Article
Timur Safin · Feb 2, 2017 19m read

Part II – making community package manager

> *This is the second part of my long post* about package managers in operating systems and language distributions*. Now, hopefully, we have managed to convince you that convenient package manager and rich 3rd party code repository is one key factor in establishing of a vibrant and fast growing ecosystem. (Another possible reason for ecosystem success is the consistent language design, but it will be topic for another day.)*

In this second part we plan to discuss the practical aspects of creating a package manager in general and their projection to the Caché database environment.

Let assume we want to develop Caché Community Package Manager (CPM for short), and let try to estimate how much we should write if we would try to establish some very basic functionality to the initial release of a package manager? Should we do anything in the kernel, or could it be done completely as an external service? What would be the *minimum viable functionality,* which we could deliver at the most beginning? When package management system becomes useful? All those questions directly or indirectly will be answered in this article below.

Container Format

1st question to answer before start - what is composing a " package " in our system? What about the simplest case ever - when only Caché classes/routines to be deployed? How we would store multiple file types there?

In the ideal case – we could use some ZIP container, but in the *simplest case* the simple XML file, as a Caché Studio project export file, could still serve the purpose. Even today, using this rudimentary container we could embed multiple supported file types (CLS, RTN, INC, CSP, ZEN, CSS, GIF, etc.) packed to the simple, XML entity. Unfortunately, we still have some limitation here, and not everything is possible to add to such Studio project, less could be added using Studio UI, more could be added via programmatic access to Studio Manager (i.e. you could add global binaries such programmatic way), but there are still some unsupported formats, which are not embeddable.

Yes, we know, XML, by its nature, is very inefficient, bloated, and verbose format. [I hate it, and would select anything else, given easy to select alternative] In theory, XML could handle binary files if they are properly base64 encoded, but those files exported would become very, very huge, very fast. In any case…

For the initial implementation, we could ignore any possible inefficiency, and address it later via extensibility interfaces to be established, hopefully, with the help of community.

Metadata file format

2nd question to answer, when designing package managers - what should we put as the metadata info? What format we will use writing metadata? (Let us first answer to semantics question, and then we will talk about possible syntax)

Certainly there should be some " dependency information " (to make possible recursive install of all dependent packages), but what else?

As usual (at least as it was used in the prior part of article) we start discussion from ancient Perl example. Here is the metadata information from some abstract CPAN module using `ExtUtils::AutoInstall` package functions,

which is not usually part of distribution, but one has handy facility for dependency tracking:

```
use inc::Module::Install;

name 'Joe-Hacker';
abstract 'Perl Interface to Joe Hacker';
author 'Joe Hacker <joe@hacker.org>';
include 'Module::AutoInstall';
requires 'Module0'; # mandatory modules
feature 'Feature1', -default => 0, 'Module2' => '0.1';

auto_install(
  make_args => '--hello', # option(s) for CPAN::Config
  force => 1, # pseudo-option to force install
  do_once => 1, # skip previously failed modules
);

WriteAll;
```

We do not see any special (domain-specific) language used, but rather see a few, specially crafted methods, which are simplifying process of describing package metadata. *Clever usage of available language constructs is one of possible approaches in package definition.*

Collections of key-values pairs as package metadata which we want to describe, is just the collection. And we could use all the formats we got used for serialization of object collections – we could use JSON or YAML for that.

```
{
  "name": "leftpad",
  "version": "0.0.0",
  "description": "left pad numbers",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": "https://github.com/tmcw/leftpad.git",
  "keywords": [ "pad", "numbers", "formatting", "format"],
  "author": "Tom MacWright <tom@macwright.org>",
  "license": "BSD",
  "readmeFilename": "README.md"
}
```

In general, using of JSON format for package description is very simple and straightforward; it is server ’ software responsibility to interpret values accordingly context. For example, while keeping us constrained by JSON syntax, we could interpret string literals depending on the current context and circumstances, like, for dependency information in the node.js/npm we could put version and address information in these numerous formats:

```
 { "foo" : "1.0.0 - 2.9999.9999"
, "bar" : ">=1.0.2 <2.1.2"
, "asd" : "http://asdf.com/asdf.tar.gz"
, "elf" : "~1.2.3"
, "lat" : "latest"
}
```

While writing our own package manager we could start from something resembling JSON format used by the NPM packages.

The only single problem here – there is no direct support for importing from .JSON files even for recent versions of Caché/Ensemble with builtin JSON support in the kernel. There is still external package.json file, created for package metadata description, and it will eventually break its ' connection with internal package metadata loaded to Caché, and will become inconsistent, if there will be some direct changes to package fields introduced by developer. We need to invent something, which will keep metadata database updated, while keeping metadata from source-tree synced with internal CPM state. That ' s why we have introduced embedded JSON blocks for package metadata.

```
Class CPM.Sample.PackageDefinition Extends CPM.Utils.PackageDefinition
{
XData Package [ MimeType = application/json ] {
    {
        "name": "cpm-embedded-package-sample",
        "description": "CPM package sample with the embedded package definition",
        "author": "tsafin",
        "version": "0.5.0",
        "license": "MIT",
        "dependencies": {
            "async": ">= 0.2.10",
            "fsplus": ">= 0.1.0",
            "language-cos": "https://github.com/UGroup/atom-language-cos.git"
        }
    }
}
}
```

The goal of such XData block is quite simple: package author writes metadata of a package in the JSON block to any random class, which will be inherited from special `CPM.Utils.PackageDefinition` class. When one recompiles his class, as side-effect of such compilation, it also updates metadata for the corresponding package (e.g. named here " cpm-embedded-package-sample "), which will be later used at the step of package export if you will try to publish modified component to the CPM registry.

At least that was original idea. We certainly would prefer to use `package.json` directly as in NPM, but JSON files were not natively supported by Caché class compiler (at least at the moment we started it 2 years ago), thus we can not rely on plain *.JSON files here, and need to wrap them anyhow.

System Dependencies

When we are talking about dependencies between packages and classes there is interesting problem arises - somehow, we have to declare dependency of our component on some particular version of " built-in " Caché/Ensemble/HealthShare/TrakCare prerequisite class(es), which may have been introduced by some particular version of a particular product?

Extending this question wider - how we could mark package dependency on anything from database system part, i.e. residing in %CACHELIB or any other similar system database? Or how we cold mark some extra license requirements (e.g. iKnow or DeepSee feature enabled)? May be such way?

```
"SysDependencies": "2015.1+, iKnow, DeepSee",
```

Right now, for simplicity matter (at least during few initial releases) we may just ignore that problem al together, and if we deploy any extra class referring to anything in the system database then we will just assume it ' s there and ready to be used. [Well the most we could do now - is to just warn user about some particular requisite in the package README.md file]

In the ideal case we may have facilities to require dependency on some particular version (i.e. " >2014.1 ") of particular product (" Cache " , " Ensemble " , " HealthShare " , " EM " , etc.) or even some particular package installed (" iKnow " , " SetAnalysis " , etc) This is too early though to try to invent some definitive mechanism, so we may leave this question unanswered. We ' ll think about tomorrow

I can't think about that right now

> *[I heard that HealthShare developers work on their own package management system, and solution of system dependency problem is of particular interest for them due to all dependency matrixes used right now in the HealthShare product. Eventually they may propose their own approach to describe system dependency and they may get back to us with CPM pull request implementing this functionality. Who knows?]*

Cross-platform binary modules

CPAN became a big thing not only due to a huge number of pure Perl modules available, but also, thanks to relatively easy way to use native C modules and libraries within Perl. So, if one needed to call some " mission critical " , highly optimized code, or wanted to call some 3rd party C/C++ library, then after some PerlXS massaging (PerlXS API Tutorial) one could perform it from his Perl code. Actually, rudimentary Caché callout interfaces were modelled very similar to original PerlXS mechanism. Worth to note, though, that GT.M XC mechanism [External Call Tables] is much closer to PerlXS in both aspects of implementation and usability, but they both (Caché Callout and GTM XC) are of the same league of "old school" FFI interfaces.

In any case, for interfaces like callout there was no a *simple and direct* way to call *any* C/C++ function or class if you do not have some wrapper cooked for this particular purpose. On the other hand, Perl had *multiple* handy utilities which simplified process of creation of such wrapper, such as:

- h2xs preprocessor to generate XS header using the given C header file (well with some limitations);
- xsubpp - preprocessor to convert XS code to pure C code, etc;

Frequently, convenience to use, and usability are key factors in wide adoptions, and that PerlXS easy to use might explain the reason why it was widely used in the Perl market, and why callout is almost unused today here.

Current situation is not as bad as it used to be with only callout/callin mechanisms available decade ago, there are a couple of rather modern, and relatively more convenient approaches to call external C/C++ code from Caché kernel, without kernel recompilation and alike:

- There is undocumented, builtin foreign-function interface [FFI] mechanism used by several Caché components (like iKnow) for calling external C++ class methods from COS class methods. Here and after we will refer to this mechanism as " cpp-dispatch " ;
- Also, there is GitHub project named CNA (Caché Native Access) which uses cross-platform LibFFI library for easy calling of *any* C-runtime function directly from COS code;

The advantage of cpp-dispatch mechanism is its ' tight integration to class dispatch mechanism in the Caché kernel, but there is catch – it ' s fully undocumented at the moment and is unsupported. Furthermore it ' s not supporting C interfaces, but only C++. [And I ' d not consider cpp-dispatch as " easy " to use if you are not Caché kernel developer, and not accustomed to their usual build procedure].

OTOH, CNA (LibFFI) is external to Caché kernel, supports any C method for the target platform, and is not relying on any unsupported interface in the Caché (only simple dynamic callout bridge is used as a glue). There is small problem on Windows though - executables generated by mingw are not very compatible with VisualC generated code if they are using, passing or returning double float values, but otherwise it ' s very stable, solid and portable. But without direct support of C++ class methods.

From the practical prospective though, taking into account multiple Caché platforms customer would be expecting to handle well (Windows, Linux, Mac OS X, or even VMS?), and the fact that both of these FFI implementations are not yet officially supported by different reasons, we should admit that they both are not yet ready for prime time and could not be recommended (at least, today) as a practical way to handle deployment of mixed C/COS packages.

> *So for the initial release, lack of binary packages support in CPM Will not be a big issue, eventually, once we will go cross-platform, with binary packages support, we may revisit this topic.*

Unit Testing

Yet, another good practice, which we have learned from CPAN, and which we would love to have applied in our package manager – is using of obligatory unit testing of module before its final installation to the target machine. If I recollect correctly (and that was more than decade ago!), each popular Perl package had built-in set of unit-tests, which supposed to be run after the moment sources compiled and before installation happen. If, some of given unit-tests, will fail on the target system, then installation will not be completed. This way CPAN system provided greater stability on a systems which had a ton of 3rd party Perl/C modules installed - they all had to pass internal unit testing before been deployed.

> *For simplicity sake we may ignore unit-tests in our 1st iteration, but once CPM will approach binary package format (i.e. ZIP) and binary modules will be added – then testing will become required step before installation.*

Command-line access

User experience is a key factor here - simplicity wins. If this system would be inconvenient to use then there is big

chance to stay unnoticed. So, to be useful to wide COS/Ensemble developers community we supposed to handle well 2 major cases of a package installation scenarios:

- to install it from interactive shell via ðo ^CPM`
- or to install packages from command-line, i.e. èpm install DeepSee-Mobile`

From practical point of view they should be interchangeable and provide the same side-effects for each operation. Administrators/DevOps would love you if you would provide them easy to use scripting interface for installation of packages to the database system.

> *In a longer terms, once infrastructure is established and mature enough, there should start development of a GUI wrapper for major package manipulation operations (e.g. making them callable from SMP pages), but GUI is not required at the 1st step.*

Mirroring, CDN and cost-effectiveness

In the early years of package managers (e.g. 199x-200x) each separate package management system introduced so far has faced all the same problem and had to deal with it separately – they had to invent *cost-effective way* to deliver data *as fast as possible*, probably using some volunteering, world-wide community support. Initially the common way to adress ths problem was establishing a big network of geographically spread mirror servers. [Which may be quite costy, if you spend your own money for hosting.] Such " old school " software repositories and mirroring networks frequently relied on community resources (good examples of such networks were CPAN, CTAN, Debian, etc.). They are still using the same approach today, but since recently there are easier, and much cheaper solutions to this same problem.

Like today, there is available a cheap facility of CDN (content delivery network) providers. If we need to host some set of *static* binary or textual files then CDN is just " that doctor ordered " . You might use some generic VM-hosting provider like Amazon or Azure, which provide CDN services, or even select among more specialized one like Amazon CloudFront, or MaxCDN, or similar. Any of mentioned CDN providers would allow you to avoid creation of your own mirroring network, and rather to use already created, and very efficient ones.

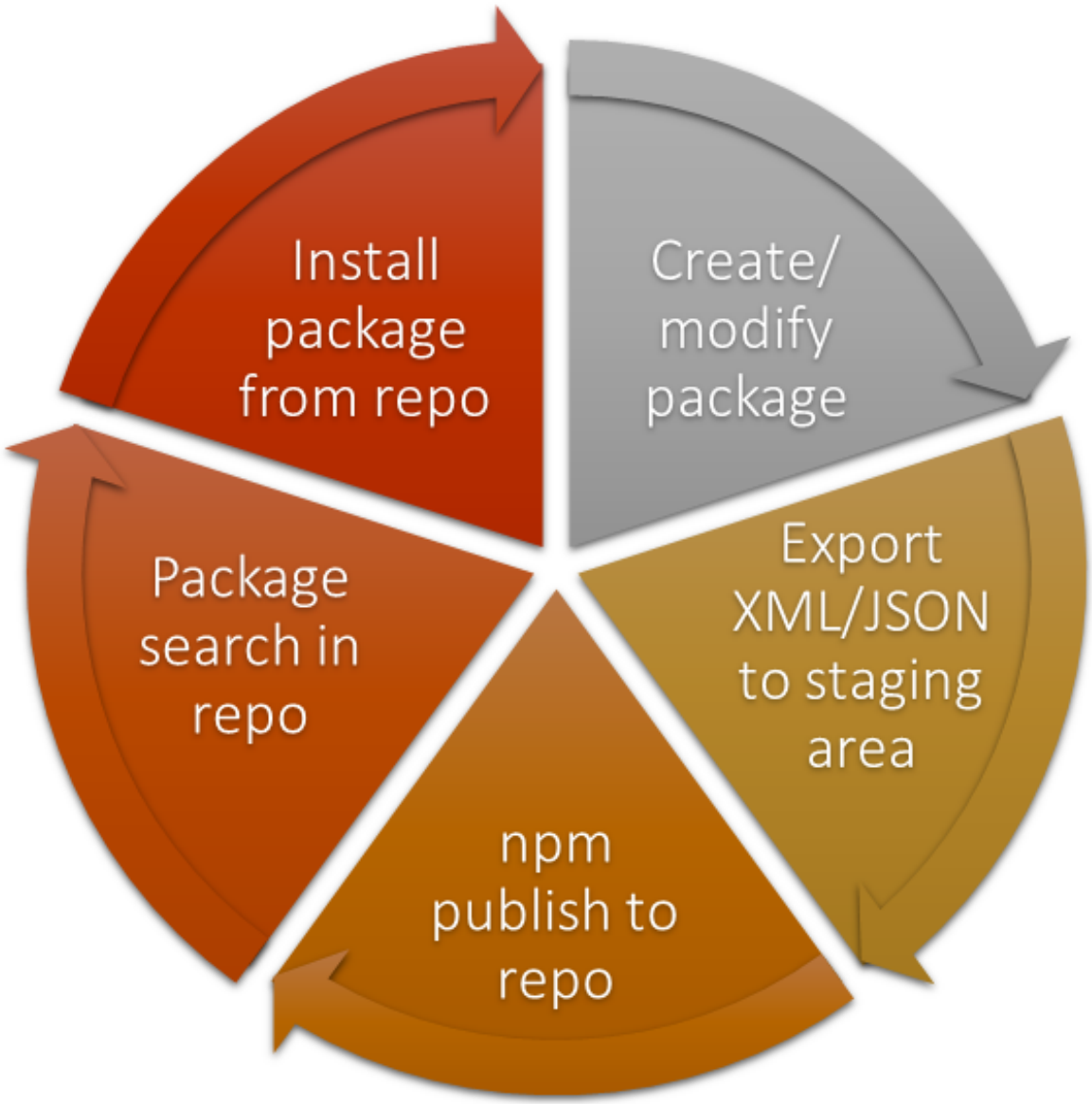
But we, for purposes of hosting CPM registry data, selected very different approach - Dmitry Maslennikov has suggested to use free CouchDB hosting (smileupps.com) for keeping our packages metadata information and their archives content, the same way as Node Package Managers (NPM) does.

Nothing could beat " free " , so we have selected this way.

> *Many have asked us – why we do not use Caché for storing such package assets, even using the compatible RESTful? The answer is obvious – someone should pay for Caché hosting (there is no yet any free), and free (as in beer) CouchDB hosting give us real independence for a longer run.*
>
> *You know that? Freedom and independence are 2 greatest values which you can not buy for money. And you should keep it as long as possible. But I digress.*

Where we are today?

[We are full cycle]



That was a long story, with a lot of interruptions, and various sabbaticals, but, regardless of numerous external reasons, and *only* after a couple of years of development, we have achieved the state when CPM is becoming useful for the average Joe McMumpster.

One could use CPM in the interactive shell or from command-line on Windows, could search component from CHUI interface, or in the nice GUI (web-site available from here cpmteam.github.io), could create component using his own project or GitHub repo as a basis, could export, and what is most important – could *install* found component using 1 simple command. This will be CPM system responsibility to fetch, unpack, and install asked component.

Nothing is rocket science here, we have seen it many times before, elsewhere, (everywhere), but this is quite a promising tool, opening new horizons for community. (Ok, ok, I ' m not very much objective here, and may slightly exaggerate).

How to install CPM?

```
git clone https://github.com/cpmteam/CPM
do $system.OBJ.ImportDir("/path/to/CPM/","*.xml","ck",,1)
```

At the moment, CPM expects to be loaded to the database and namespace named CPM, [though it ' s not enforcing it yet]. Eventually (for the release V1.0 at least) we will provide setup package for easy setup. Right now, for version V0.8 (or alpha) this is quite in programmers ' mode only, you supposed to know what to do with GitHub and know how to load multiple classes into the system.

How to get into the CPM shell?

```
USER>do ^CPM
```

This supposed to work from any namespace, because of automatically installed routine and class mappings.

CPM commands cheat-sheet

There are few commands available in the CPM shell today, they all are listed below. Well, to be precise, there are few more, but which were introduced as aliases to those mentioned. In any case, there are only 9 basic commands implemented. In the future we anticipate more command implemented via CPM plugins mechanism (there is special class established, which should be inherited from to make new command available to the system).

The most important command – help

```
CPM:CPM>> help

Caché Package Manager

Available commands:
   install     install package
   package     package creation commands
   export      prepare to publish - export package(s)
   search      search packages
   list        list of installed and available packages
   purge       purge all installed packages
   config      get/set configuration parameters
   help        this help
   quit        quit from shell

CPM:CPM>>
```

Help screen about (some) command:

```
CPM:CPM>> help install
/dryrun|/d
/s|/silent
/verbose|/v
```

Yes, this is that much terse. Eventually we (or any volunteer) will clarify each subcommand, but at the moment it is

as it is.

install	/dryrun¦/d /s¦/silent /verbose¦/v	Install package from repository
package	/debug /c¦/create <name> /delete /prj:¦/from: <project.pkg> /autoload /author: <name> /description: <text> /license: <license-type> /version: <version-string> /i:¦/import: <github-url>	Package manipulations (including creation and deleting). The simplest case is to make a new package is to load meta directly from GitHub repository (/import suboption).
export	/v¦/verbose /minor /major	Export package description to the " staging " area, from wh upload package to the registry site.
search	<package-regexp>	Search for package given the search pattern
list	/v¦/verbose /local /remote	List all known packages (local and remote). Verbose mode will add extra information (e.g. list of files, mo package)
purge		Purge (local) package database

config	/g¦/get
	/s¦/set
	/debug
		Write to or read from configuration database. Also allows to c configuration (i.e. GitHub token, or last update time):
		`CPM:CPM>> config^CPM.Config("GitHub.Token")="`
		`09757a726ae733"^CPM.Config("updated")=148598`

help		
quit		

2nd by importance command is " quit " . Once you know how to ask for the help, and how to leave shell – you are well

equipped and prepared for work with CPM.

CPM command-line interface (CLI)

As I've already said it many times - user experience is a big success booster, if system is inconvenient then it may be left unnoticed*

As developer would expect from the normal package manager we provide 2 ways to work with it:

- Interactively, in the usual CHUI mode via CPM shell in the Caché terminal, .e.g

```
do ^CPM
install intersystems-ru/monlbl-viewer
```

- Or directly, from OS command-line shell (which will be simply redirecting to the equivalent CPM shell command via special entry point BATCH^CPM)

```
cpm install intersystems-ru/monlbl-viewer
```

- At the moment of announcement only single running Caché instance is supported, and there are no options to select configuration name and target namespace, but this limitation will be lifted very soon.
- Furthermore, at the moment of announcement only Windows platform has CLI wrapper implemented to some degree, but Linux is the obvious next target, so stay tuned.

Process of package publishing

That brilliant Dmitry Maslennikov' idea to simply use CouchDB hosting as a package repository store has made many things much simpler, at least for those of us, who have spent some time dealing with modern JavaScript/NodeJS components and familiar with NodeJS/npm tools. CouchDB hosting mimics NPM REST API, thus we could rely on `npm` itself for publishing new packages to the remote registry. [The only change you have to take care of – is to redefine address of NPM registry to that used by CPM https://cpmisc.smileupps.com].

Here is simplified picture of a process of publishing of a new package:

1. You prepare package definition inside of the CPM shell (see numerous subcommands of `package` command).
2. When you are done preparing – you stage intermediate files using `export package-name;`
3. Export is exporting 2 major files:

- `package.json` with package metadata being serialized to JSON format,
- and XML Export file with all relevant classes for this given package;

1. `npm publish --registry https://cpmisc.smileupps.com` will pack all assets, mentioned in the `package.json`, and upload them to the CPM registry at the address https://cpmisc.smileupps.com

But here is a bad news for you – at the moment we operate CPM registry in the *"curated mode"* (i.e. only current CPM devteam could publish new packages to the registry). That is not intended limitation, but rather temporary decision, while proper means of authorization will be implemented in the front-end and back-end. Today we start from 17, specially crafted packages in the initial registry, but we hope to get much larger registry very soon (with the help of developers community, with your help).

CPM repository

We (as the InterSystems Caché/Ensemble community) have already a good chunk of useful open-source projects available on GitHub, SourceForce, BitBuckets, and elsewhere. Here is statistics for several InterSystems hubs known to me on GitHub (as of the moment of writing of this article):

Hub	Own Repositories	Forks
Intersystems-ib	8	1
Intersystems-ru	53	47
Intersystems	7	29

Not all of those repositories were suitable for our purpose as sample for CPM component (some of them were C/C++ projects, some – Java, some hybrid desktop applications), but many of them were Caché ObjectScript based, and we could easily select the first few which will be used at the CPM launch.

bdeboe/isc-iknow-dictbuilder
bdeboe/isc-iknow-explorer
bdeboe/isc-iknow-extractor
bdeboe/isc-iknow-ifindportal
bdeboe/isc-iknow-rulesbuilder
bdeboe/isc-iknow-setanalysis
eduard93/Cache-FileServer
eduard93/Cache-MDX2JSON
intersystems-ru/CacheGitHubCI
intersystems-ru/EnsembleWorkflow
intersystems-ru/EnsembleWorkflowUI
intersystems-ru/iknowSocial
intersystems-ru/monlbl-viewer
intersystems-ru/REST
intersystems-ru/webterminal
tsafin/cache-map-reduce
tsafin/cache-zerocopy-tree

Some of those components were quite simple and could be loaded locally to any namespace, some of those had projection-based autoinstall scripts (creating some side-effects like database/namespace created, and web application configured), and few of them were “ super-wise ” and had manifest-based setup routines. For the initial release we have excluded COS packages which assumed to be loaded to %CACHELIB (including our favorite cache-tort-git component, but we will address this limitation as soon as possible). Because it ’ s hard to live in Studio without proper Git support from cache-tort-git. Yes, Atelier has built-in Git support, but no – we will not migrate to Atelier in CPM development due necessity to support some releases which are slightly older than Caché 2016.2.

Final words

So, today we are approaching a moment, when Caché Community Package Manager is becoming useable. So all invited to give it a try: download CPM, got to the shell, play with components, look for another components. There are multiple issues at the moment, which should be addressed before we call it product quality, but hey, it ’ s open-source, if you dislike it – you go and fix it!

In any case, even at this early stage of CPM development we would love to hear your feedback, suggestions, or (which is even better) to see your pull-requests with extensions to the system!

This gonna be fun - go play with it!

#C++ #Callout #Change Management #Code Snippet #Deployment #Git #Namespace #Node.js #Tips & Tricks #Caché