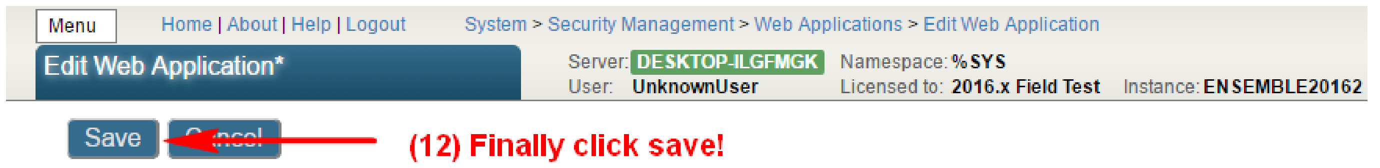
Article
Nikita Savchenko · May 6, 2016 8m read

Installing Caché Applications Using Class Projections

Greetings! This article describes yet another simple way of creating installers for the solutions based on InterSystems Caché. The topic covers applications, which can be installed or completely removed from Caché with one action only. If you are still documenting installation instructions that have more than one step to do to install your application — it's high time you automated this process.

The formulation of the problem

Let’s assume that we’ve developed a small utility for Caché that we want to distribute afterwards. Of course, it would be perfect not to bother users who are going to install it with unnecessary details about configuration and installation. Besides, these instructions would have to be very comprehensive and intended for users who may not know anything about Caché. In case of a web utility, the installer will not only ask the user to import its classes to Caché, but also, as a minimum, to configure the web application for access to it, which is a considerable amount of work:



Of course, all these actions can be performed programmatically. You would only need to find out how to do it. But even in this case, we would need, for example, to ask the user to execute one command in the terminal.

Installation via a single import operation

Caché enables us to perform installation during class import. This means that user will only need to import an XML file with a package of classes using any convenient method:

1. By dragging and dropping an XML file to the Studio area.
2. Through the management portal: System Explorer -> Classes -> Import.
3. Through the terminal: do $system.OBJ.Load("C: \ FileToImport.xml","ck").

The code, which we prepared in advance for installing our application will be executed immediately after the class import and compilation. In case the user would like to uninstall our application (delete the package), we will also have an ability to clean up everything the application has created during installation.

Creating a projection

To extend the Caché compiler behavior, or, in our case, to execute the code during the class compilation or decompilation we need to create a projection class in our package. It is a class which extends %Projection.AbstractProjection, and overrides two it's methods: CreateProjection, which is executed during the compilation, and RemoveProjection, which is triggered during the recompilation or class delete.

Typically, it is a good manner to name this class as Installer. Let's look onto a simple installer example for our package named "MyPackage":

```
Class MyPackage.Installer Extends %Projection.AbstractProjection [ CompileAfter = (Class1, Class2) ]
```

```
{

Projection Reference As Installer;

/// This method is invoked when a class is compiled.
ClassMethod CreateProjection(cls As %String, ByRef params) As %Status
{
   write !, "Installing..."
}

/// This method is invoked when a class is 'uncompiled'.
ClassMethod RemoveProjection(cls As %String, ByRef params, recompile As %Boolean) As %Status
{
   write !, "Uninstalling..."
}

}
```

The behavior here can be described as next:

- When importing and compiling package first time, only the CreateProjection method is triggered;
- When compiling MyApp.Installer next times, or in a case when the "new" installer class is imported over the "old" one, the method RemoveProjection will be triggered for the old class with %recompile parameter equal to 1, and after that the CreateProjection method of the new class is called;
- In a case of the package removal (and MyApp.Installer at the same time), only the RemoveProjection method will be called with parameter recompile = 0.

It is also important to note the following:

- Class keyword CompileAfter should include a list of class names of our application, the compilation of which we need to perform before executing methods of the projection class. It is **always** recommended to fill this list with all the classes you have in your application, because if the error comes up during the installation, we don't need to execute the code of our projection class;
- Both methods accept the cls parameter — it is the top class name, in our case MyApp.Installer. The idea comes from the origin sense of creating projection classes — such "installer" can be done for any class of our application separately, by deriving them from the class, derived from %Projection.AbstractProjection. Only in this case the sense will appear, but for our task it is redundant;
- Both CreateProjection and RemoveProjection methods take the second parameter params — it is an associative array, which handles information about current compilation settings and parameter values of the current class in "parameter name" — "value" pairs. It is quite easy to explore what's inside this parameter by executing zwrite params;
- RemoveProjection method takes recompile parameter, which is equal to 0 only when the class is deleted, but not when recompiled.

Class %Projection.AbstractProjection also has other methods, which we can redefine, but we don't need to do this for our task.

An example

Let's go deeper with the task of creating the web-application for our utility and create a simple case. Suppose we have utility, which is a REST-application, which just sends a response "I am installed!" when is opened in the browser. To create such application we need to create a class that describes it:

```
Class MyPackage.REST Extends %CSP.REST
{
```

```
XData UrlMap
{
<Routes>
   <Route Url="/" Method="GET" Call="Index"/>
</Routes>
}

ClassMethod Index() As %Status
{
   write "I am installed!"
   return $$$OK
}

}
```

Once the class is created and compiled, we need to register it as a web-application entry point. I illustrated the way how it can be configured in the top of this article. After performing all these steps it would be nice to check if our application works by visiting http://localhost:57772/myWebApp/ (Note the following: 1. Slash at the end is required; 2. Port 57772 may differ in your system. It will match the port of your Management Portal's port).

All of this steps, of course, may be automated with some code inside CreateProjection method for creating web-application, and code in RemoveProjection method, which will delete it as well. Our projection class, in this case, will look as follows:

```
Class MyPackage.Installer Extends %Projection.AbstractProjection [ CompileAfter = MyPackage.REST ]
{

Projection Reference As Installer;

Parameter WebAppName As %String = "/myWebApp";

Parameter DispatchClass As %String = "MyPackage.REST";

ClassMethod CreateProjection(cls As %String, ByRef params) As %Status
{
   set currentNamespace = $Namespace
   write !, "Changing namespace to %SYS..."
   znspace "%SYS" // we need to change the namespace to %SYS, as Security.Applications class exists only there
   write !, "Configuring WEB application..."
   set cspProperties("AutheEnabled") = $$$AutheUnauthenticated // public application
   set cspProperties("NameSpace") = currentNamespace // web-application for the namespace we import classes
to
   set cspProperties("Description") = "A test WEB application." // web-application description
   set cspProperties("IsNameSpaceDefault") = $$$NO // this application is not the default application for the
namespace
   set cspProperties("DispatchClass") = ..#DispatchClass // the class we created before that handles the requests
   return ##class(Security.Applications).Create(..#WebAppName, .cspProperties)
}

ClassMethod RemoveProjection(cls As %String, ByRef params, recompile As %Boolean) As %Status
{
   write !, "Changing namespace to %SYS..."
   znspace "%SYS"
   write !, "Deleting WEB application..."
   return ##class(Security.Applications).Delete(..#WebAppName)
}

}
```

In this example each MyPackage.Installer class compilation will create a web-application, and each "decompilation" will remove it. It would be nice to add more checks whether our application exists or not before we create or delete it (by using, for example, ##class(Security.Applications).Exists("Name"), but for the simplicity of this example it is left as a homework for those who reads this article.

After creating MyPackage.REST and MyPackage.Installer classes, we can export these classes as one XML file and share this file with everyone we want. Those who imports this XML will have the web-application set up automatically and they can start using it in browser.

Result

Unlike the method of deploying applications using the %Installer class, which was described on InterSystems community, this method has the next advantages:

1. The "pure" Caché ObjectScript is used. As for %Installer, it is needed to fill xData-block with specific markup described by not a little piece of documentation.
2. Method which installs our application is executed immediately after class compilation, and we have no need to execute it manually;
3. Method which deletes our application is automatically executed if the class (package) is removed, that cannot be implemented by using %Installer.

The method of application installing is already in use within my projects — Caché WEB Terminal, Caché Class Explorer and Caché Visual Editor. You can find an example of the Installer class there.

Just to mention, there is one other post on developer community describing the power of projections usage written by John Murray.

Also it is worth to mention Package Manager project, which is intended to let third-party apps for InterSystems Data Platform to be installed only by one command or click like it happens in npm-like package managers.

#Compiler #Deployment #Object Data Model #Terminal #Caché

Source URL: https://community.intersystems.com/post/installing-cach%C3%A9-applications-using-class-projections