Article
David Hockenbroch · Feb 19, 2024 7m read

Reserving a License

" I have been waiting for thirty seconds for service. This is outrageous! I am leaving

" I am very sorry to hear that, sir. Perhaps, next time, you should make a reservation. "

If you heard that comment at your favorite restaurant, you would think the person saying it was being ridiculous. However, in the context of your API, it makes perfect sense. Just like your favorite eatery, your API has some regular patrons who, as you know, will be visiting one day or another. It would be great to be able to make a standing reservation for them as well.

It will involve a couple of IRIS fundamentals. First, we will have to gain an understanding of the %SYSTEM.License interface. This is an interface provided to obtain information about the license usage for an instance of IRIS. Second, we will have to learn about the %CSP.SessionEvents class. This class allows us to override various methods that are called throughout the lifecycle of a CSP session.

We will start with the %SYSTEM.License interface. Whereas some of the methods in this class can be called using $SYSTEM.License, others must be called via ##class(%SYSTEM.License). For the sake of consistency, we will use ##class(%SYSTEM.License) since it can be used for all methods. We will need to be able to check the number of licenses available and those used by a specific user for our use case.

First, we will check the number of available license units. If at least one license is available, we do not need to check anything else because we already have enough room for our reservation. We should also review if the user who is trying to log in is the one for whom we are holding the reservation. It can be achieved by a simple call to the LUAvailable method as follows:

```
if (##class(%SYSTEM.License).LUAvailable()) || ($USERNAME = username) > 0{
    return $$$OK
}
```

Next, we should check if our reserved user is currently logged in. If they already have a table at our restaurant, we do not need to bother checking if there is an available spot for them. If the user always connects from a specific address, we can use the LicenseCount method to verify that. It takes the user identifier as an argument and returns an integer to tell us whether or not the user is consuming any licenses right now. For example:

```
set licenseusage = ##class(%SYSTEM.License).LicenseCount("David@127.0.0.1")
```

It will tell us if this user is using a license. If it returns 0, the user is not using it, but if it returns 1, the user does exploit the license. Be aware that if this method returns a number greater than 1, it means that the user has exceeded their maximum number of connections and has begun consuming one license per

connection. In that case, you have a new problem to fix!

If we do not know from what address the connection will be coming, we need to approach this situation differently and take a few extra steps. We should use the ConnectionList class query here. Consider the following code:

```
ClassMethod CheckReservation(username As %String) As %Status
{
    try{
        if (##class(%SYSTEM.License).LUAvailable() > 1) || ($USERNAME = username) {
            return $$$OK
        }
        set rs = ##class(%ResultSet).%New("%SYSTEM.License:ConnectionList")
        set sc = rs.%Execute()
        if $$$ISERR(sc) {$$$ThrowStatus(sc)}
        while rs.%Next(){
            if $P(rs.%GetData(1),"@",1) = username{
                return $$$OK
            }
        }
        $$$ThrowStatus($$$ERROR(5001,"Reserved User Not Logged In."))
    }
    catch ex{
        do ex.Log()
        return ex.AsStatus()
    }
}
```

Please note that this method allows us to reserve only one license based on a username. If your use case requires securing more than one license, you will have to look through the results of the ConnectionList query to count how many customers have already logged in and, perhaps, wandered over to the bar for their first Scotch in the meantime. Then, you will have to ensure that the remaining LUAvailable is sufficient to seat all of your upcoming reservations, not just those greater than one.

This method typically accesses the LicenseCount query from the %SYSTEM.License class using a %ResultSet object. Although it is generally preferred to utilize %SQL.Statement, the %PrepareClassQuery method always fails on this query. That is why we cannot employ it here. This query returns a row for each connection, and then, we iterate through the result set. For each row, the first column contains the license ID, which looks like the username and IP address separated by an @. Since we are only looking for the username (not the combined username and IP address), we want to compare the first part of that column with the supplied username, and if they match, we will return a successful status. However, if we get through the entire result set without finding the username, we will return an error.

If we wished to change this method to check for a role instead (say we want always to allow at least one user with the %All role to log in to let a superuser gain access to the system at all times), we could accomplish that by switching to the %SYS namespace and verifying each user ' s roles as we iterate through the loop. We could employ ##class(Security.Users).Get($P(rs.%GetData(1), " @ " ,1),.props) and then review if our role was included in props(" Roles ") by using props(" Roles ") [" %All " . We do need to remember to return to the original namespace eventually, so we end up being logged in to the correct application namespace. It should look more like the following code:

```
ClassMethod CheckReservation(role As %String) As %Status
{
    try{
        set returnns = $NAMESPACE
        zn "%SYS"
        set sc = ##class(Security.Users).Get($USERNAME,.props)
        if (##class(%SYSTEM.License).LUAvailable() > 1) || (props("Roles") [ role) {
            zn returnns
            return $$$OK
        }
        set rs = ##class(%ResultSet).%New("%SYSTEM.License:ConnectionList")
        set sc = rs.%Execute()
        if $$$ISERR(sc) {$$$ThrowStatus(sc)}
        while rs.%Next(){
            set sc = ##class(Security.Users).Get($P(rs.%GetData(1),"@",1),.props)
            if props("Roles") [ role{
                zn returnns
                return $$$OK
            }
        }
        zn returnns
        $$$ThrowStatus($$$ERROR(5001,"Reserved User Not Logged In."))
    }
    catch ex{
        do ex.Log()
        return ex.AsStatus()
    }
}
```

There are some other practical class queries within the License API that you may want to use as well. If you wish to verify a login to a specific application, ConnectionAppList would be more appropriate. If you want to inspect the type of license: User, CSP, Mixed, or Grace - go for the UserList class query. If you operate the ProcessList query, you can get even deeper into the details of the processes run by the user. Just bear in mind that whenever you get carried away a bit too much, simply slow down the login process.

Now that we have this chunk of code, we must figure out where to put it. However, to make it possible, we need to learn some basics about the %CSP.SessionEvents class. This class contains several methods that you can override to control what happens when CSP sessions start, end, or get a time-out, or when a user logs in or out. Here, we are particularly interested in the OnStartSession() method that runs before a license that was allocated for the session. We can run our method there, and when it returns an error, all we have to do is to set %session.EndSession to 1. It will give the user the standard out of licenses message and ban them from logging in.

We will create a class called User.SessionEvents with the extension %CSP.SessionEvents and our previously defined class method. Then we will override only one method in it to make sure that we keep our space available for other reservations. That method will look like the following example:

```
ClassMethod OnStartSession()
{
    if $$$ISERR(##class(User.SessionEvents).CheckReservation("David")){
        set %session.EndSession=1
    }
```

```
    Quit $$$OK
}
```

If you have multiple namespaces, remember to create this class in all of them with a CSP application that will use it.

The final step will be configuring our application to enable it to use these session events. We can do it in the management portal by going to System Administration, Security, Applications, and Web Applications. We should click applications in which we wish to use the custom session events class. Then, in the Event Class box, we must enter User.SessionEvents. From that point on, every time we log in, our customized OnStartSession method will be called, and other users will not be able to log in if there is no room for our reservation.

There is one critical implementation detail to consider here. Do we want these session events to apply to the System Management Portal or not? If we do, we might put ourselves into a situation where something goes wrong, but we will not be able to log in and fix it. If we do not, there is a chance that someone logging in will take up the license unit we were trying to reserve for ourselves. Perhaps we should edit the code and see if the user has a certain role that allows them to bypass the reservation check. However, this part is up to you.

From now on, every time your regular arrives, you can guarantee they will have a seat!

#Access control #Authentication #ObjectScript #Security #System Administration #Tips & Tricks #InterSystems IRIS