Article
Rustam Ibragimov · Sep 4, 2018 4m read

UDL support

Have you ever thought why the code that you write in Caché Object Script is exported to an XML file?

The news is that it ' s time to change. Starting from the release of Atelier, a new development environment, and Caché 2016.2, you will be able to export and import files in the format that you use in the studio, into the so-called UDL (Universal Definition Language) format. You can now write code that is not only fast but also looks beautiful in the exported form. However, we get another equally serious issue: if all new projects are in the UDL format, what will happen to backward compatibility?

Let ' s get started

InterSystems treats the topic of backward compatibility very seriously, so we have decided to breathe new life into old projects and give them UDL support as well.

Caché Source Control

The first in line is a simple, yet useful project called Cache Source Control. Its task is very simple – take all project files and export them to a certain folder after each successful compilation, as well as to import new files that appeared in that folder. Prior to conversion, this source control system could export files in two formats: xml — all files had an .xml extension (cls, mac, int, dfi).xml — these files had a .cls .xml extension, etc. All you need to do now for everything to work is to change the ExpMode parameter to UDL.

Installation

Installation requires just 4 simple steps:

- Download or clone the repository
- Import the SourceControl.cls.xml file to Studio
- Open the portal, go to Configuration/Additional Settings/Source Control and choose Util.SourceControl as a source control tool in your namespace
- Open the terminal, open your namespace and run the following command: do ##class(Util.SourceControl).Init()

But what if I want to export to a different folder?

Solution
You have several parameters that you can use for this purpose. ..#ExpMode can be 0, 1, or 2 for export to .xml, .cls.xml or UDL formats, respectively. The ..#SourceFolder parameter specifies the destination for our files. ..#RefreshTime specifies the periodicity at which the folder needs to be checked for new files to be imported. Generally speaking, in combination with git, this project gives you everything you need.

CacheUpdater

The next old project to be resurrected was CacheUpdater. The best way to describe its functionality would be to use an example. For example, you have a GitHub repo and 100500+ servers using it. If you decide something, the

whole thing will turn into a nightmare – you will need to update each and every server manually. CacheUpdater solves this problem by periodically synchronizing local files with GitHub.

Installation

Its installation is a bit trickier:

- Download all the files
- Import Task.cls.xml to Studio
- Open System Operation/Task Manager/New Task
- Come up with a name for the task, choose a namespace where you want to synchronize
- files, choose GitHubUpdater as the task type, enter the details of your repo and GitHub
- account
- Click Finish
- Select the update frequency

You are ready for battle, no additional configuration is needed.

→[Link to the project repository CacheUpdater]

CacheGitHubCI

This project is a fully-fledged system for continuous integration with GitHub.
Project installation

- Just download Install.cls.xml (you can download all the file if you want) and import it to your
- project
- Then run the following in the terminal ^
- Set pVars("Namespace") = "{namespace}"
- Set pVars("IP") = "{IP}"
- Do ##class(CacheGitHubCI.Install).setup(.pVars)

where “ namespace ” is the namespace that you want to install the project in. If it doesn ’ t exist, it will be created automatically; IP is an optional parameter of your server (used for webhooks).
Fora example,

```
Set pVars("Namespace") = &amp;quot;SAMPLES&amp;quot;
Set pVars(&amp;quot;IP&amp;quot;) = &amp;quot;45.45.45.45:57776&amp;quot;
Do ##class(CacheGitHubCI.Install).setup(.pVars)
```

That ’ s it, everything is set for an operability check. We will now need to repeat the same steps to create a Task as we did with CacheUpdater'e. Great, everything works! I guess you are now wondering why you need CacheGitHubCI altogether if you can use CacheUpdater.

Answer
CacheGitHubCI enables you to specify the actions that have to be completed before and after compilation and use unit tests. You can configure webhooks for a global update of everything as soon as there is a new commit in the repo – and much, much more. Detailed information about all the bells and whistles is available in the repo itself.

→[Link to the project repository CacheGitHubCI]

Summary

We have updated 3 repos so far. There are more to go, including Cache Tort Git, which enables you to work with

git in combination with TortoiseGit.

#Caché #Development Environment