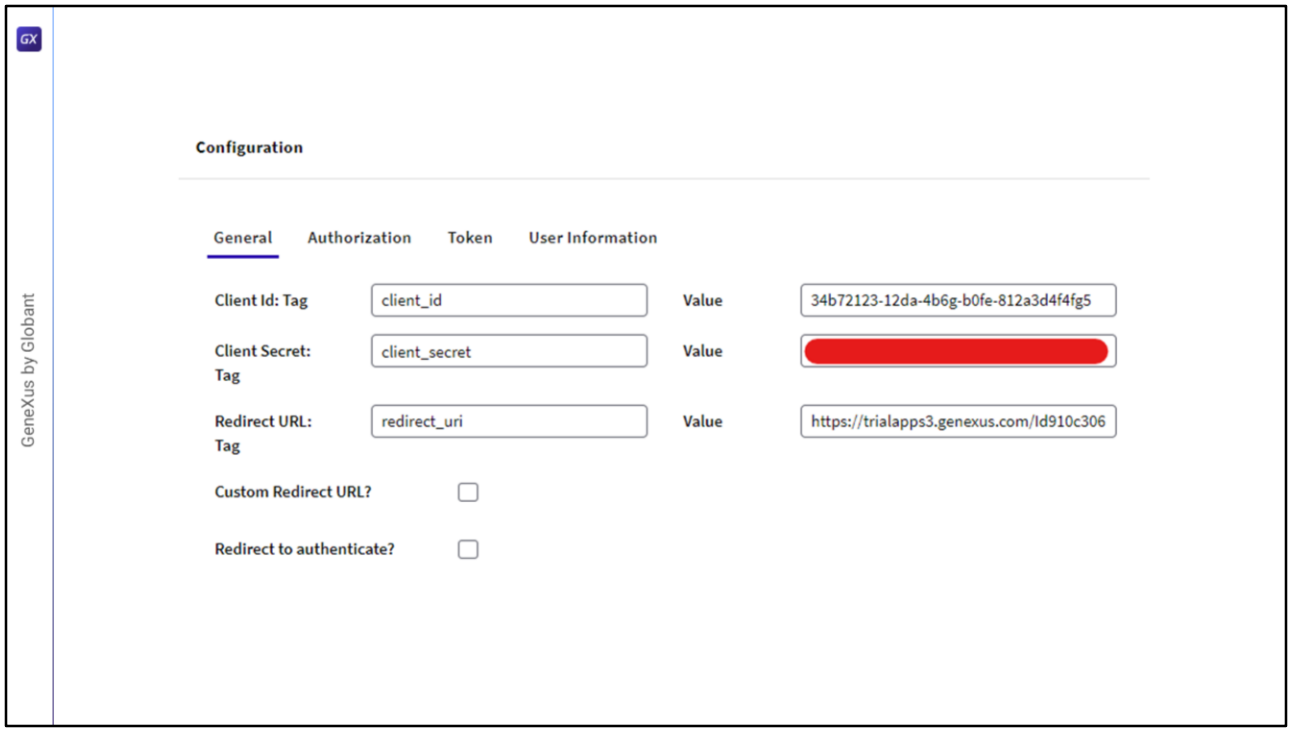
First demo: OpenID Connect



For this demo, we will use the OAuth 2.0 protocol in GAM. Our identity provider will be Azure Active Directory from Microsoft.

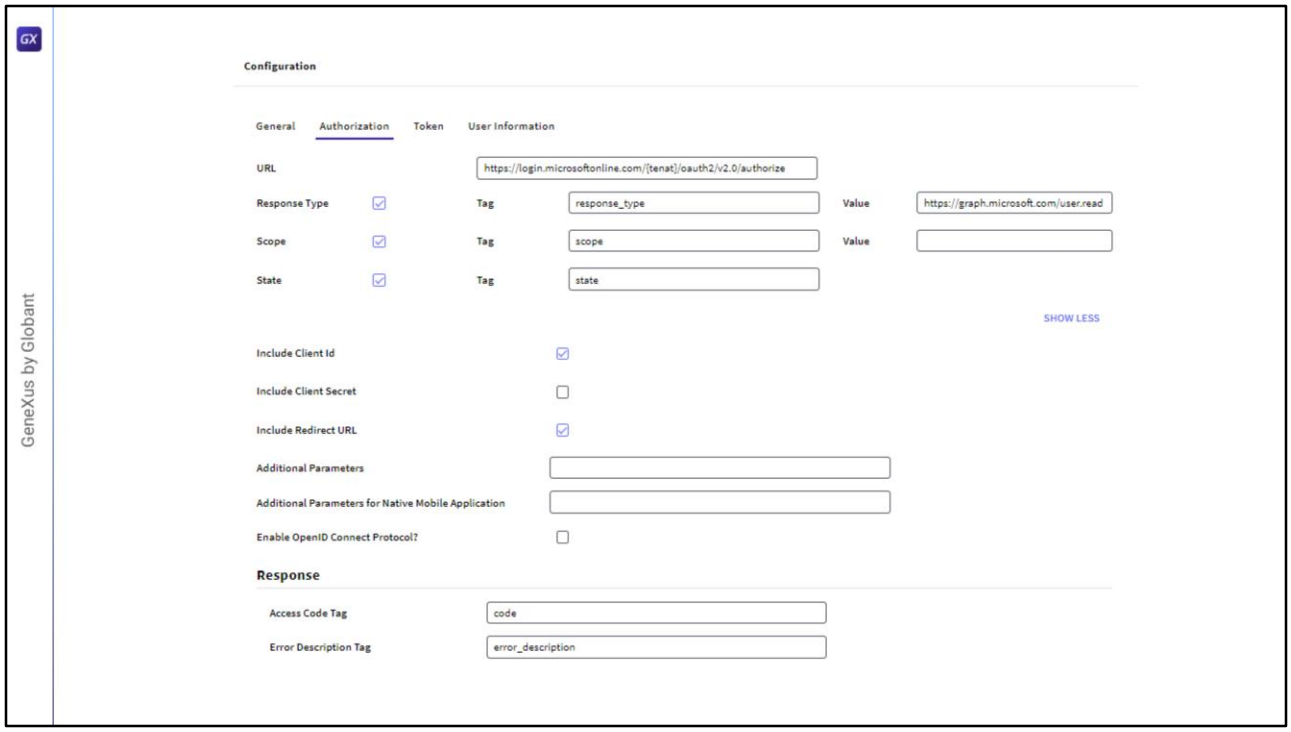
We will assume that configuration on the Azure side has already been made correctly and we will not go into detail about it. To learn how to do it, you can see a detailed article about it in the GeneXus Wiki.

First, we create a new GAM OAuth 2.0 Authentication Type and define the basic concepts, such as Name, Description, etc.



In the General tab, the following must be defined:
First, we set the Client ID and Client Secret obtained from Azure.
The redirection URL must be the Base URL of our application's back end.

As we said in the previous video, we will not select the Redirect option to authenticate because we want to log in from the GAM itself.

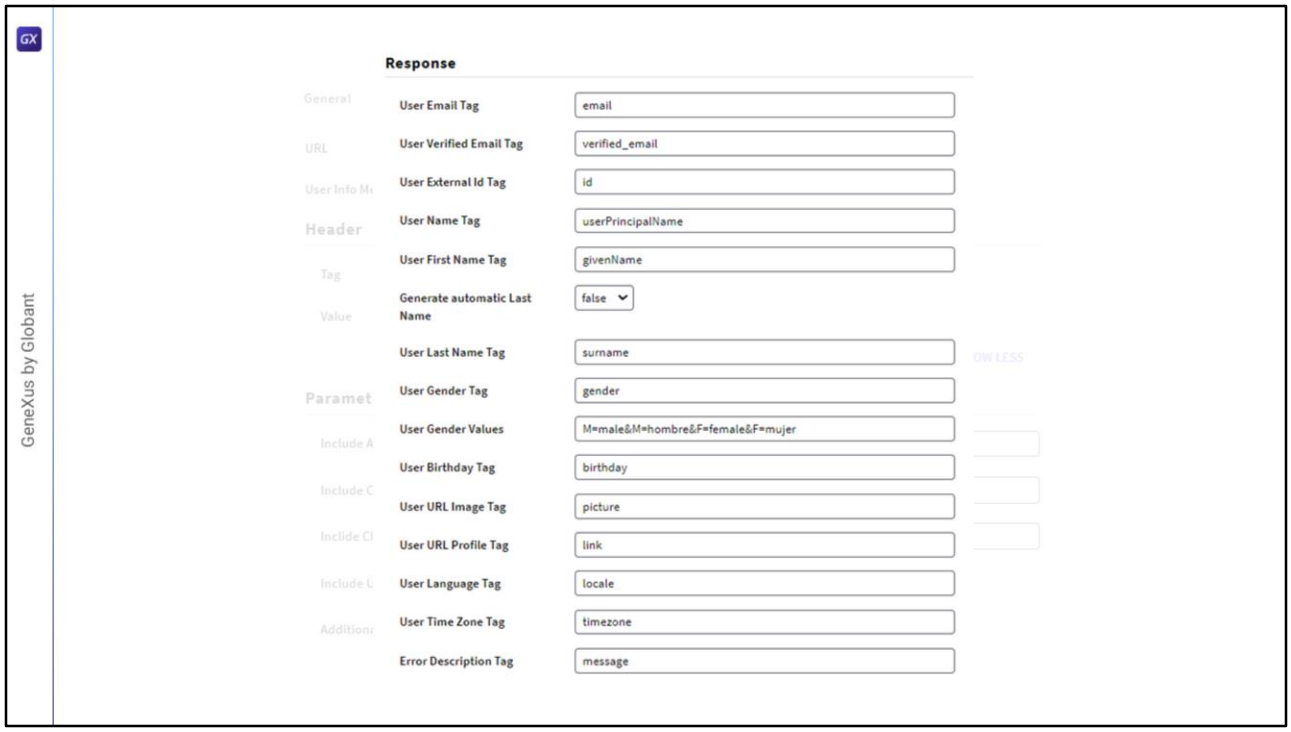


Now, we go to the Authorization tab.
There, we set the Azure URL obtained from its portal, which looks as follows.
Next, we modify the Response type, which must contain the URL shown on the screen.

The rest is left with the default values.

Now we have the Token tab.
Once again, we set the Azure URL obtained from its portal, and leave the rest by default, except for the Grant Type and Additional Parameters fields, which we set with what is shown on the screen.

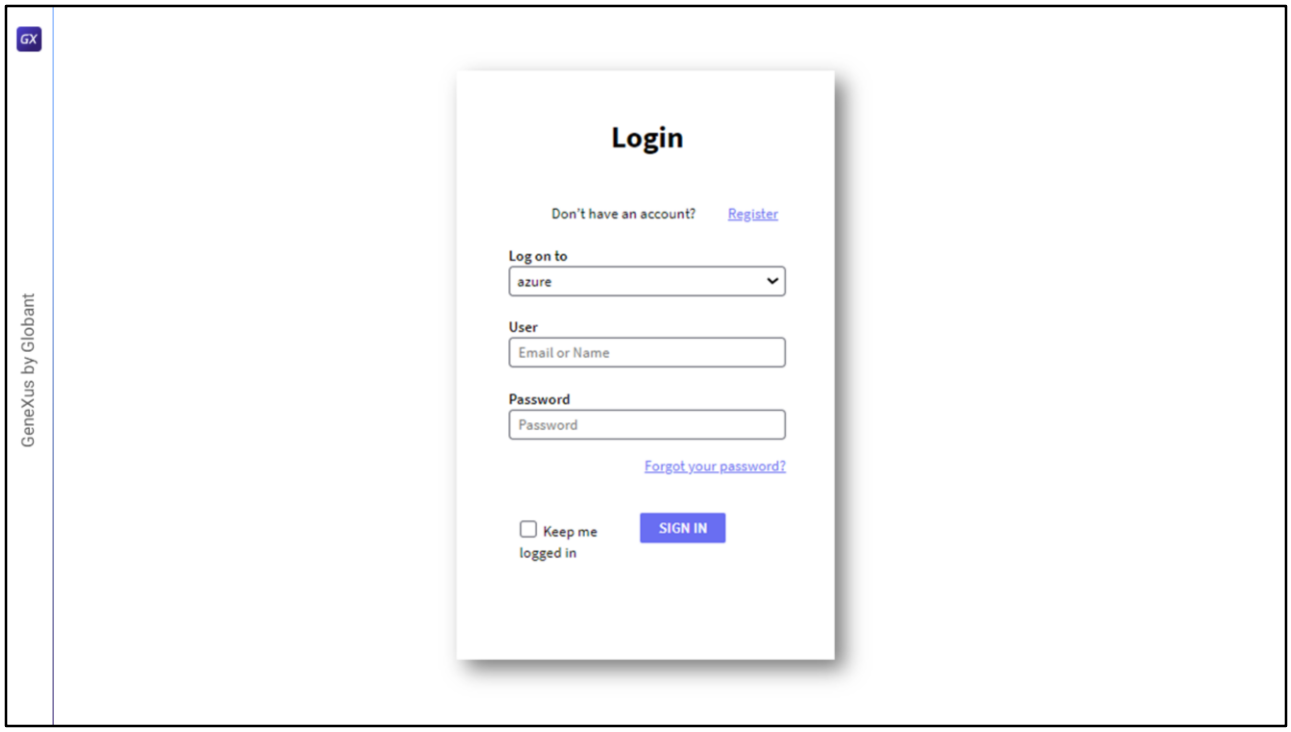
Note that this should only be changed when we don't want to redirect at login time and we want it to be done from the GAM login.
Otherwise, the Grant Type must be left with the default value (which is *authorization_code*) and without additional parameters.



Lastly, in the User Information tab we set the URL shown on the screen (also obtained from Azure) and do not include anything. The Access Token is included by default.
The Response fields should have the following values.

This is how the user is created in the local GAM, and from where the user information is mapped according to the configured IDP.
The IDP must return a unique user identifier, which must be set to "User External Id Tag" to ensure that in subsequent GAM logins the same user is being authenticated.

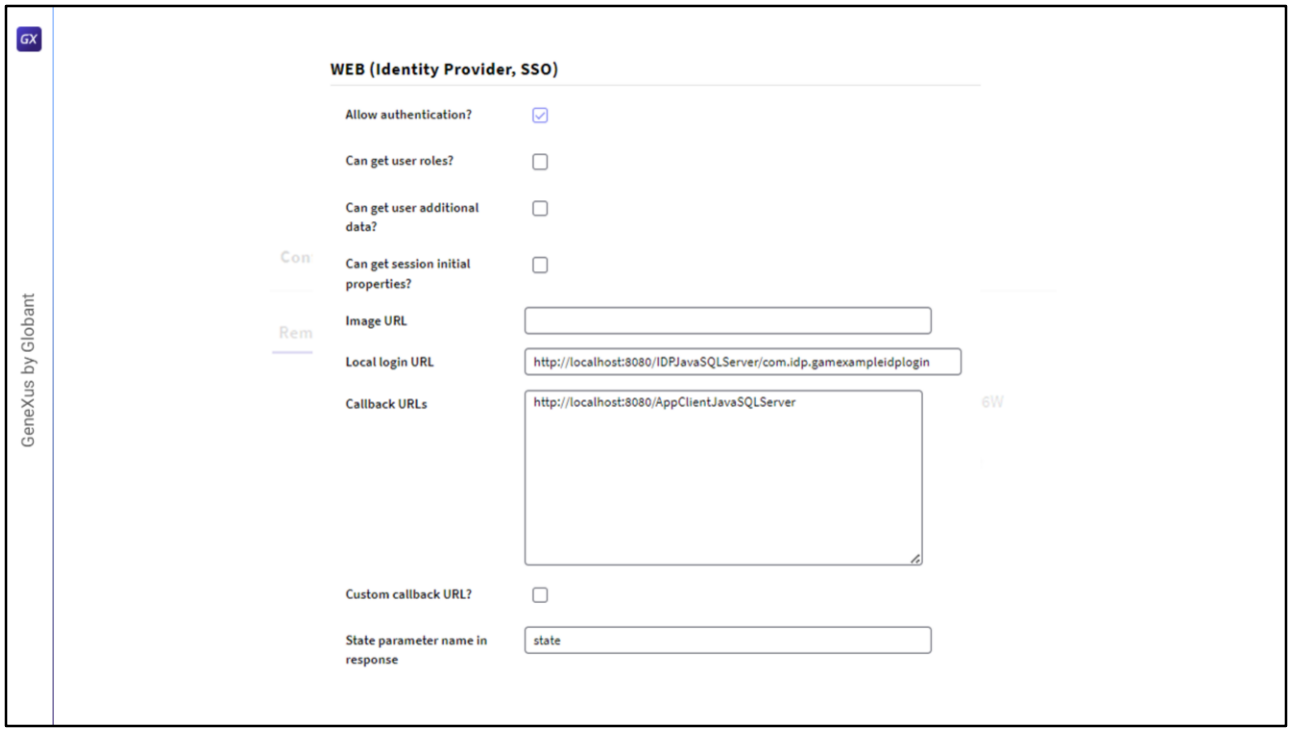
The configuration is now complete.



Now, we go to Login, select the authentication type we have just created, and enter the credentials of a user defined in Azure.

That's all.

Second demo: IDP



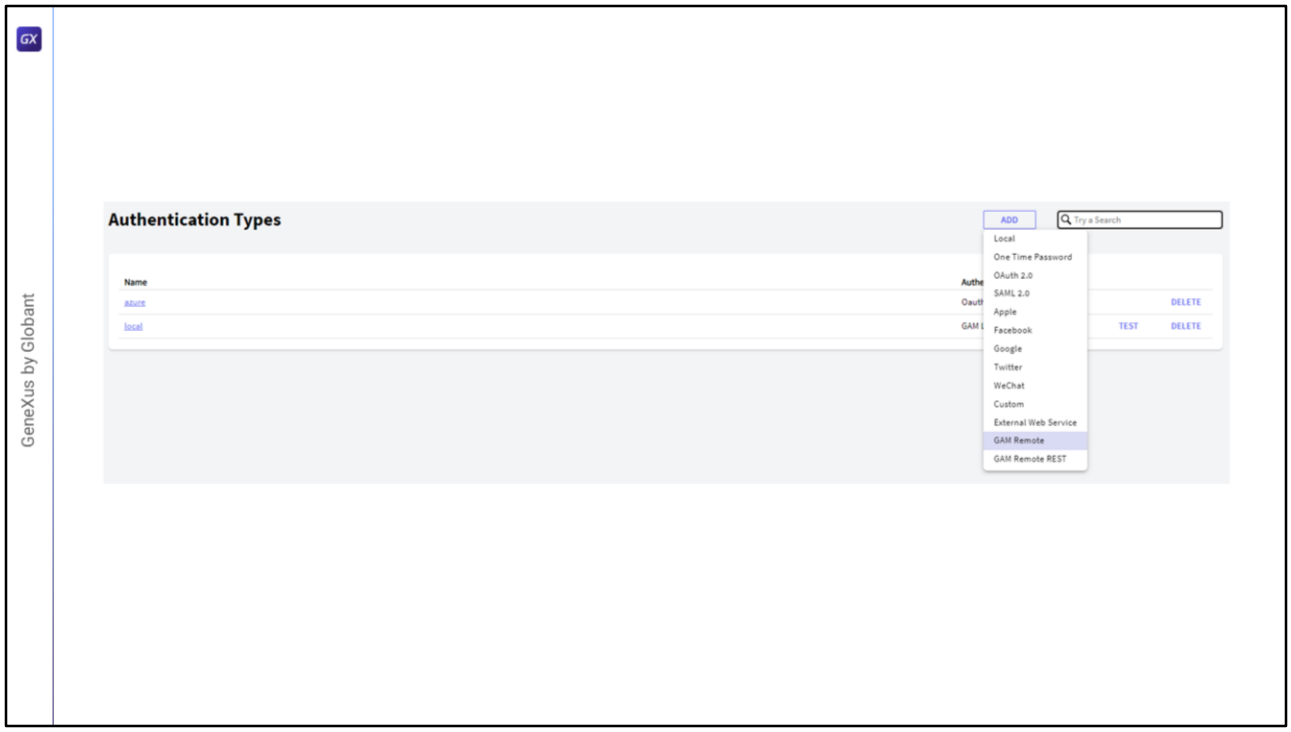
For this demo, we will use the OAuth 2.0 protocol again. The GAM, through it, will be our identity provider.
First, we configure our GAM application defined on the IDP server that will act as provider.

To do so, we select the “Remote Authentication” tab in the application settings from the GAM back end.
We save the Client ID and Client Secret to set them later in the client application.
Next, we select the option to allow authentication in the WEB section (Identity Provider, SSO). There, you can indicate if you want to share with the Client the users' roles, additional information, etc.

It is important to show the local login and callback URLs in the demo.
The first one must correspond to the login URL of the server application. In this case, we will use the example web panel provided by GAM called **GAMExampleIDPLogin**, which will perform the login process in the IDP. It is worth mentioning that in versions prior to GeneXus 18, the GAMRemoteLogin Web Panel is used instead of the GAMExampleIDPLogin used in the demo.
The second one must be the path of the client application from where the IDP will be invoked with a call after the login process is completed. This last parameter can be composed of more than one URL, which must be separated by semicolons.

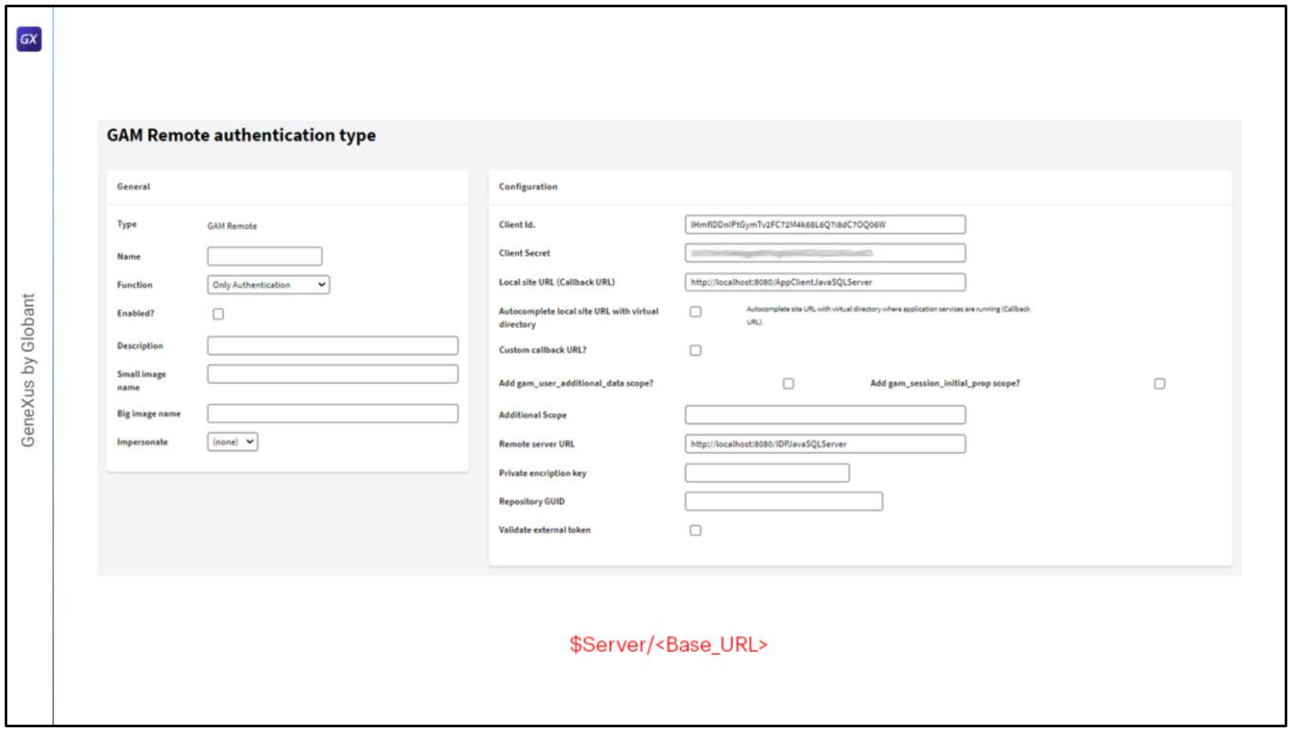
Of course, GAM is where the users that will be used to log in to the IDP must be defined.

With this configuration and a user created, we have finished the process from the IDP side.



Let's look at the Client side.

The first step is to go to Authentication Types from the GAM back-end menu, and create a GAM Remote type.



It is important to configure the following:

Set the Function property to Only Authentication since on the IDP server side we do not indicate that the user roles are shared. If the other option (Authentication and roles) is set, we will get an error when logging in.

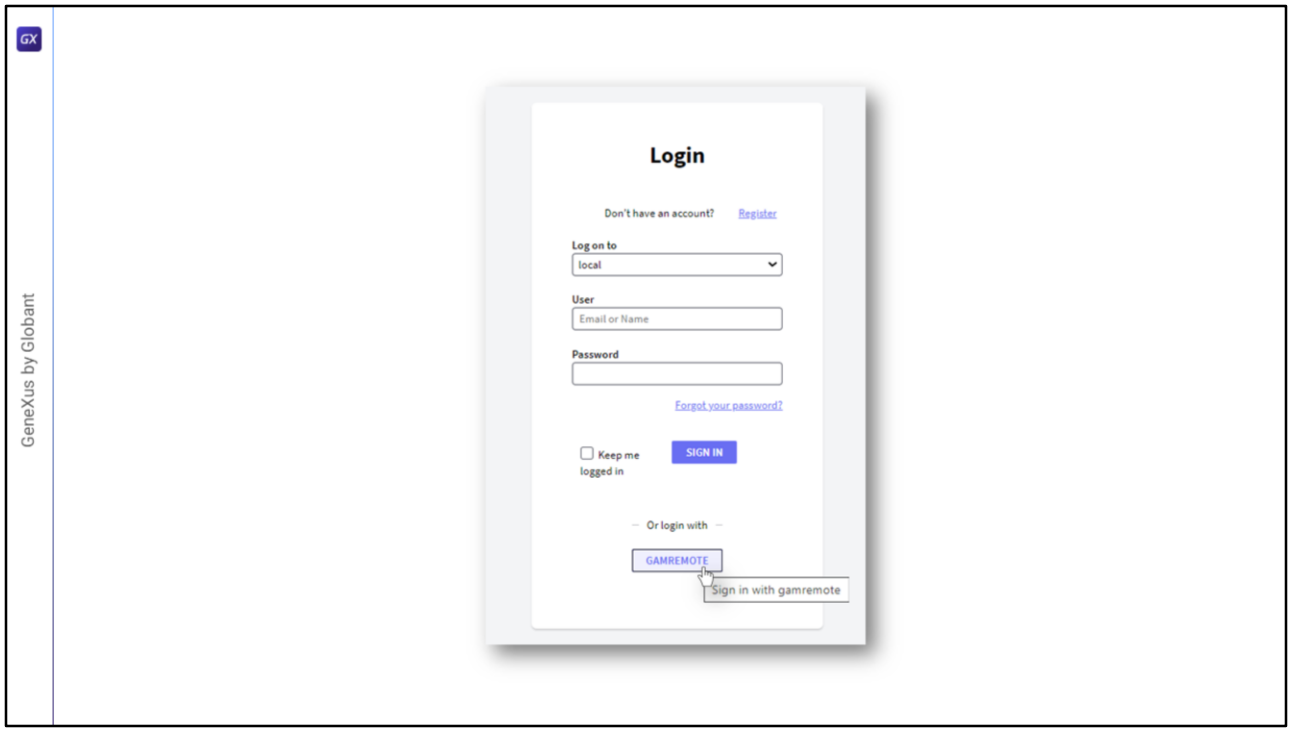
The next thing to configure is the Client Id and Client Secret saved from the IDP.

Later on, we will configure the “Local site URL” property with the address of our client application, the same that we already specified in the Callback URL in the server; also, the “Remote server URL” property with the IDP address, following the format shown in red.

Additional comments:
The property “Add gam_user_additional_data scope?” must be activated when we want to send additional user data. On the server side, the Allow authentication property must be selected in the Web section (Identity provider, SSO).

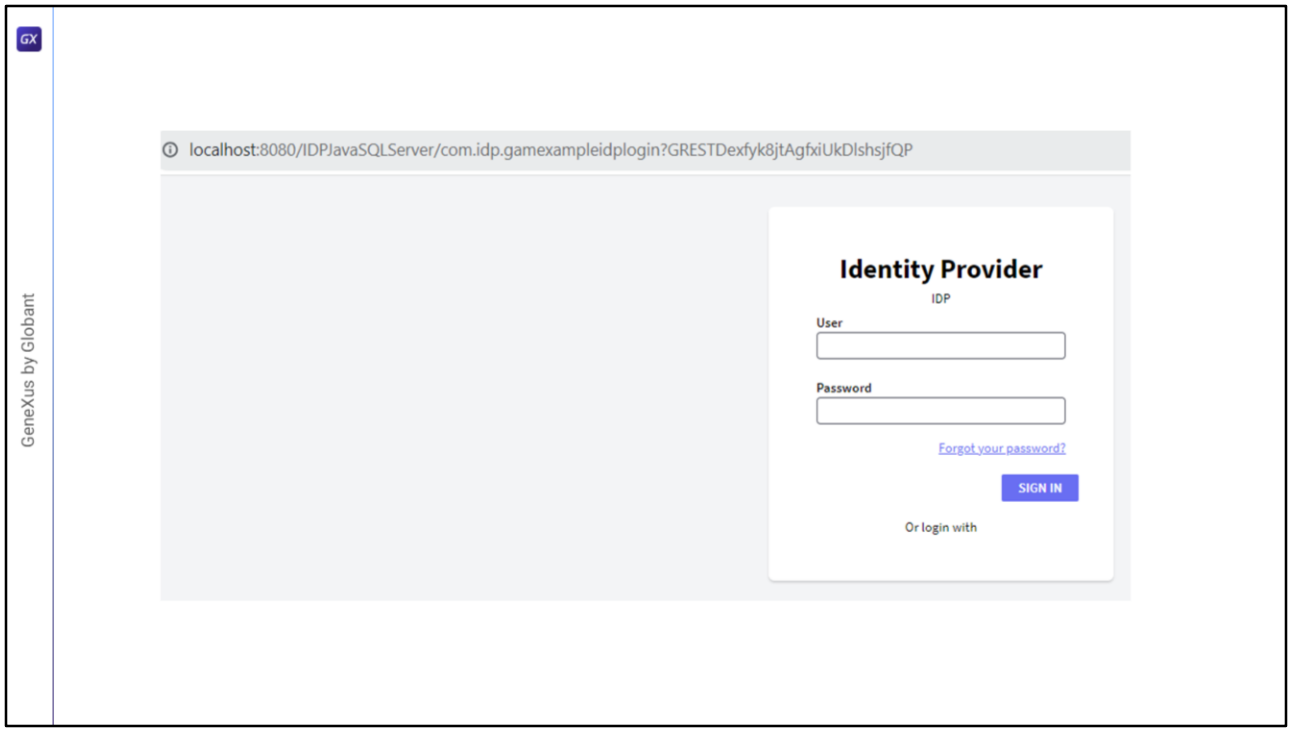
The “Add gam_session_initial_prop scope?” property involves asking the IDP to return the initial properties dynamically set at login to the client. Of course, the IDP must also be configured to send this information.

Finally, the “Validate External Token” property validates the expiration of the session according to the expiration of the token and renews it automatically without having to do it manually.

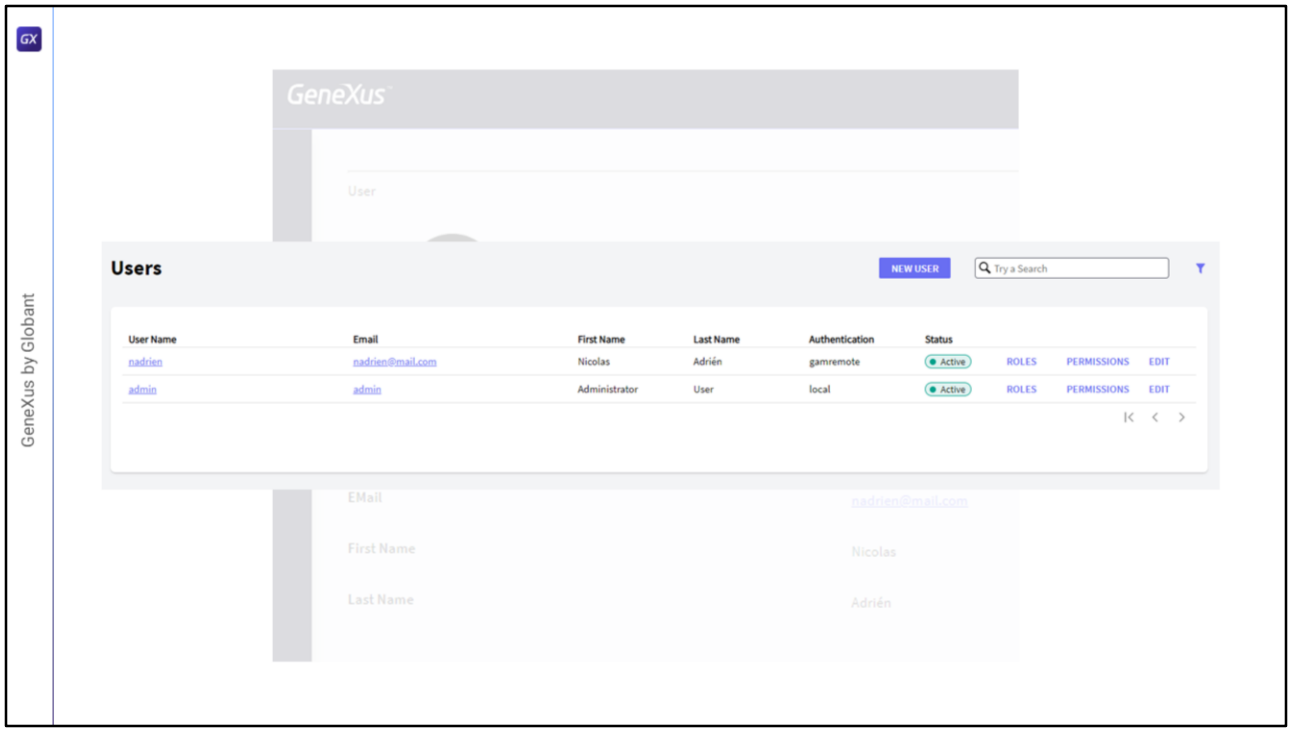


For the purposes of the demo, we create a Web Panel in the Client application, where it shows the data of the logged in user. Of course, this object has integrated security activated with the Authentication value.

When we want to access it, since we are not logged in, we are redirected to the login. Note that since we have the OAuth authentication type defined, from the login we have the option to access through it.

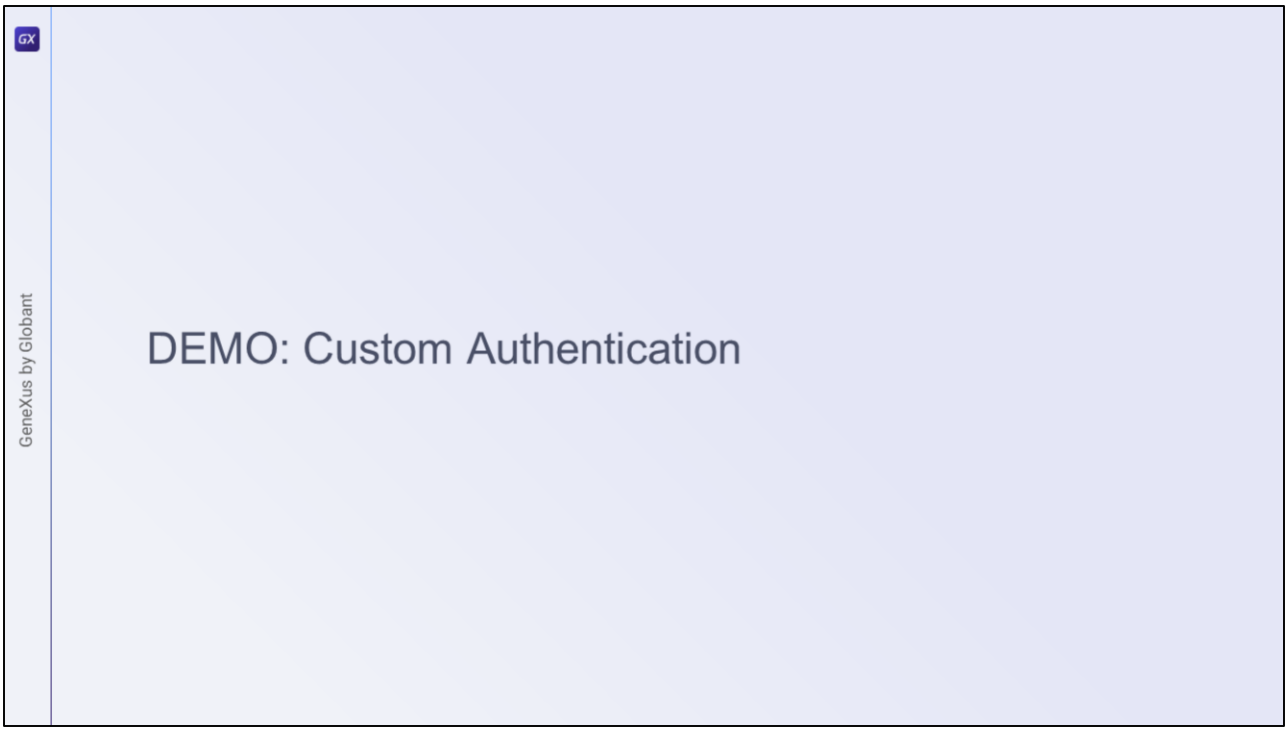


When clicking on this option, we see that it redirects us to the IDP and its remote login. We log in with the user that we had defined in the IDP.



We are now redirected to our Web Panel with the information of the logged in user.

In the back end of the client application, we can see the user we had created in the IDP with its information.



Third demo: Custom Authentication

```
Parm(in:&StrInput, out:&StrOutput); //&StrInput and &StrOutput are varchar(256)


&Key = '03E1E1AAA5BCA19FBA8C42058B4ABF28'
&GAMWSLoginIn.FromJson(&StrInput) // &GAMWSLoginIn is &GAMWSLoginInSDT data type

//Decrypt parameters
&UserLogin = Decrypt64( &GAMWSLoginIn.GAMUsrLogin, &Key )
&UserPassword = Decrypt64( &GAMWSLoginIn.GAMUsrPwd, &Key )
&GAMWSLoginOut = New GAMWSLoginOutSDT() //&GAMWSLoginOut is &GAMWSLoginOutSDT data type
&GAMWSLoginOut.WSVersion = GAMAutExtWebServiceVersions.GAM10
&GAMWSLoginOut.User = New GAMWSLoginOutUserSDT()

Do 'ValidUser'

&StrOutput = &GAMWSLoginOut.ToJson()

Sub 'ValidUser'
    If &UserLogin = !"user"
       If &UserPassword = !"password"
          &GAMWSLoginOut.WSStatus = 1
          &GAMWSLoginOut.User.Code = !"code"
          &GAMWSLoginOut.User.FirstName = !"FirstName"
          &GAMWSLoginOut.User.LastName = !"LastName"
          &GAMWSLoginOut.User.EMail = !"name2@domain.com"
          Do 'GetRoles' //optional
       Else
          &GAMWSLoginOut.WSStatus = 3
       EndIf
    Else
       &GAMWSLoginOut.WSStatus = 2
    EndIf
EndSub

Sub 'GetRoles'
    &GAMWSLoginOutUserRol = New()
    &GAMWSLoginOutUserRol.RoleCode = "role_1"
    &GAMWSLoginOut.User.Roles.Add(&GAMWSLoginOutUserRol)
    &GAMWSLoginOutUserRol = New()
    &GAMWSLoginOutUserRol.RoleCode = "role_2"
    &GAMWSLoginOut.User.Roles.Add(&GAMWSLoginOutUserRol)
EndSub
```

To perform a Custom authentication, we must create a procedure.
In the GeneXus Wiki, you can find the example shown on the screen, with a very simple logic already defined. It is up to the developer to modify it as needed.

First, we see that two Varchar are defined as rules: an input and an output one, which will bring the data entered by the user and return the result of the login with certain user information (if successful, of course).

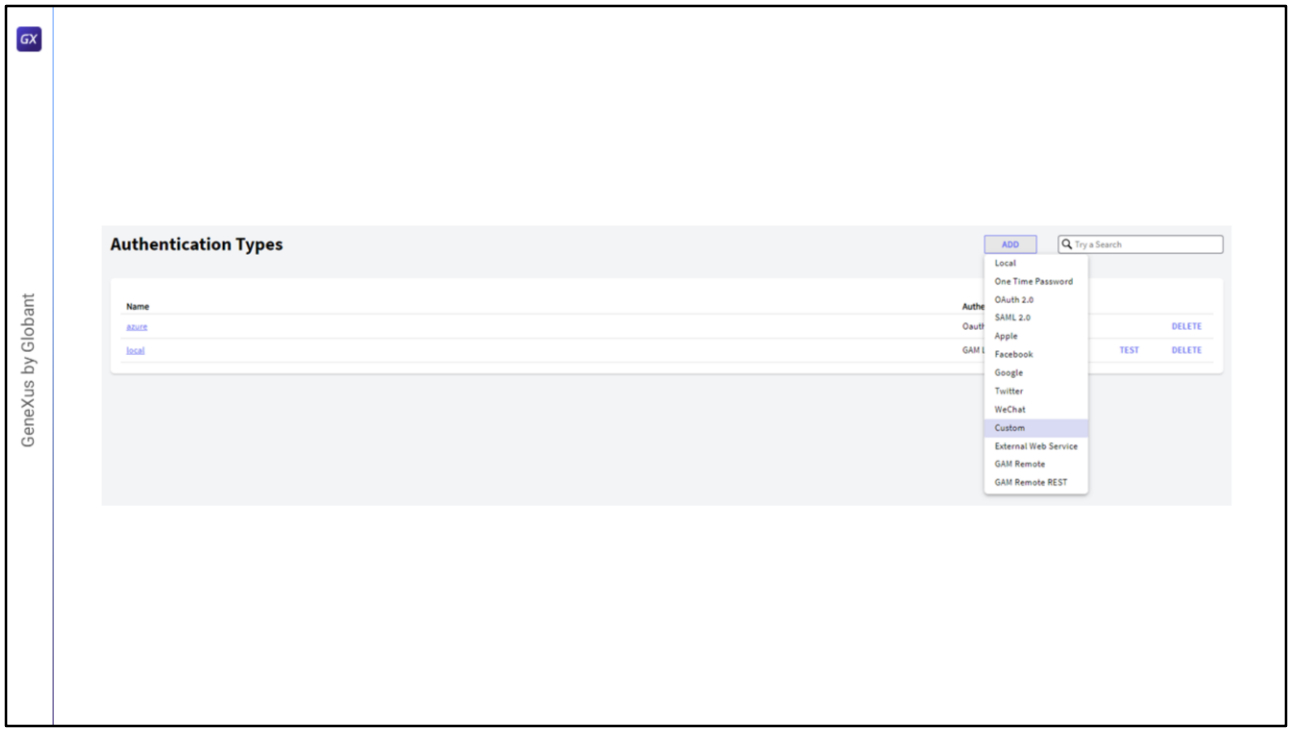
Then, a key is defined that we will explain in detail later on, and the parameters of that input parameter from the rules are decrypted, in addition to creating a data type that will be loaded in the output parameter at the end of the process.

Next, in the ValidUser method the user name and password are validated, in the example, by verifying that the user name is “user” and the password is “password.” Otherwise, different errors are returned depending on the failure.
This method should be changed for a more secure login logic that does not distinguish between errors based on username or password.

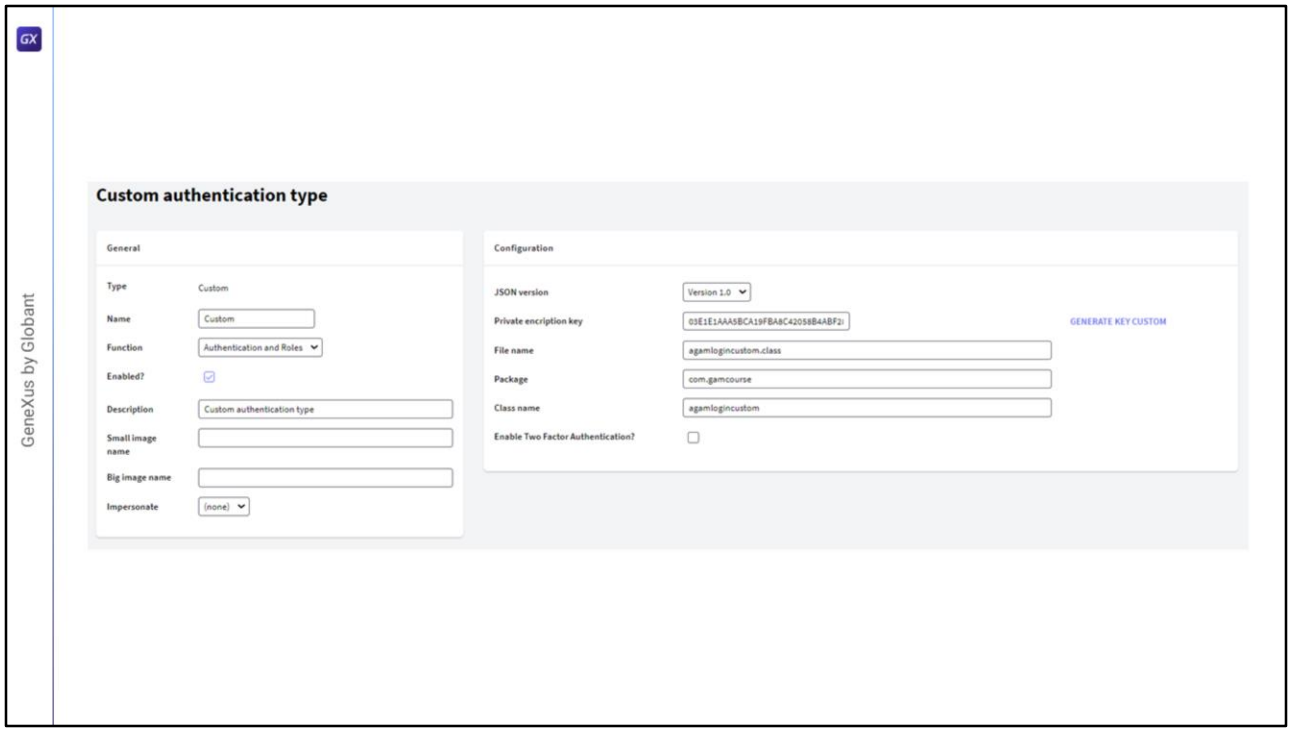
Optionally, the GetRoles method can be used to define certain roles for the logged in user.

This method is useful when we want to program how we validate a user's password,

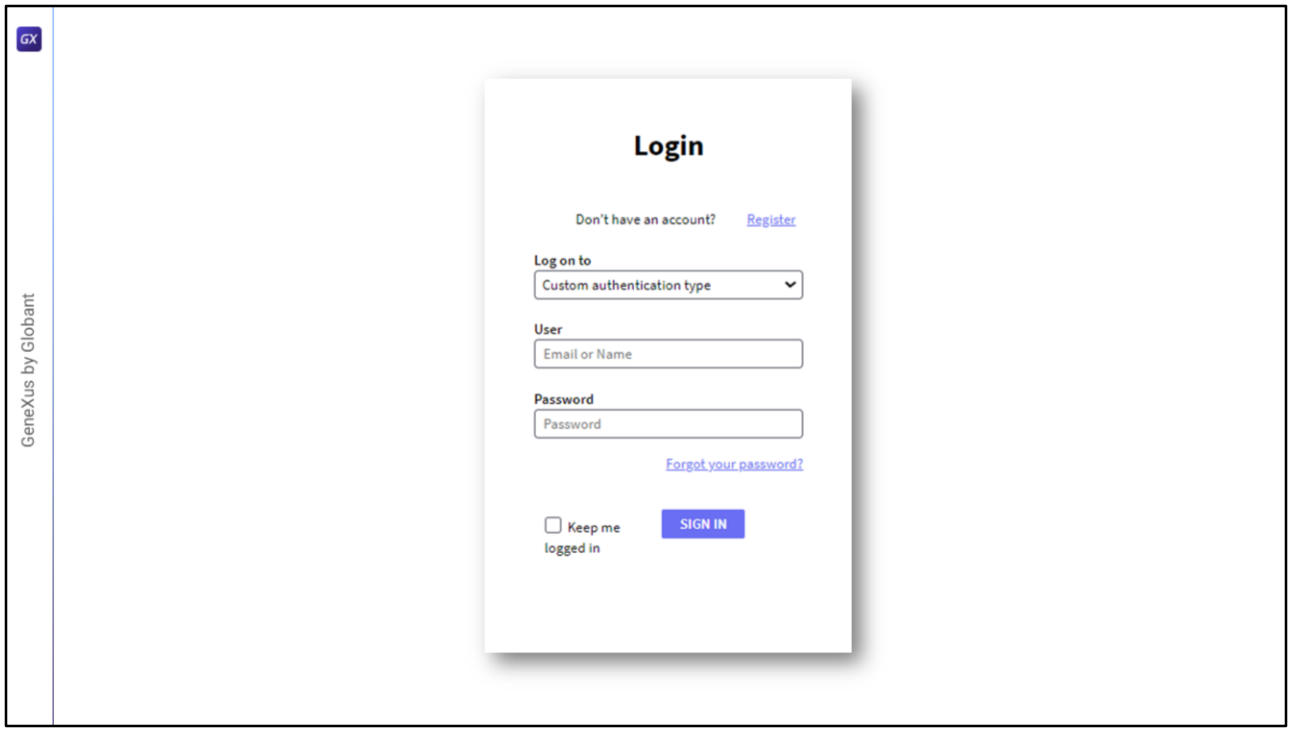
either to validate it against a local database, against an LDAP, or against another place where the user's credentials are stored.



Now that we have a custom authentication procedure, we need to configure it in GAM. The first step is to go to Authentication Types, and create a new one of Custom type.



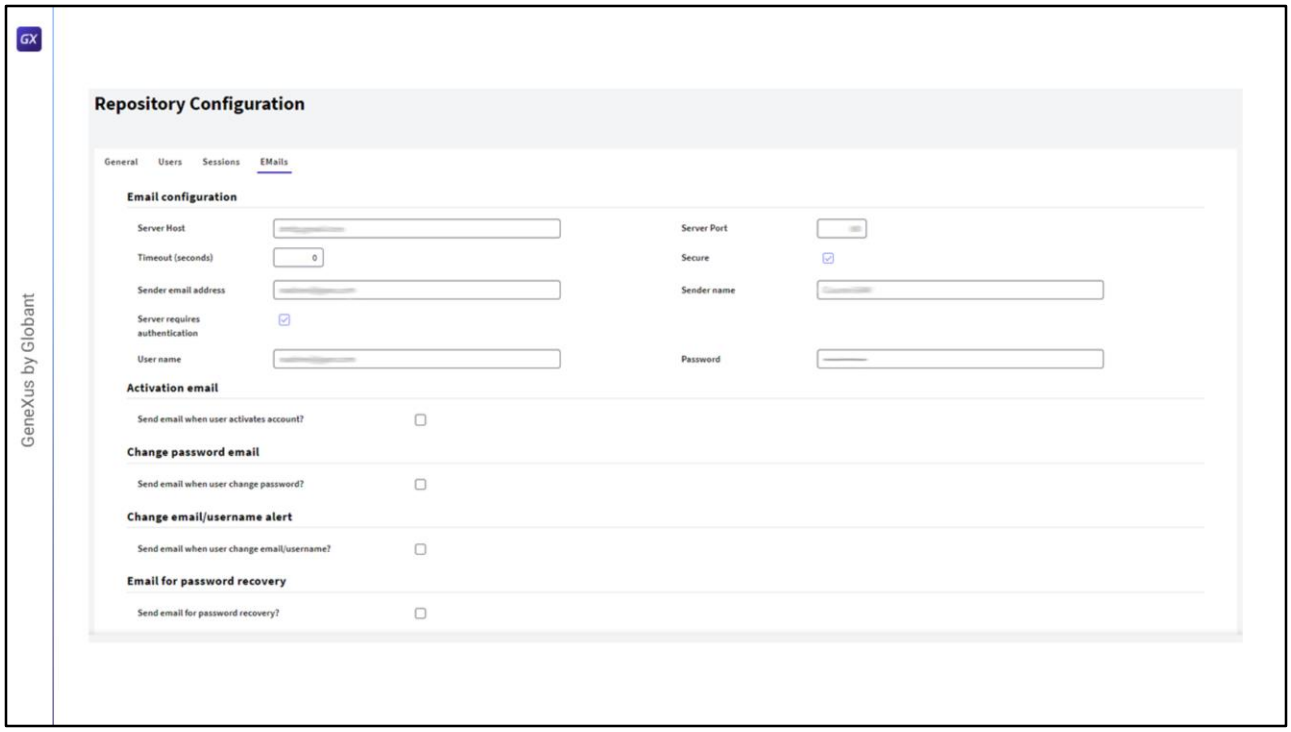
Here, the settings to highlight are as follows:
Function: It allows specifying if we want the authentication type to be Authentication and roles, or only Authentication. In this case, we leave the first option.
Private encryption key: here we must configure the encryption key used in the procedure to decrypt the user and password received. As you may remember, in the slide of the GeneXus procedure that I showed before, a key was defined that we enter in this property. It is useful because the GeneXus encryption function uses it to encrypt the username and password when they are sent to the program.
File name: here we specify the name of the file corresponding to the external procedure. In the case of Java, it is optional.
Package: in the case of Java models, the same Java package name property value is specified here, and in the case of .NET models the value of the application namespace property is specified here. This property is optional and depends on whether the procedure or program used has a package or not.
Lastly, class name, which is a mandatory property that specifies the name of the procedure's class.



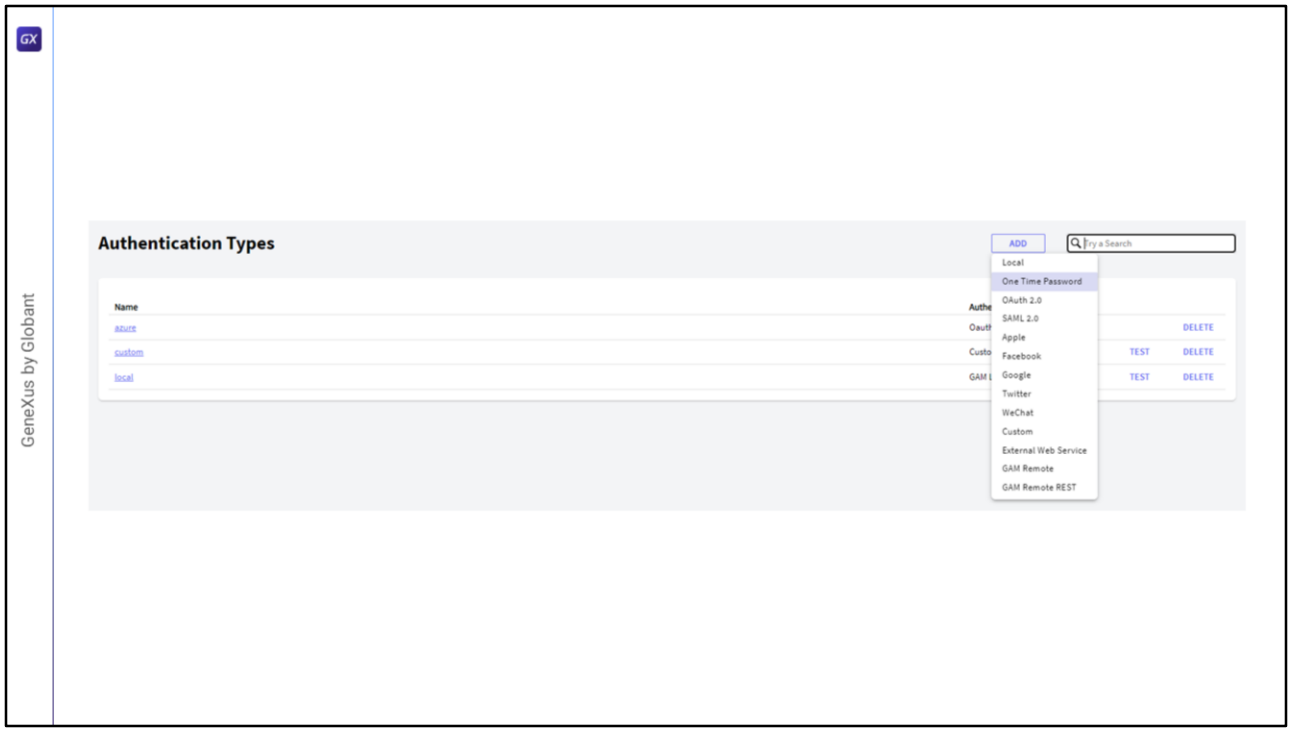
Once everything is configured, we simply set the custom authentication type in our login, and the login is made.

Note that in this case the authentication type is selected through the highlighted combo box because we indicated that it should not be redirected to the IDP. Otherwise, the authentication type is shown as an icon at the bottom of the login screen as we saw in the IDP Demo.

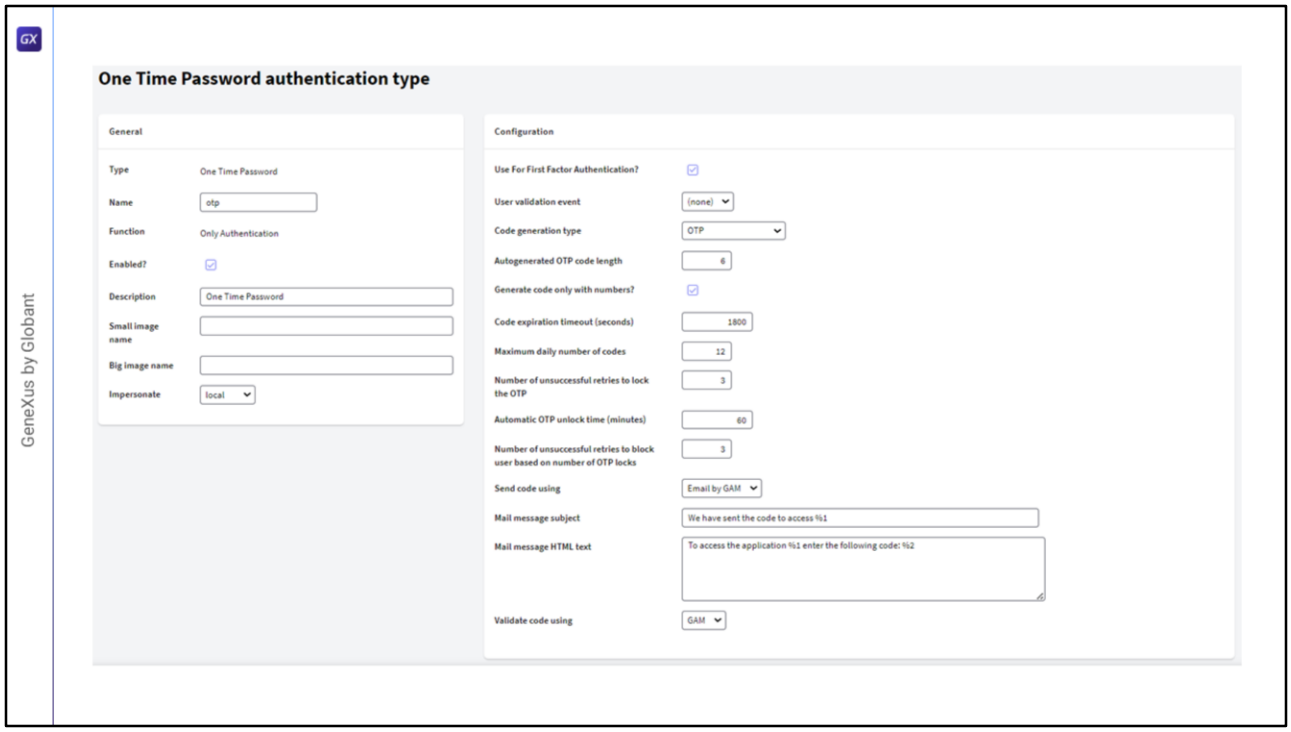
OTP.



A prerequisite to make OTP work is that the repository must have the email service configured to send the codes.
This is configured in the “Repository Configuration” option of the GAM back end.

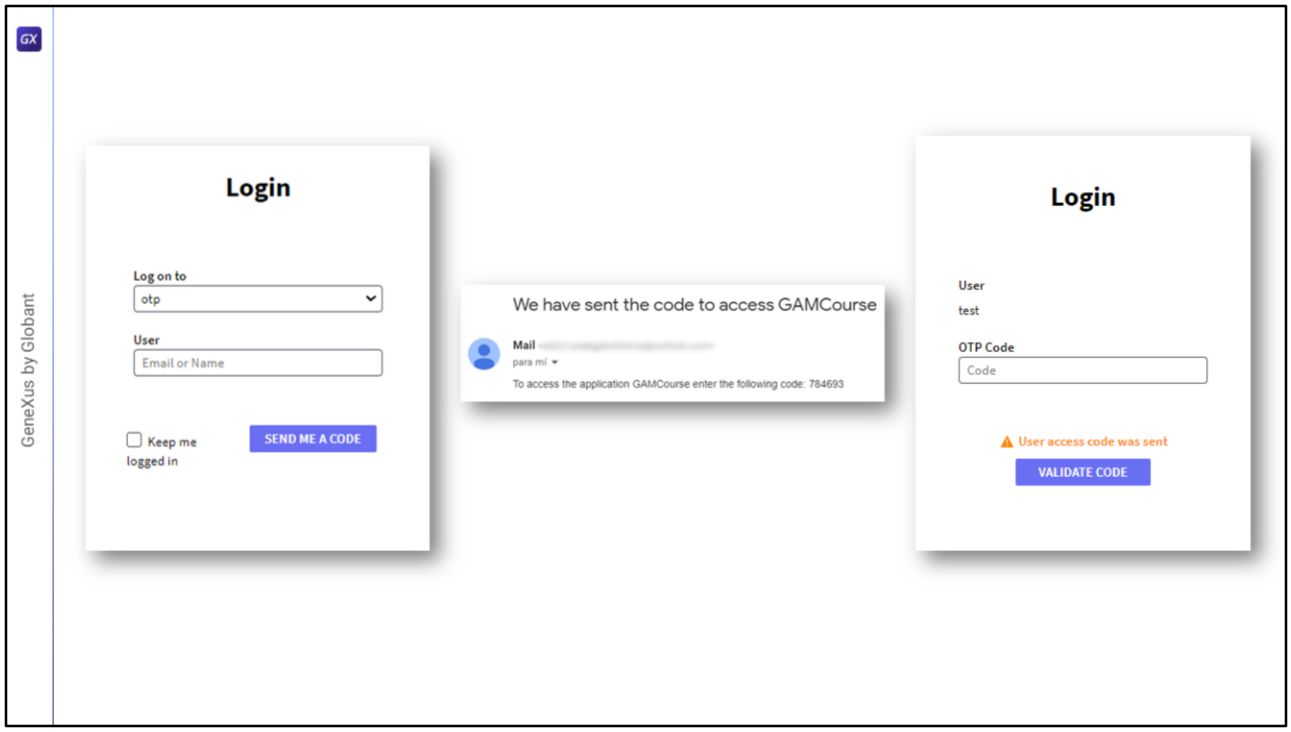


Now, to define this type of authentication, everything is done and configured again through the GAM back end.
As in the previous Demo, we go to Authentication Types and add a new type.
In this case, we select the One Time Password type.

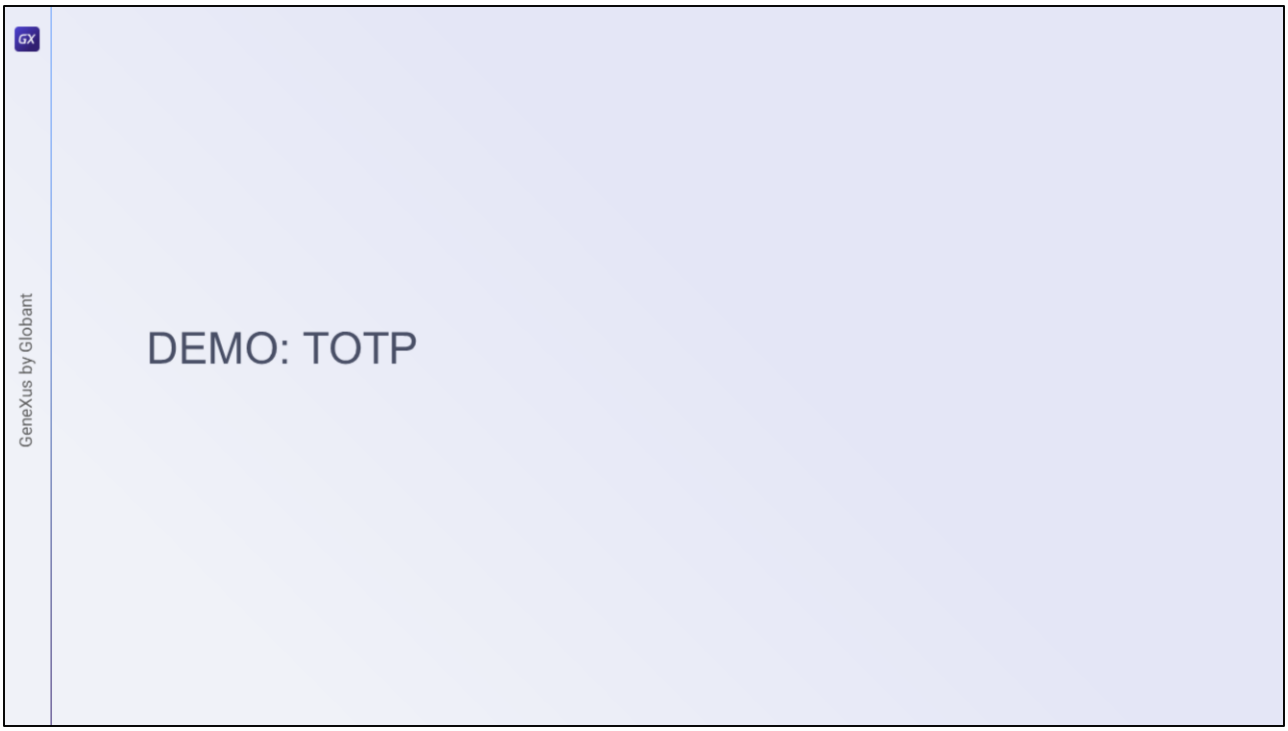


Let's describe the most important properties:
Impersonate: Here we specify the type of authentication where users are going to be validated when using OTP. As I mentioned earlier in the theory section, the users must already exist. This is the only authentication type that requires configuring this property since the users must already exist in the GAM database.
Use as first factor authentication: If you don't configure this property, OTP could only be used as a second factor. In this case, we enable it.

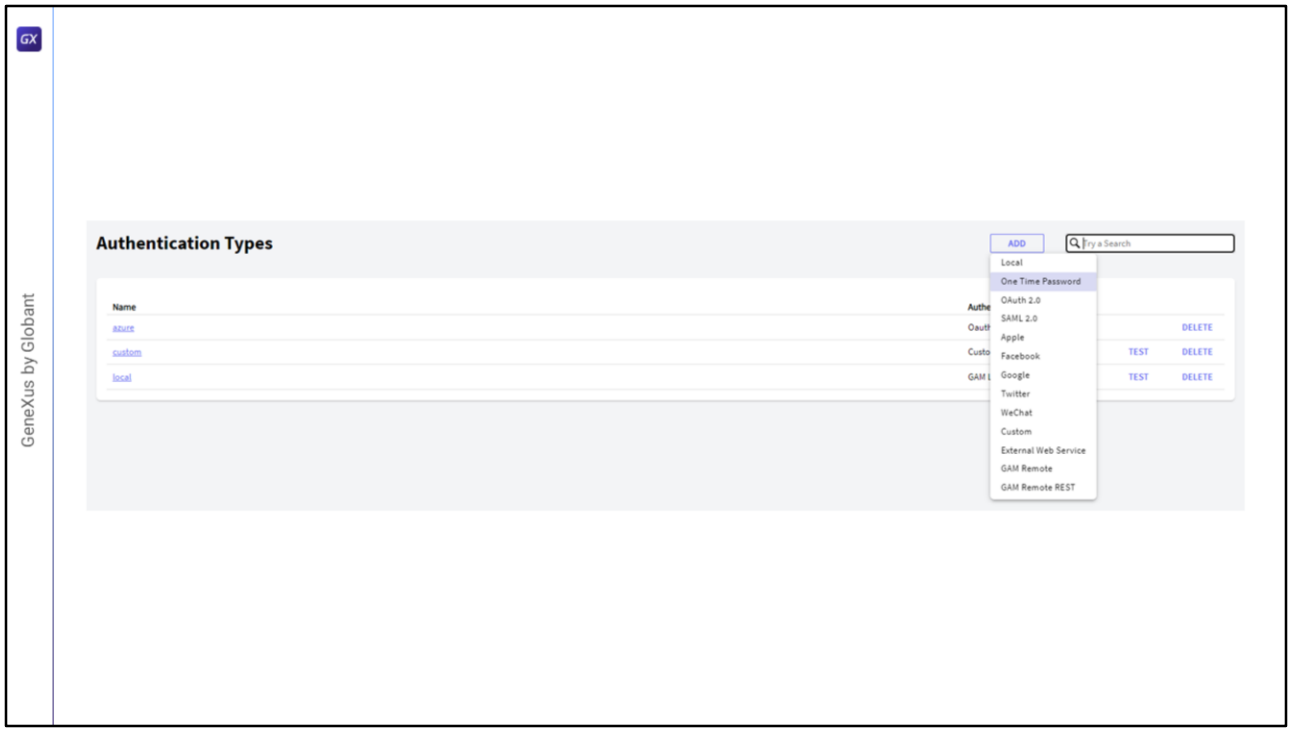
The rest of the properties are used to define properties of the code to send and the email format.
In this case, we will use GAM's default format, which is email, but remember that it is possible to send the code via SMS. If the developer chooses this second way, he/she must implement and configure the GAM event that must be selected in the "Send code using" property.



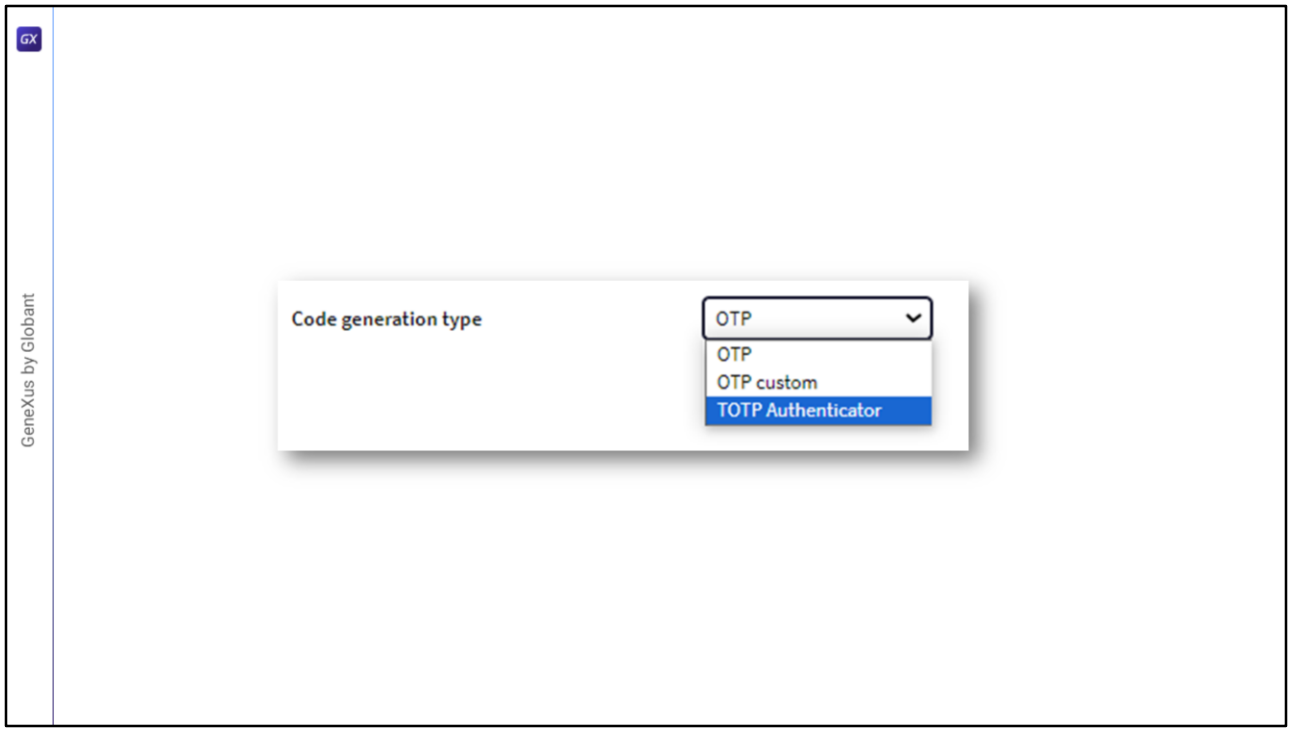
Finally in the Login, we select the OTP type, where we can see that we will only be asked for the user name to send the code.
After receiving the code via email, simply log in to authenticate in the system.



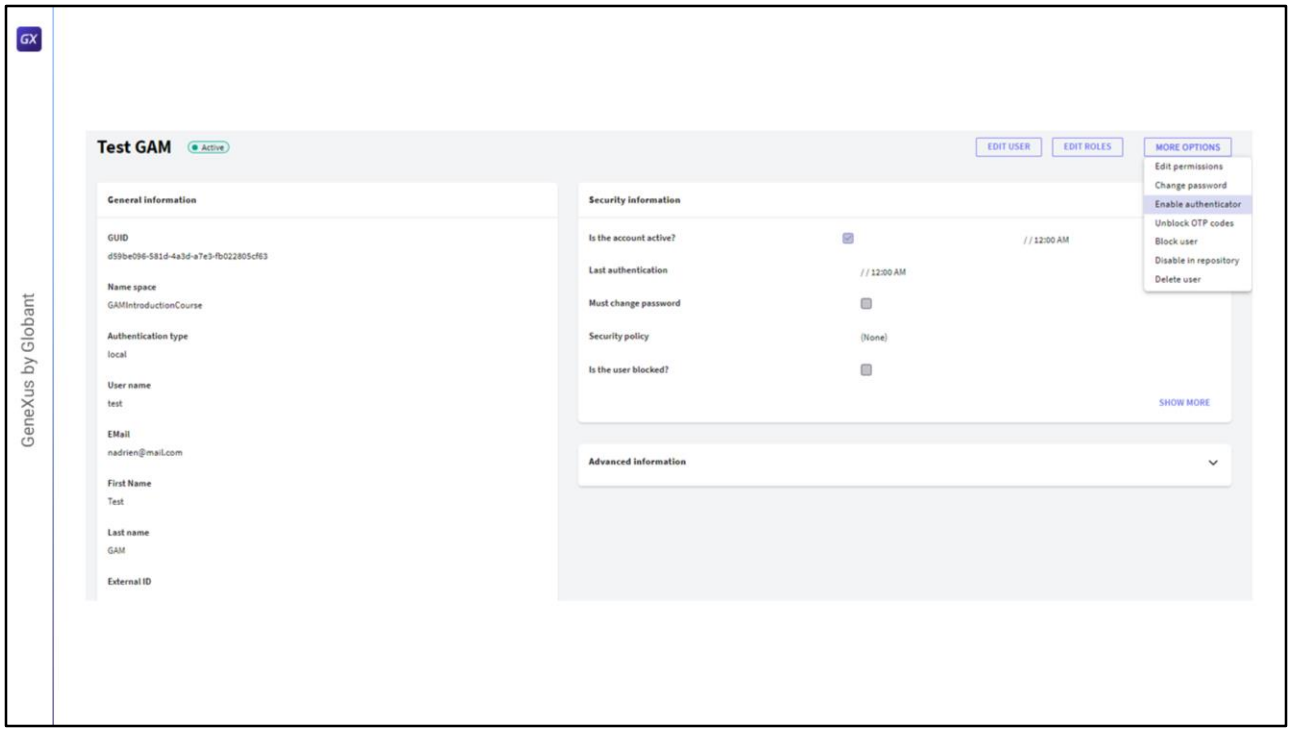
In this demo, the steps to configure a new TOTP authentication type are almost the same as OTP, except for one difference.



To define this type of authentication, we go again to Authentication Types and add a new type.
In this case, we also select One Time Password type.



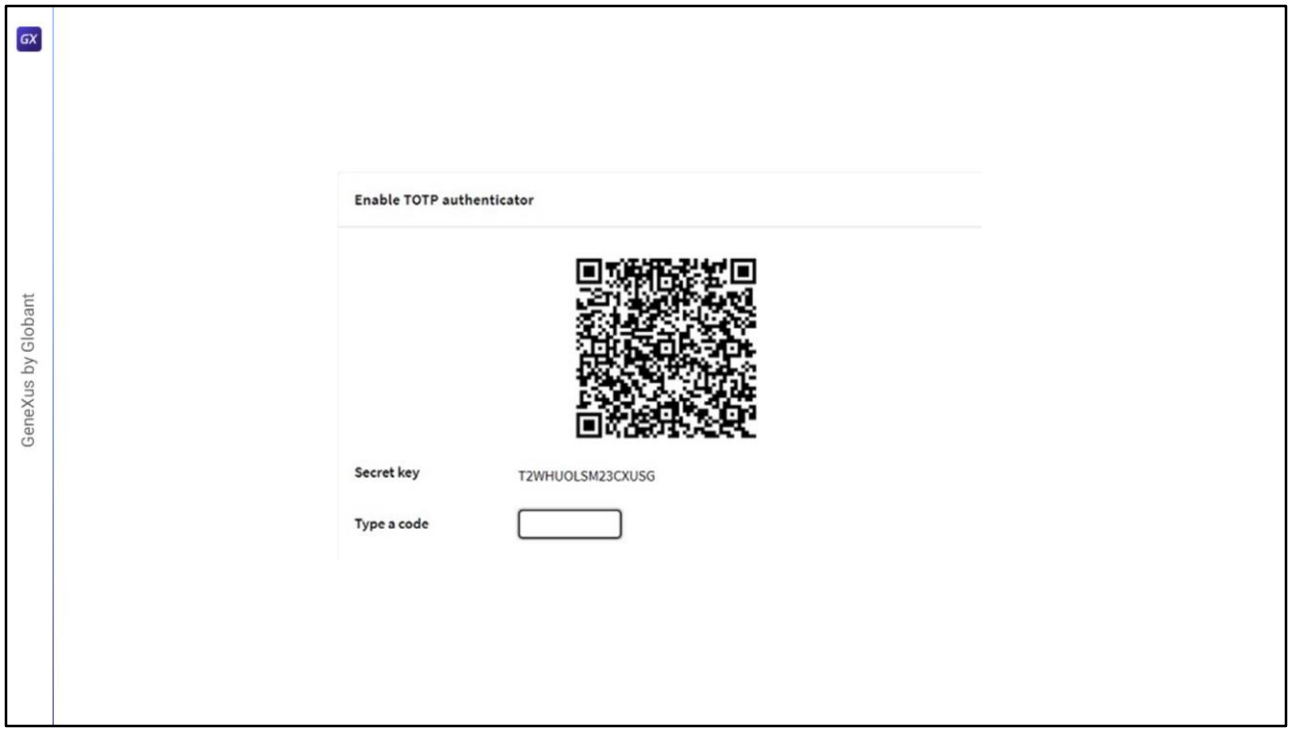
The difference with OTP is the property shown on the screen, where in this case we choose TOTP Authenticator.
The rest of the properties are for code configurations that are not relevant here.



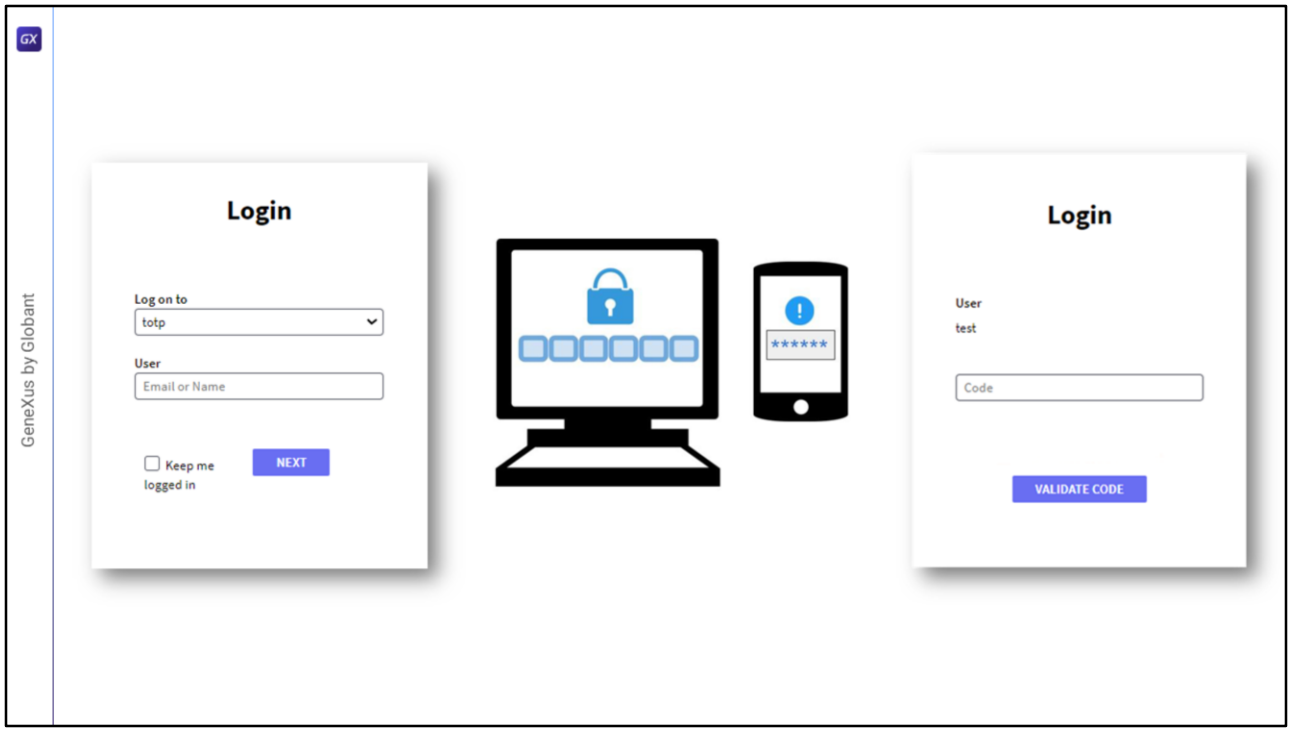
Let's see the most important caveat about OTP.

Each user must enable authentication through their settings. The most important difference is that this code algorithm is time-based and the codes are generated by the different authenticator applications.

For the purposes of the demo, it was created using the administrator user of the GAM back end for a “test” user of a sample application.
The steps to be followed consist of going to the user in question and clicking on Enable authenticator.



Once there, the QR code will be provided to configure in a software system or mobile application based on one-time password authentication. After reading it, it will return the code to enter in the “Type a code” field.



Lastly, the login is the same as in all the previous types, and in this case there is an intermediary application that provides the access code.


GeneXus by Globant
GeneXus™
by Globant
training.genexus.com