Article Elliott Grey · Mar 7, 2023 9m read

Creating a REST API with JWT Authentication in ObjectScript

Foreword

InterSystems IRIS versions 2022.2 and newer feature the ability to authenticate to a REST API using JSON web tokens (JWTs). This feature enhances security by limiting where and how often passwords transfer over the network in addition to setting an expiration time on access.

The goal of this article is to serve as a tutorial on how to implement a mock REST API using InterSystems IRIS and lock access to it behind JWTs.

NOTE I am NOT a developer. I make no claims as to the efficiency, scalability, or quality of the code samples I use in this article. These examples are for educational purposes ONLY. They are NOT intended for production code.

Prologue

With that disclaimer out of the way, let's explore the concepts we're going to be dissecting here.

What is REST?

REST is an acronym for REpresentational State Transfer. It is an architecture for programs to communicate with web applications and access functions those applications have published.

What is a JWT?

A JSON web token (JWT) is a compact, URL-safe means of representing claims transferred between two parties that can be digitally signed, encrypted, or both. If you want to learn more about JWTs and other JSON web classes InterSystems IRIS supports, read <u>this post</u>.

Getting Our Hands Dirty

According to the Spec

To consume a REST API, we first need to have a REST API. I've provided a <u>sample OpenAPI 2.0 specification</u> here that's Table Top Role Playing Game (TTRPG) flavored. It's the one I'll be using throughout the examples here. There are plenty of examples of how to write your own online so feel free to dive into that, but the specification is just a blueprint. It doesn't do anything other than inform us how to use the API.

REST API Generation

InterSystems IRIS provides a very neat way of generating REST API code stubs. <u>This documentation</u> provides a complete way of generating the code stubs. Feel free to use the OpenAPI 2.0 specification I provided in the previous section here.

Implementation

Here's where we're going to dig deep. The generation section will have created three .cls files for you:

- 1. impl.cls
- 2. disp.cls
- 3. spec.cls

We are going to spend the bulk of our time in impl.cls, maybe touch disp.cls for debugging, and leave spec.cls alone.

In impl.cls are code stubs for the methods disp.cls will call when it receives an API request. The OpenAPI specification defined these signatures. It can tell what you want it to do, but you ultimately need to implement it. So let's do that!

Creation

One of the ways we use a database is adding objects to it. These objects serve as the foundation for our other functions. Without any existing objects, we won't have anything to view so we're going to start with our object model: a Character!

A Character will necessarily have a name and optionally specify their class, race, and level. Below is an example implementation of the TTRPG.Character class

```
Class TTRPG.Character Extends %Persistent
{
  Property Name As %String [ Required ];
  Property Race As %String;
  Property Class As %String;
  Property Level As %String;
  Index IndexName On Name [ IdKey ];
  ClassMethod GetCharByName(name As %String) As TTRPG.Character
  {
    set character = ##class(TTRPG.Character).%OpenId(name)
    Quit character
  }
}
```

Since we want to store Character objects in the database, we need to inherit the %Persistent class. We want to be able to look up our characters by name as opposed to assigning an arbitrary ID key to them so we set the [IdKey] attribute on the Index for the Character.Name property. This also guarantees uniqueness of the character name.

With our foundational object model defined, we can dissect the REST API implementation. The first method we'll explore is the PostCharacter method.

As an overview, this part consumes an HTTP POST request to the /characters endpoint with our defined character properties in the body. It should take the provided arguments and create a TTRPG.Character object out of them, save it to the database, and let us know whether it succeeded or not.

```
ClassMethod PostCharacter(name As %String, class As %String, race As %String, level A
s %String) As %DynamicObject
{
    set results = {} // create the return %DynamicObject
    //create the character object
    set char = ##class(TTRPG.Character).%New()
    set char.Name = name
    set char.Class = class
    set char.Race = race
    set char.Level = level
    set st = char.%Save()
    if st {
        set charInfo = {}
        set charInfo.Name = char.Name
        set charInfo.Class = char.Class
        set charInfo.Race = char.Race
        set charInfo.Level = char.Level
        set results.Character = charInfo
        Set results.Status = "success"
    }
    else {
        Set results.Status = "error"
        Set results.Message = "Unable to create the character"
    }
    Quit results
}
```

Now that we can create characters, how do we retrieve the one we just made? According to the OpenAPI specification, the /characters/{charName} endpoint allows us to retrieve a character by name. We retrieve the character instance, if it exists. If it doesn't exist, we return an error letting the user know that a character with the provided name doesn't exist. This is implemented in the GetCharacterByName method.

```
ClassMethod GetCharacterByName(charName As %String) As %DynamicObject
{
   // Create a new dynamic object to store the results
        Set results = {}
        set char = ##class(TTRPG.Character).GetCharByName(charName)
        if char {
           set charInfo = {}
            set charInfo.Name = char.Name
            set charInfo.Class = char.Class
            set charInfo.Race = char.Race
            set charInfo.Level = char.Level
            set results.Character = charInfo
            Set results.Status = "success"
        }
        // If no character was found, set an error message in the results object
        else {
            Set results.Status = "error"
            Set results.Message = "No characters found"
```

```
}
// Return the results object
Quit results
```

}

But that's just your character. What about all the other characters that other people have made? We can view these characters using the GetCharacterList method. It consumes an HTTP GET request to the /characters endpoint to compile a list of all characters in the database and returns that list.

```
ClassMethod GetCharacterList() As %DynamicObject
{
    // Create a new dynamic object to store the results
        Set results = {}
        set query = "SELECT Name, Class, Race, ""Level"" FROM TTRPG.""Character"""
        set tStatement = ##class(%SQL.Statement).%New()
        set qstatus = tStatement.%Prepare(query)
        if qstatus '= 1 { Do ##class(TTRPG.impl).%WriteResponse("Error: " _ $SYSTEM.S
tatus.DisplayError(qstatus)) }
        set rset = tStatement.%Execute()
        Set characterList = []
        while rset.%Next(){
            Set characterInfo = {}
            Set characterInfo.Name = rset.Name
            set characterInfo.Race = rset.Race
            Set characterInfo.Class = rset.Class
            Set characterInfo.Level = rset.Level
            Do characterList.%Push(characterInfo)
        }
        if (rset.%SQLCODE < 0) {write "%Next failed:", !, "SQLCODE ", rset.%SQLCODE,
": ", rset.%Message quit}
        set totalCount = rset.%ROWCOUNT
            // Set the status, totalCount, and characterList properties in the result
s object
            Set results.Status = "success"
            Set results.TotalCount = totalCount
            Set results.CharacterList = characterList
        // Return the results object
        Quit results
}
```

And that's our API! The current specification does not provide a way to update or delete characters from the database, and that's left as an exercise to the reader!

IRIS Configuration

Now that we have our REST API implemented, how do we get it to communicate with IRIS? In the Management Portal, if you go to System Administration > Security > Applications > Web Applications page, you can create a new web application. The name of the application is the endpoint you'll use when making requests. For example, if you

named it /api/TTRPG/, requests for the API will go to http://{IRISServer}:{host}/api/TTRPG/{endpoint}. For a local default normal-security install of IRIS, this looks like http://localhost:52773/api/TTRPG/{endpoint}. Give it a nice description, set the desired namespace, and click the radio button for REST. To enable JWT authentication, select the "Use JWT Authentication" box. The JWT Access Token Timeout determines how often a user will need to receive a new JWT. If you plan on testing the API for an extended time, I'd recommend making this value an hour (3600 seconds) and the JWT Refresh Token Timeout (the period within which you can renew before your token is expired for good) to be 900 seconds.

System > Security Management > Web Applications > Edit Web Application - (security settings)*	
Edit Web Application	Save Cancel

Edit definition for web application /api/TTRPG:

General	Application Roles Matching Roles	
Name	/api/TTRPG Required. (e.g. /csp/appname)	
Description	REST API for TTRPG characters	
Namespace	e TESTING V Default Application for TESTING: /api/ttrpg	
Enable Application		
Enable	REST Dispatch Class TTRPG.disp	
	Required.	
	Redirect Empty Path	
	JWT Access Token Timeout 3600 JWT Refresh Token Timeout 900	
	○ CSP/ZEN □ Analytics ☑ Inbound Web Services □ Prevent login CSRF attack	
Security Settings	Resource Required Group By ID	
	Allowed Authentication Methods Unauthenticated Password Kerberos Login Cookie	
Session Settings	Session Timeout 900 seconds Event Class .cls	
	Use Cookie for Session Always V Session Cookie Path /api/TTRPG/ V Session Cookie Scope Strict V User Cookie Scope Strict V	

Now that the application is configured, we need to configure IRIS itself to allow for JWT authentication. You can configure this option in System Administration > Security > System Security > Authentication/Web Session Options. At the bottom is the JWT issuer field and the signature algorithm to use for signing and validating the JWTs. The issuer field will appear in the claims section of the JWT and its purpose is to inform who gave you this token. You could set it to "InterSystems".

Testing Time

Everything's configured and implemented, so let's give it a whirl! Load your favorite API request making tool (I'll be using a Firefox extension called RESTer in the examples) and we'll start constructing REST API requests.

First, let's try to list out any characters that exist.

We received a 401 Unauthorized error. This is because we aren't logged in. You might be thinking, Elliott, we didn't implement any login functionality to this REST API. That's ok because InterSystems IRIS handles it for us when we use JWT authentication. It provides four endpoints that we can use to manage our session. These are: /login, /logout /revoke and /refresh. They can be customized in the disp.cls as in the below example:

```
Parameter TokenLoginEndpoint = "mylogin";
Parameter TokenLogoutEndpoint = "mylogout";
Parameter TokenRevokeEndpoint = "myrevoke";
Parameter TokenRefreshEndpoint = "myrefresh";
```

Let's access the /login endpoint now.

The body of this request is not shown for security measures, but it follows this JSON structure:

```
{"user":"{YOURUSER}", "password":"{YOURPASSWORD}"}
```

In return for our password, we receive a JWT! This is the value of "accesstoken". We're going to copy this and use it in our requests going forward so we don't have to transmit our password all the time.

Now that we have a JWT for authentication, let's try creating a character!

We format our request as below:

Using the bearer token as a header in the format of "Authorization: Bearer {JWTValue}". In a curl request, you can write this with -H "Authorization: Bearer {JWTValue}"

Let's create another character for fun, use whichever values you'd like.

Now let's trying listing out all characters that exist in the database.

We get our two characters we made back! But what if we just wanted to access one? Well, we implemented that with the /characters/{charName} endpoint. We can format that request like this:

That's our REST API at work, folks! When done with your session, you can logout at the /logout endpoint using your JWT. This will revoke the JWT and blacklist it so you cannot use it again.

Conclusion

InterSystems IRIS version 2022.2+ features the ability to authenticate to a REST API using JSON web tokens (JWTs). This feature enhances security by limiting password use and setting an expiration date on access to the API.

I hope this primer on generating a REST API and securing it with JWTs through IRIS was helpful. Please let me know if it was! I appreciate any feedback.

#JSON #REST API #Security #Tutorial #InterSystems IRIS

Source URL: https://community.intersystems.com/post/creating-rest-api-jwt-authentication-objectscript