



D13.1 – Software Release Procedures and Tools JRA2

Version 1.0 (*final*)

20 June 2019

Grant Agreement number:	823914
Project acronym:	ARIADNEplus
Project title:	Advanced Research Infrastructure for Archaeological Dataset Networking in Europe - plus
Funding Scheme:	H2020-INFRAIA-2018-1
Project co-ordinator name, Title and Organisation:	Prof. Franco Niccolucci, PIN Srl - Polo Universitario "Città di Prato"
Tel:	+39 0574 602578
E-mail:	franco.niccolucci@pin.unifi.it
Project website address:	www.ariadne-infrastructure.eu

The research leading to these results has received funding from the European Community's Horizon 2020 Programme (H2020-INFRAIA-2018-1) under grant agreement n° 823914.

Authors

Massimiliano Assante, ISTI - CNR

Gianpaolo Coro, ISTI - CNR

Luca Frosini, ISTI – CNR

Pasquale Pagano, ISTI - CNR

Manuele Simi, ISTI – CNR

Contributors

Leonardo Candela, ISTI - CNR

Roberto Cirillo, ISTI - CNR

Andrea Dell’Amico, ISTI – CNR

Lucio Lelii, ISTI - CNR

Francesco Mangiacrapa, ISTI – CNR

Giancarlo Panichi, ISTI – CNR

Fabio Sinibaldi, ISTI – CNR

Quality control check

Julian Richards, UoY-ADS

Document History

- 11.06.2019 – Draft Version 0.1
- 12.06.2019 – Executive Summary and Introduction
- 13.06.2019 – Section 3 revision
- 14.06.2019 – New Section 4 and Section 5
- 18.06.2019 – New Section 6
- 20.06.2019 – Overall revision
- 21.06.2019 – Quality control check

Table of Contents

Document History	3
Table of Contents	4
1 Executive Summary	6
2 Introduction and Objectives	7
3 Software management: Continuous Integration	9
3.1 Continuous Integration: Workflow	9
3.2 Version Control System (VCS)	9
3.2.1 Benefits of Using a Version Control System	11
3.2.2 Git	11
3.2.2.1 Create new Git Repositories	12
3.2.2.2 Initialize an empty Repository	12
3.2.2.3 Add an existing Repository	12
3.2.2.4 Commit Best Practices.....	12
3.2.2.5 Git Branching Strategy	13
3.2.2.6 Tag Revisions for Releases	13
3.2.2.7 Configure authors info	15
3.3 Maven	15
3.3.1 Building gCube software.....	16
3.3.2 Maven Repositories.....	17
3.3.3 Build profiles.....	17
3.3.4 Settings Files.....	18
3.3.5 Build Configurations	19
3.4 Jenkins.....	20
3.4.1 Jenkins Rules.....	20
3.4.2 Isolated Builds	21
3.4.3 The gCube Template Project	21
3.4.4 Connecting Jenkins Projects	23
3.5 Connecting Git Repositories to Jenkins projects	24
3.5.1 What is a webhook?	24
3.5.2 Gitea Plugin on Jenkins (only for Jenkins admins)	24
3.5.2.1 Installation	24

3.5.2.2	Configuration.....	24
3.5.3	Build project configuration.....	25
3.5.4	Webhook on the Gitea repository.....	26
3.5.5	Testing the Gitea/Jenkins round trip	28
4	Software management: Continuous Delivery	30
4.1	Principles.....	30
4.2	Jenkins Release Pipeline.....	30
4.3	Software Releases	33
4.3.1	Developer Activity	34
4.3.2	Release Manager Activity	34
4.3.3	Maven Release plugin	36
5	Software Methods Provision and Integration	37
5.1	Functional specifications.....	38
5.2	Software and Algorithms Integration.....	41
6	Tools Provision and Integration	43
6.1	SmartGears	43
6.2	OAuth2.0.....	45
7	References	46

1 Executive Summary

This deliverable D13.1 – “Software Release Procedures and Tools JRA2” describes the activities carried out during the first six months of the ARIADNEplus project within Task 13.4 (T13.4) Software integration and release (JRA2.4) (Work Package 13 - WP13) and describes the procedures governing the release of software, methods, and tools for the ARIADNEplus infrastructure.

This task is in charge of managing the process of the software maintenance, enhancement, and provisioning in JRA work packages. Thus, it i) defines release and provisioning procedures; ii) establishes the release plan; iii) coordinates the release process; iv) operates the tools required to support the release and provisioning activities; v) validates the software documentation; vi) takes care of the distribution of the software and its provisioning. This task benefits from the practices established and experience gained within the D4Science infrastructure.

The procedures are documented through a set of documentation pages for single facilities hosted by the gCube wiki. In particular, this report provides the instructions and rules for three main patterns governing the provision of software and tools to the ARIADNEplus infrastructure.

The first pattern is related to the provision of software to the gCube infrastructure enabling technology. This pattern describes the procedures for the storage of software and its management; the continuous integration of the software to build releasable software artefacts; and the generation of software distribution packages.

The second pattern is related to the provision of software methods elaborated by the ARIADNEplus community requiring execution within the ARIADNEplus infrastructure.

The third pattern is instead related to the provisions and integration of tools into the ARIADNEplus infrastructure.

2 Introduction and Objectives

A distributed and dedicated infrastructure, tools and well-established procedure are fundamental to properly manage software integration and releases.

This deliverable describes the workflow, tools, and the practices required to release and integrate software, methods, and tools within the D4science infrastructure (<https://www.d4science.org/>) [4] to properly support the operation of the ARIADNEPlus infrastructure (<https://ariadne.d4science.org/>).

The first identified objective is related to the provision of software to the gCube [1] (<https://www.gcube-system.org/>) infrastructure enabling technology. This pattern describes the procedures for the storage of software and its management; the continuous integration of the software to build releasable software artefacts; the generation of software distribution packages and automatic management of releases.

"Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly." Martin Fowler (<https://www.martinfowler.com/>)

Continuous Integration is a well-known procedure in software development that aims to reduce time to deliver software improving its quality. Continuous Integration consists of a set of practices and policies that should be enforced as widely as possible among the development team. The most important practices to follow are:

- keep source code in a repository, preferably using a Version Control System (VCS) e.g. SVN, GIT;
- developers commit on mainline of the source repository as frequently as possible, keeping the mainline builds successfully;
- automated builds on source code changes basis;
- automated builds should include all source code (even database scripts!). The idea is that anyone can, standing in front of a hitherto unused machine, check out source code from the repository, build it and have a running system on the machine;
- builds are self-testing. Source code includes tests that are automatically executed at each build. Testing include not only unit-testing, but also functional and deployment testing for instance.

"Continuous Delivery is a software development discipline where you build software in such a way that the software can be released to production at any time."

You're doing continuous delivery when:

- *Your software is deployable throughout its lifecycle*
- *Your team prioritizes keeping the software deployable over working on new features*
- *Anybody can get fast, automated feedback on the production readiness of their systems any time somebody makes a change to them*
- *You can perform push-button deployments of any version of the software to any environment on demand"* Martin Fowler

gCube achieves *Continuous Delivery* by continuously integrating the software crafted by the development team, building executables, running automated tests on those executables to detect problems and automatically packaging and distributing complete releases.

The second identified objective is related to the provisioning of software methods elaborated by the ARIADNEplus community to manipulate and elaborate data that require execution within the ARIADNEplus infrastructure.

The third identified objective is instead related to the provisioning and integration of tools into the ARIADNEplus infrastructure. Those tools are independent from the infrastructure but thanks to the integration into the infrastructure they can improve the quality of service and performance.

Section 3 describes the patterns for continuous software integration. It presents the selected Version Control System tool; the established procedure to develop a gCube software component; and the continuous integration architecture and procedure to build the gCube Software components.

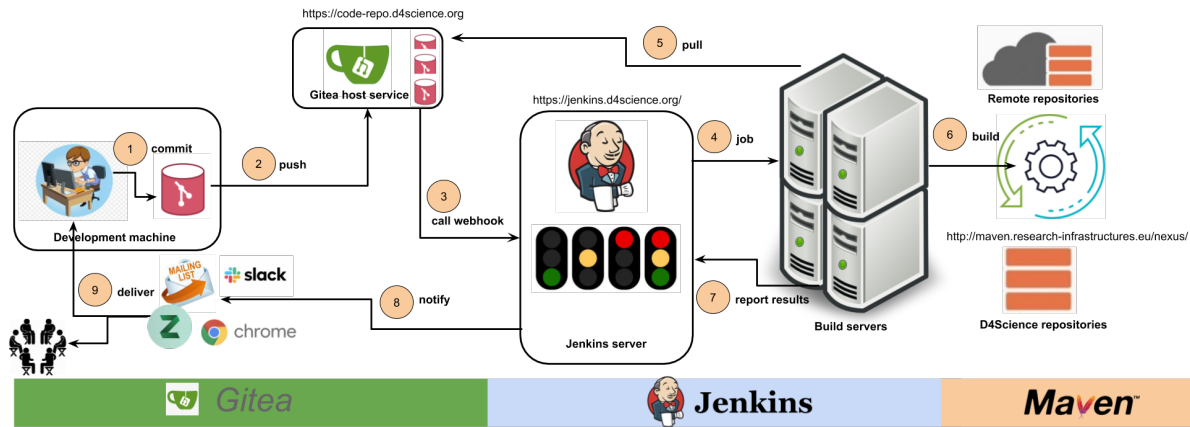
Section 4 describes the software release procedure and the practices expected by the involved actors.

Section 5 describes the second identified objective for the provisioning, management and integration of software methods.

Section 6 describes the third identified objective for the provisioning, management and integration of tools.

3 Software management: Continuous Integration

3.1 Continuous Integration: Workflow



The enabling technologies selected to properly support the continuous integration process in gCube are: Gitea (as Git hosting service), Jenkins (as automation server) and Maven (as project management and comprehension tool).

Their proper configuration and interactions are the foundation for automating the non-human part of the software development process, with continuous integration and facilitating technical aspects of continuous delivery.

The workflow involves interaction with the Git repositories (managed by the D4Science Gitea instance) and uses the Maven project management and comprehension tool. Maven is based on the concept of a Project Object Model (POM). Maven can manage a project's build, reporting and documentation from a central piece of information. Every gCube component is described by a POM. Thanks to POM component definition, Maven, among another things, allows one to:

- make the build process easy;
- provide a uniform build system;
- provide quality project information;
- provide guidelines for best practices development;
- allow transparent migration to new features.

3.2 Version Control System (VCS)

A Version Control System is software keeping track of every modification to the files belonging to a repository. VCSs are especially used to keep track of software source code.

The nature of a VCS is described in the following note by Atlassian (a leading software company that develops products for software developers, project managers, and content management).

What is a version control system (Atlassian.com CC BY 2.5 AU)

For almost all software projects, the source code is like the crown jewels - a precious asset whose value must be protected. For most software teams, the source code is a repository of the invaluable knowledge and understanding about the problem domain that the developers have collected and refined through careful effort. Version control protects source code from both catastrophe and the casual degradation of human error and unintended consequences.

Software developers working in teams are continually writing new source code and changing existing source code. The code for a project, app or software component is typically organized in a folder structure or "file tree". One developer on the team may be working on a new feature while another developer fixes an unrelated bug by changing code, each developer may make their changes in several parts of the file tree.

Version control helps teams solve these kinds of problems, tracking every individual change by each contributor and helping prevent concurrent work from conflicting. Changes made in one part of the software can be incompatible with those made by another developer working at the same time. This problem should be discovered and solved in an orderly manner without blocking the work of the rest of the team. Further, in all software development, any change can introduce new bugs on its own and new software can't be trusted until it's tested. So testing and development proceed together until a new version is ready.

Good version control software supports a developer's preferred workflow without imposing one particular way of working. Ideally it also works on any platform, rather than dictate what operating system or tool chain developers must use. Great version control systems facilitate a smooth and continuous flow of changes to the code rather than the frustrating and clumsy mechanism of file locking - giving the green light to one developer at the expense of blocking the progress of others.

Software teams that do not use any form of version control often run into problems like not knowing which changes that have been made are available to users or the creation of incompatible changes between two unrelated pieces of work that must then be painstakingly untangled and reworked. If you're a developer who has never used version control you may have added versions to your files, perhaps with suffixes like "final" or "latest" and then had to later deal with a new final version. Perhaps you've commented out code blocks because you want to disable certain functionality without deleting the code, fearing that there may be a use for it later. Version control is a way out of these problems.

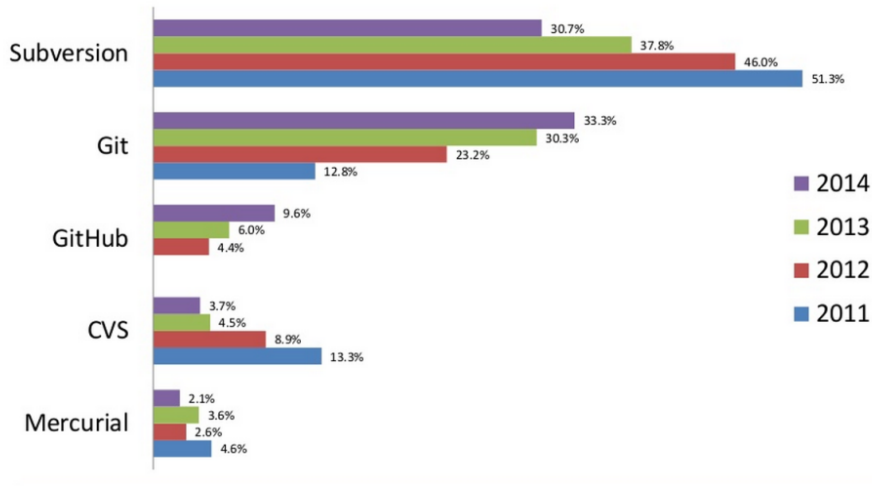
Version control software is an essential part of the every-day of the modern software team's professional practices. Individual software developers who are accustomed to working with a capable version control system in their teams typically recognize the incredible value version control also gives them even on small solo projects. Once accustomed to the powerful benefits of version control systems, many developers wouldn't consider working without it even for non-software projects.

In recent years, the use of Git has grown. According to an Eclipse community survey, in 2014 Git surpassed SVN as a VCS of choice for Java developers.

Primary Code Management



What is the primary source code management system you typically use? (Choose one.)



The gCube software system follows this trend and it adopts Git as its code management system. In the rest of this section we provide the overview of the best practices adopted by the gCube developer community to use Git and to migrate the existed code from SVN to Git.

3.2.1 Benefits of Using a Version Control System

The main advantages of using a VCS in gCube include streamlining the development process, management of code for multiple projects and keeping a history of all changes within the code.

A VCS saves all the changes in a repository. Hence, if the developers make a mistake, they can undo it. At the same time, they can compare the new code with a previous version(s) to resolve any issues. This can reduce human errors and unintended consequences to a great extent.

Additionally, it can be integrated with several software development tools like PaaS providers, integrated development environments (IDE) and build automation tools playing a key-role in Continuous Integration and release management procedures.

3.2.2 Git

A complete code hosting solution for Git has been deployed on the D4Science infrastructure (<https://code-repo.d4science.org/>) to properly manage the Git repositories and its settings and to enable the creation of new components. In particular, Gitea has been selected. Gitea has a user interface which looks like the most famous git repository i.e. GitHub. It is an open source project published under the MIT license (<https://gitea.io/>).

The rest of this section uses the Gitea D4Science installation as reference to perform the task and provide examples and screenshots, but the various steps can be easily replicated/adapted within any other Git code hosting solution.

3.2.2.1 Create new Git Repositories

From the web interface, click on the New Repository button:

- Make sure the owner is the organization;
- Create the repo as private.

If the name of the repository is **myNewRepo** and it is created within the **gCubeSystem** organization, the repo URL is:

`https://code-repo.d4science.org/gCubeSystem/myNewRepo.git`

3.2.2.2 Initialize an empty Repository

If the repository belongs to a new project (i.e. it is not imported), on the developer machine:

```
$ touch README.md
$ git init
$ git add README.md
$ git commit -m "first commit"
$ git remote add origin https://code-repo.d4science.org/gCubeSystem/myNewRepo.git
$ git push -u origin master
```

It is strongly recommended that each repository has a README.md (in Markdown format) in the root folder and each significant subfolder. The README should briefly explain the content of the repository, how to build it and link the related wiki documentation.

3.2.2.3 Add an existing Repository

On the developer machine, in the root folder of the repository, add the new remote to the configuration and then push the entire repository in the master branch:

```
git remote add origin https://code-repo.d4science.org/gCubeSystem/myNewRepo.git
git push -u origin master
```

3.2.2.4 Commit Best Practices

General rules about commits in the Version Control System:

- Each commit **MUST** be as self-contained as possible. A commit can potentially be cherry picked by any branch to apply its change to the target branch;
- Each commit **MUST** have an appropriate comment. General rules about commit comments:
 - Each commit **MUST** have an appropriate comment.
 - Use the comment to explain what and why, NOT how
 - Capitalize the first character of the comment
 - End the comment with a period
 - Use the imperative mood in the comment
 - Limit the comment to 2 or 3 sentences.

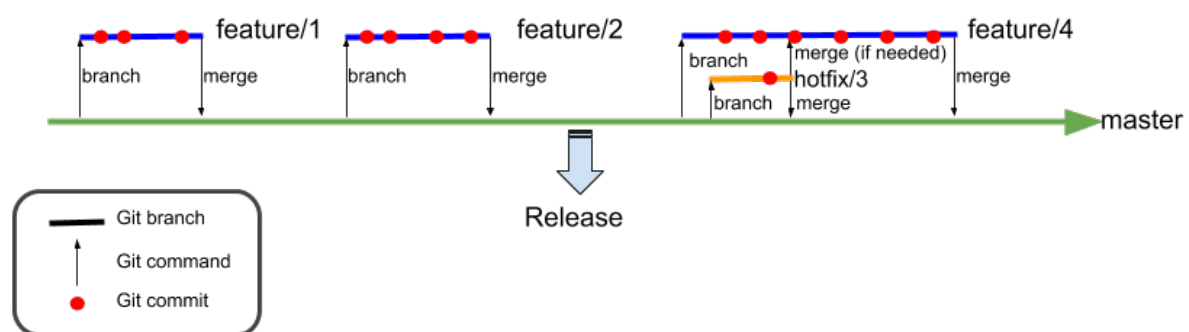
Some rules to avoid bad practices in usage:

- VCS is not a backup system! Developers who use it as such tend to do end-of-day commits, checking in everything at the end of the day. The result is useless, a random diff across the code with changes that are impossible to understand by anyone, including the original author once a few months have passed;
- Avoid lazy commit messages, any commit labelled as "misc fixes and cleanups" or similar. Messages must be meaningful;
- Avoid whitespace/formatting/cosmetic changes together with code changes in the same commit;
- Never save two unrelated changes in one commit. Something like "Fixed bug 2345 and renamed all foo to bar". Unless bug 2345 required the renaming, fixes would be split into multiple commits.

3.2.2.5 Git Branching Strategy

To properly use the VCS (i.e. Git) for the Continuous Integration Procedure (see section 3.3) the developer **MUST** respect the following simple strategy:

- The master branch is the stable branch. Must be always in a releasable state;
- Feature (named feature/issue) branches are created from master (issue is the tracker issue number that describes the feature);
- Occasionally, feature branches can be created from other feature branches, although this practice is discouraged;
- Features may be experimental. If not pursued, branches are discarded without corrupting the stability of the master branch;
- When a feature is complete, the corresponding feature branch is merged into the master branch;
- When the master has enough stable features, it is released;
- If an issue (typically a bug requiring immediate attention) is detected in the master, a hotfix/issue branch is created from the master (issue is the tracker issue number that reports the problem);
- Once the hotfix is complete it is merged to the master and any open feature branch (if needed).



A useful guide on how to branch and merge with Git can be found at the following link <https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging>

3.2.2.6 Tag Revisions for Releases

Git support tagging is a mechanism to identify a revision. Tags are mainly used to declare component releases.

3.2.2.6.1 Creating Tags

Git supports two types of tags: lightweight and annotated. In gCube we use lightweight tags to mark a specific point in the repository's history.

The following command creates a tag on the last commit in the current branch:

```
$ git tag <tagname>
```

Tags can be created anytime in the project's history.

Suppose your commit history looks like this:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

Now, suppose you forgot to tag the project at v1.2, which was at the “updated rakefile” commit. To tag that commit, you specify the commit checksum (or part of it) at the end of the command:

```
$ git tag v1.2 9fceb02
```

You will have to explicitly push tags to a shared server after you have created them.

```
$ git push origin <tagname>
```

3.2.2.6.2 Tagging Rules

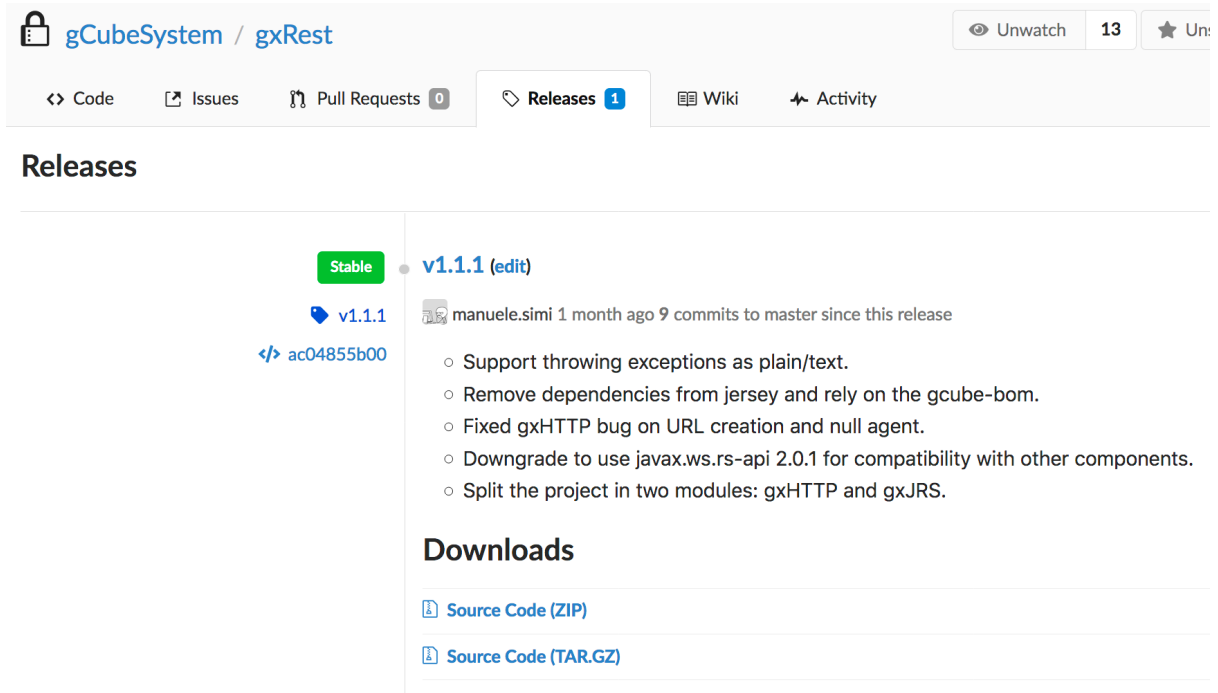
The following best practice per tagging has been defined in the gCube system:

- We tag only the master branch;
- Tags are used to mark releases;
- Tags are applied after the related gCube release is declared closed;
- The tag name must be in the format "vX.Y" or "vX.Y.Z" (where X,Y,Z are follow the versioning schema major.minor[.build]).

3.2.2.6.3 Release Tags

It is strongly recommended to create a release from each tag in Gitea.

This is an example of release created in the gXRest repository starting from the tag v1.1.1:



gCubeSystem / gxRest

Unwatch 13 Unstar

Code Issues Pull Requests 0 Releases 1 Wiki Activity

Releases

Stable • **v1.1.1** (edit)

 v1.1.1  ac04855b00

 manuele.simi 1 month ago 9 commits to master since this release

- Support throwing exceptions as plain/text.
- Remove dependencies from jersey and rely on the gcube-bom.
- Fixed gxHTTP bug on URL creation and null agent.
- Downgrade to use javax.ws.rs-api 2.0.1 for compatibility with other components.
- Split the project in two modules: gxHTTP and gxJRS.

Downloads

 [Source Code \(ZIP\)](#)

 [Source Code \(TAR.GZ\)](#)

The list of commits messages since the previous release tag serves as releases notes for v.1.1.1. Tagged releases can be also downloaded from Gitea.

3.2.2.7 *Configure authors info*

3.2.2.7.1 Single repo configuration

In the root folder of the repository:

```
$ git config user.email "john.green@company.com"
$ git config user.name "John Green"
```

3.2.2.7.2 Global configuration for all repos

In any folder:

```
$ git config --global user.email "john.green@company.com"
$ git config --global user.name "John Green"
```

3.3 Maven

A build tool is a tool that automates everything related to building a software project.

The advantage of automating the build process is to minimize the risk of humans making errors while building the software manually. Additionally, an automated build tool is typically faster than a human performing the same steps manually. gCube adopts Maven as its build tool.

3.3.1 Building gCube software

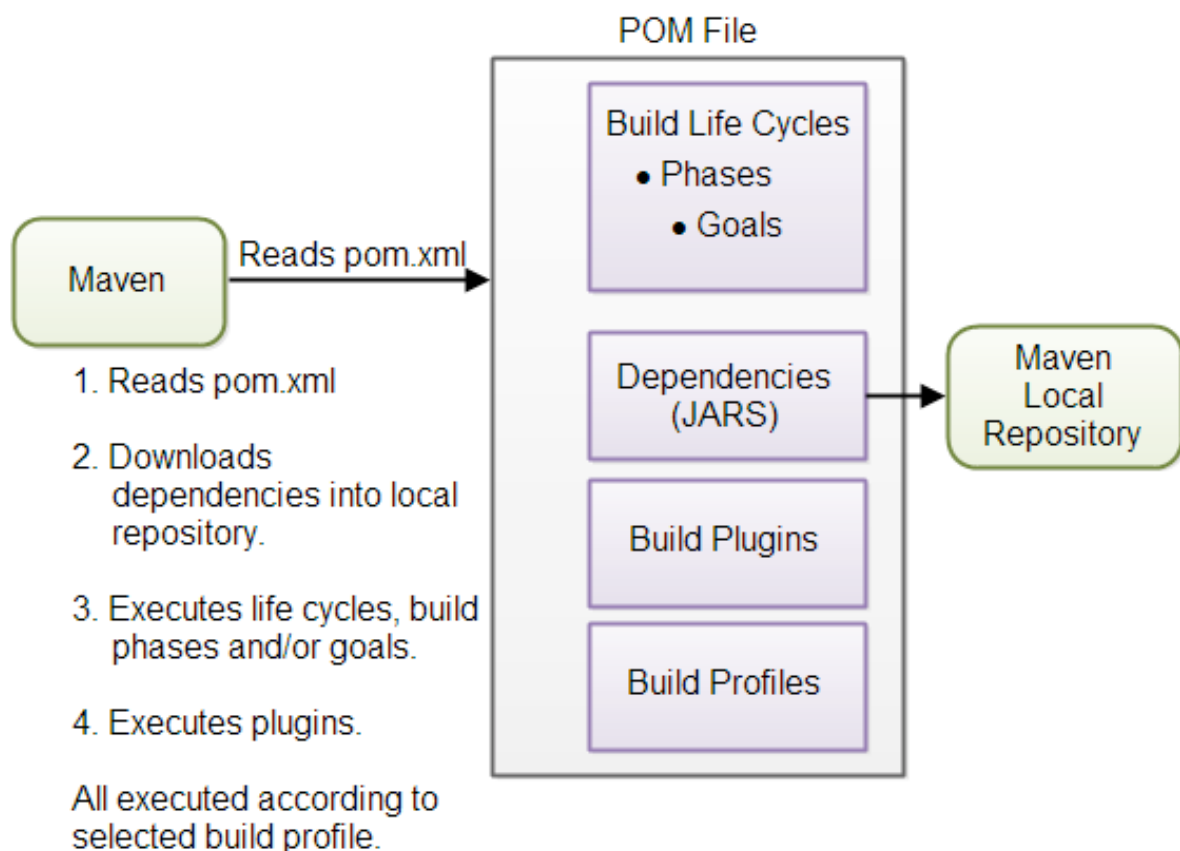
Building a gCube software component typically includes these activities:

- Verifying the quality of the source.
- Generating documentation from the source code.
- Compiling source code.
- Packaging compiled code into JAR, WAR or ZIP files.
- Installing the packaged code on a server, in dedicated software repository.

Such activities are automated by plugging them inside the Maven build process.

Maven is centered around the concept of POM files (Project Object Model). A POM file is an XML representation of project resources like source code, test code, dependencies (external JARs used) etc. The POM contains references to all of these resources. The POM file should be located in the root directory of the project it belongs to.

The following diagram illustrates how Maven uses the POM file, and what the POM file primarily contains:



Build Plugins and Profiles have been highly exploited to configure the build process (see sections 3.3.3, 3.3.5) and the release process (see section 4.2)

3.3.2 Maven Repositories

Maven requires a dependency repository to work properly. The D4Science infrastructure makes use of an instance of Nexus (<https://www.sonatype.com/nexus-repository-oss>). Nexus is a free artifact repository with universal support for many formats which is compatible with Maven.

The D4Science Nexus instance is configured to host the following repositories:

- *gcube-snapshots*: for artifacts under development
- *gcube-staging*: for artifacts to be deployed on the D4Science production infrastructure
- *gcube-releases*: for artifacts deployed on the D4Science production infrastructure

3.3.3 Build profiles

Build profiles are used when a software component needs to build in different ways. For instance, you may need to build your project on your local computer, for development and test. And you may need to build the same component for deployment on your production environment on a remote build server. To enable different builds, different build profiles can be added to the POM files. When executing Maven we can tell which build profile to use.

A gCube artifact can be built in 3 different situations:

- on a personal (development) machine, likely after new code has been developed, by a gCube developer;
- on a Jenkins slave, likely triggered by a new commit in the Git repository;
- on a Jenkins slave, as part of the release pipeline.

To support these scenarios, there is a Maven component (named *maven-parent*), which **MUST** be used as super component by all the gCube components. The gCube Maven Parent POM (or super POM) helps avoid redundancies or duplicate configurations using inheritance between POM files. It helps in easy maintenance in the long term.

The gCube Maven Parent defines the following build profiles:

- *gcube-developer*:
 - dependencies resolved against *gcube-snapshots* and *gcube-releases*
 - snapshot artifacts deployed to *gcube-snapshots*
 - deployments of releases artifacts are not permitted.
- *jenkins-snapshots*:
 - dependencies resolved against *local-snapshots* and *gcube-releases*

- snapshot artifacts installed to local-snapshots
 - snapshot artifacts deployed to gcube-snapshots
 - deployments of release artifacts are not permitted
- jenkins-releases:
 - dependencies resolved against gcube-releases
 - deployments of snapshot artifacts are not permitted
 - release artifacts deployed to gcube-releases
- dry-run:
 - this profile disables all the deployments (regardless the maven commands) by skipping the deploy phase. It can be combined with the previous profiles;
- disable-java8-doclint:
 - this profile sets an additional parameter for javadoc generation to disables the doclint. It avoids the build fails if formal/syntax errors are found in javadoc comments. It can be combined with the previous profiles.

3.3.4 Settings Files

The activation and switch among the build profiles is done through different settings.xml files available in the Configs project:

(<https://code-repo.d4science.org/gCubeSystem/Configs/src/branch/master/Maven/1.1.0/>).

Namely:

- gcube-developer-settings.xml

Used by: gCube Developer

Installed on: development machine

- jenkins-snapshots-settings.xml

Used by: Jenkins pipeline jobs

When: To test a complete snapshot and deploy to the gcube-snapshot remote repo

Installed on: slave node

- jenkins-snapshots-dry-run-settings.xml

Used by: Jenkins pipeline jobs

When: To test a complete snapshot without deploying on a remote repo

Installed on: slave node

- jenkins-job-settings.xml

Used by: Jenkins jobs

When: To build a snapshot project and deploy to the gcube-snapshot remote repo

Installed on: slave node

- jenkins-release-settings.xml

Used by: Jenkins pipeline jobs
 When: To test a complete release and on the gcube-releases remote repo
 Installed on: slave node

- jenkins-release-dry-run-settings.xml

Used by: Jenkins pipeline jobs
 When: To test a complete release without deploying on a remote repo
 Installed on: slave node

3.3.5 Build Configurations

The build configurations must guarantee the integrity of the Continuous Integration pipeline. Jenkins builds (see section 3.4) must use only the outcomes of other Jenkins builds. A snapshot artifact built on a development machine and deployed to the Maven snapshot repository is a potential threat for the integration process. There are several cases in which such an artifact could not be in sync with the content of the SCM repository. For instance:

- it is built from a branch not built on Jenkins;
- the source code that generates it is committed but not pushed;
- it has dependencies on the local environment not reproducible on the Jenkins slaves;
- ..and so on.

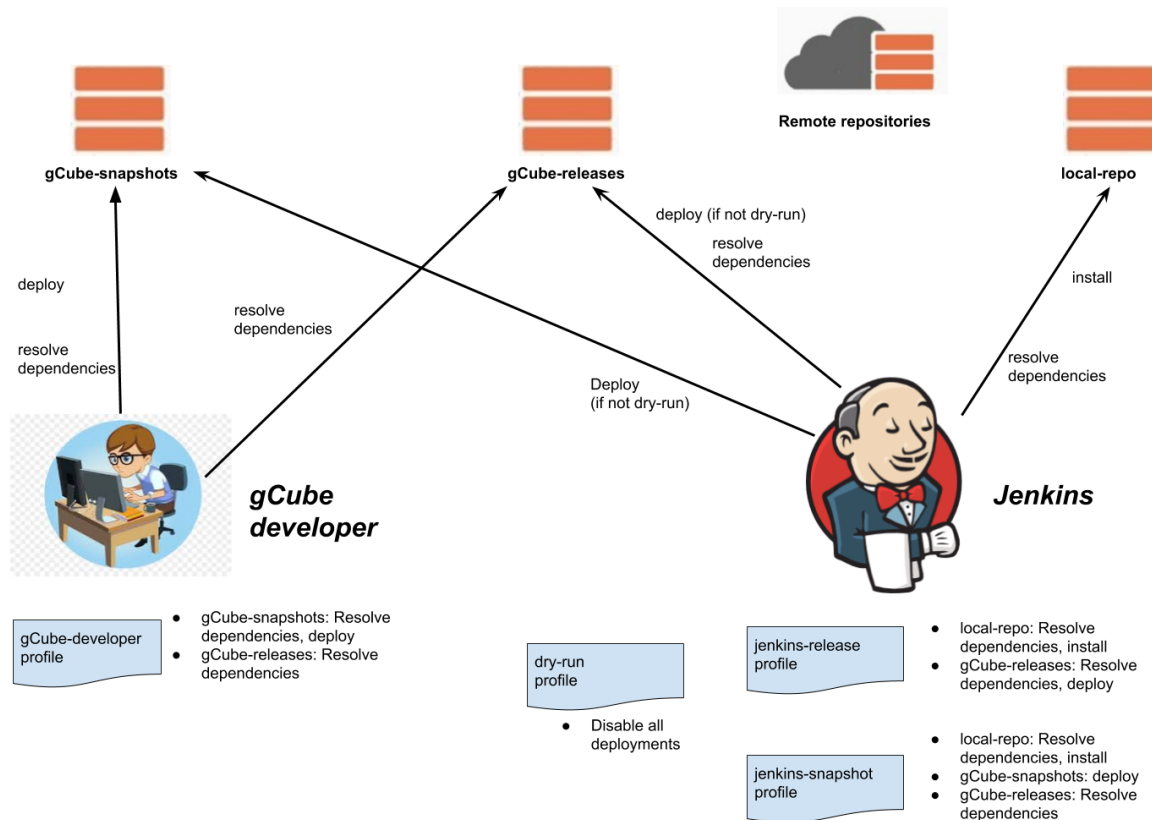
What we need:

- Jenkins must use only artifacts generated by Jenkins builds to resolve dependencies;
- Snapshot artifacts deployed to Maven snapshot repository by developers must not be seen by Jenkins;
- Jenkins must deploy to Maven snapshot repository after successful builds;
- Release artifacts are built and deployed to the Maven release repository (gcube-releases) only by Jenkins.

What we are trying to avoid:

- Reconfigure all the Jenkins projects at each release time;
- Change all the POMs before starting the release integration;
- Create ad-hoc artifacts or repositories.

By combining the build profiles in the appropriate settings file, we support the following build scenarios with the Maven Repositories.



3.4 Jenkins

Jenkins provides a user-friendly dashboard to manage all the steps to create, remove and modify and monitor a component, which must be integrated and deployed in the D4Science infrastructure.

Jenkins enables us to create a workflow to integrate a component. This workflow is often referred as a pipeline. A Jenkins instance has been deployed in the D4science infrastructure and it is available at <https://jenkins.d4science.org/>.

One of biggest risks of a non-properly configured Continuous Integration pipeline is to trigger too many builds on Jenkins and transform the pipeline into something we can call Continuous Building. Multiple builds of the same project simultaneously on the same slave can interfere with each other and inconsistent situations are very common. For instance, this is the case when each commit is immediately to the master branch or when the Git repository is wrongly used as a backup system.

This deliverable illustrates a few procedures to prevent this misbehavior. Some of them assume a proper usage of Git from the user.

3.4.1 Jenkins Rules

gCube requires the following rules and practices for any Jenkins project:

- a Jenkins project builds always the master branch;

- if a Jenkins project is configured to build a second branch, this branch must generate a software artifact with a different version than master branch;
- the development of new features and bug fixing are done in dedicated branches (task branching);
- merges into master branch are performed when the feature/fix is stable;
- commits are not always pushed to the remote repository, but only when they add a significant, self-contained and consistent piece of working code;
- the master branch **MUST** be always in a releasable state

By following them, builds are triggered only when a stable feature is merged into master, while commits in the other branches do not involve Jenkins. If two branches are built at the same time in a Jenkins project (which should be temporary), their different versions guarantee that there are no conflicts in the published artifacts.

3.4.2 Isolated Builds

Golden Rule: *Jenkins builds MUST depend only on the output of other Jenkins builds (exceptions are Releases and third-party artifacts).*

A mistake to avoid with the pipeline is to use the public gcube-snapshot Maven Repository. We must guarantee that the SNAPSHOT dependencies resolved during the builds on the servers are **ALWAYS** coming from other Jenkins builds. If we let the builds use the SNAPSHOT versions manually deployed by the developers, the entire pipeline is **INVALID**.

One long-standing solution to this is to use the “Use private Maven repository” option in the “Advanced” section of the Maven build. This creates an isolated local Maven repository for the job (in \$WORKSPACE/.repository) which prevents these problems. Jenkins releases since 1.448 let you specify a Maven repository per executor, which is a more efficient way to solve the same problem. Read more in the Jenkins section.

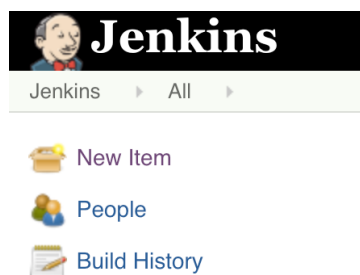
Regarding the dependencies of released artifacts:

- for gCube components, they are resolved against the gcube-releases Maven Repository;
- for third-party components, they are resolved against Maven Central or other external repositories.

3.4.3 The gCube Template Project

The easiest way to create a new project is to clone the *gCubeTemplate* project.

1. In the Jenkins interface, select New Item from the options on the left panel:



2. In the new item page, assign a name to the project

3. Scroll down until the bottom of the page and use the last option to create the new item from other existing. Insert *gCubeProjectTemplate* as input (it should auto-complete).

If you want to create a new item from other existing, you can use this option:


Copy from

and confirm with OK.

4. In the new project page, you need to:

- uncheck the "Disable this project" checkbox
- provide the URL to the Git repo to build

General
Source Code Management
Build Triggers
Build Environment
Pre Steps
Build
Post Steps
Build Settings

Post-build Actions

☐ This build requires lockable resources
☐ This project is parameterized
☐ Throttle builds
☒ Disable this project
☐ Execute concurrent builds if necessary
☐ Restrict where this project can be run

Advanced...

Source Code Management

☐ None
☒ Git

Repositories

Repository URL

?

Credentials

git.gcube/***** (Git user that can access the gCube/D4Science private

Adel

The new project must then be configured to connect to the Git repository hosting the source code to build.

In the Source Code Management page:

- set option to "Git",
- provide the URL to your repo (e.g. <https://code-repo.d4science.org/gCubeSystem/gxRest.git>),
- set the credentials to *git.gcube*,
- specify **/master* as branch to build,
- and in "Poll triggers" section check "Poll SCM" option with no schedule defined. This setup basically tells Jenkins to poll your Gitea repository only when requested via a webhook (see section 3.4.4).

The screenshot shows the Jenkins configuration page for Source Code Management. The 'General' tab is selected. Under 'Source Code Management', the 'Git' option is chosen. The 'Repositories' section contains a 'Repository URL' field with the value 'https://code-repo.d4science.org/gCubeSystem/gxRest.git' and a 'Credentials' dropdown set to 'git.gcube/*****'. The 'Branches to build' section has a 'Branch Specifier' field with the value '*/master'. The 'Repository browser' is set to '(Auto)'. The 'Build Triggers' section has several options, with 'Poll SCM' checked. The 'Schedule' field is empty. At the bottom, there is a note: 'No schedules so will only run due to SCM changes if triggered by a post-commit hook' and an 'Ignore post-commit hooks' checkbox which is unchecked.

3.4.4 Connecting Jenkins Projects

A project can have one or several *upstream* projects, which means that a build for the current project may be scheduled when an upstream build is finished. Per default every stable upstream build will schedule a build in the downstream project, but there are several options and plugins, which can customize this behaviour.

A project can have one or several *downstream* projects. The current project is then known as an upstream project of the downstream project. See Upstream project for what this means regarding scheduling of builds.

The upstream job is the one that is triggered before the actual job is triggered. The downstream job is the one that is triggered after the actual job is triggered. We can configure the actual job not to be triggered if the upstream job is failed. In the same way, we can configure the downstream job not to be triggered if the actual job is failed.

Order of job triggers:

Upstream Job -> Actual Job -> Downstream Job

Every Jenkins project MUST define its upstream projects in order to guarantee consistency and compatibility with the components it depends on.

3.5 Connecting Git Repositories to Jenkins projects

The gCube continuous integration process MUST guarantee that a software component is built right after an updated version is pushed to the Git repository. This section describes how the Git repository and the related Jenkins project must be configured to achieve such integration.

3.5.1 What is a webhook?

A webhook is a mechanism to automatically trigger the build of a Jenkins project upon a commit pushed in a Git repository.

This section details the steps to have Jenkins automatically create a build if it detects changes to a Gitea repository. This can be a very useful improvement to continuous integration setup with Jenkins because this method is only telling Jenkins to attempt a new build when a change is detected rather than polling on an interval, which can be a very inefficient.

3.5.2 Gitea Plugin on Jenkins (only for Jenkins admins)

This plugin allows to configure Jenkins to talk to Gitea.

3.5.2.1 Installation

In Jenkins: download and install the Gitea plugin in Jenkins.

Go in the page: Manage Jenkins -> Manage Plugins -> Available -> Gitea plugin

And install the plugin.

3.5.2.2 Configuration

Go in the page: Manage Jenkins -> Configure System -> Gitea Servers:

Gitea Servers

Gitea Server

Name

Server URL

Gitea Version: 73ce024

☒ Manage hooks

Credentials
Add

Hook management will be performed as git.gcube

Alias URL

Delete

Add

3.5.3 Build project configuration

In Jenkins, under the project settings page "Source Code Management":

- set option to "Git",
- provide URL to your repository (e.g. <https://code-repo.d4science.org/gCubeSystem/gxRest.git>),
- set the credentials to *git.gcube*,
- specify **/master* as branch to build,
- and in "Build Triggers" section check "Poll SCM" option with no schedule defined. This setup basically tells Jenkins to poll your Gitea repo only when requested via the webhook.

General
Source Code Management
Build Triggers
Build Environment
Pre Steps
Build
Post Steps
Build Settings
Post-build Actions

☐ None
☒ Git

Repositories

Repository URL

Credentials
Add

Advanced...

Add Repository

Branches to build

Branch Specifier (blank for 'any')

Add Branch

Repository browser

Additional Behaviours

☐ Subversion

Build Triggers

☒ Build whenever a SNAPSHOT dependency is built
☐ Schedule build when some upstream has no successful builds

☐ Trigger builds remotely (e.g., from scripts)
☐ Build after other projects are built
☐ Build periodically
☐ GitHub hook trigger for GITScm polling
☒ Poll SCM

Schedule

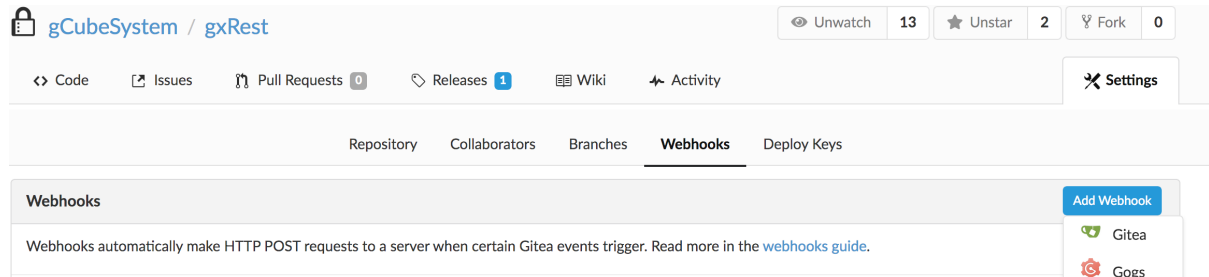
No schedules so will only run due to SCM changes if triggered by a post-commit hook

Ignore post-commit hooks ☐

3.5.4 Webhook on the Gitea repository

In Gitea, under repo -> Settings -> Webhooks:

- click on "Add Webhook",
- select "Gitea" in the drop down list



In the Webhook form:

- set the URL to <https://jenkins.d4science.org/gitea-webhook/post?job=project>, where *project* is the name of the project in Jenkins to build,
- set post content type to "application/json",
- check "Push Events" for the *Trigger On* option,
- clear the secret (leave it blank).

Update Webhook

Gitea will send POST requests with a specified content type to the target URL. Read more in the [webhooks guide](#).

Target URL *

`https://jenkins.d4science.org/gitea-webhook/post?job=gxRest`

POST Content Type

application/json

Secret

Trigger On:

☒ Push Events
☐ All Events
☐ Custom Events...

☒ Active
 Information about triggered events will be sent to this webhook URL.

[Update Webhook](#) [Remove Webhook](#)

Recent Deliveries [Test Delivery](#)

✓ `17421524-ac8c-4708-b697-e9a2dcc7155` 2019-05-09 22:14:17 CEST

Request **Response** 200

Headers

```
Request URL: https://jenkins.d4science.org/gitea-webhook/post?job=gxRest
Request method: POST
Content-Type: application/json
X-GitHub-Delivery: 17421524-ac8c-4708-b697-e9a2dcc7155
X-GitHub-Event: push
X-Gitea-Delivery: 17421524-ac8c-4708-b697-e9a2dcc7155
X-Gitea-Event: push
X-Gogs-Delivery: 17421524-ac8c-4708-b697-e9a2dcc7155
X-Gogs-Event: push
```

Do note that:

1. the value of the job option is case-sensitive (in the example, *gxrest* instead of *gxRest* would fail).
2. HTTPS is needed to send a POST request

At this point clicking on "Test Delivery" button should produce a successful delivery attempt (green checkmark, as shown in the figure).

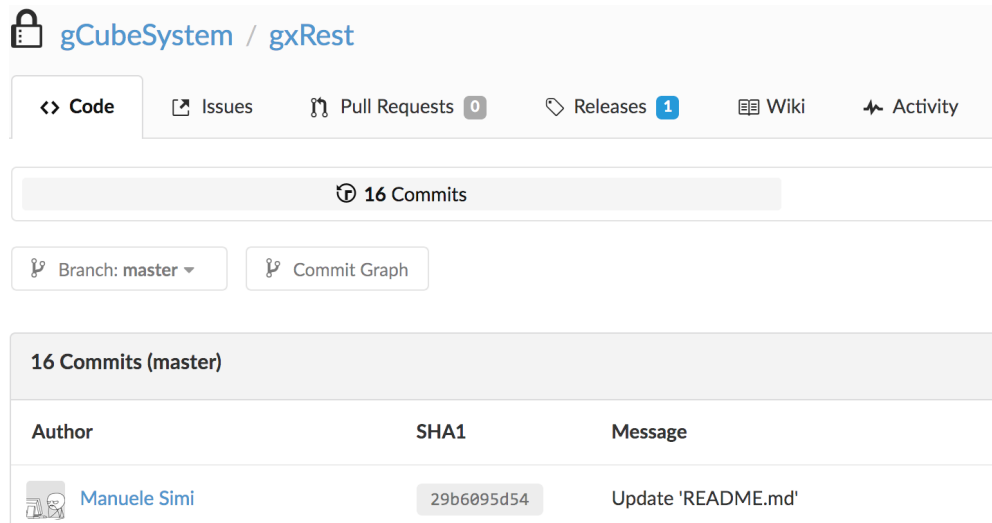
If the test deliveries fail, try to see if you can POST to Jenkins webhook URL (<http://jenkins.d4science.org/gitea-webhook/post>). E.g. using [Postman](#) or with curl:

```
curl -vvv -H "Content-Type: application/json" -H "X-Gitea-Event: push" -X POST http://jenkins.d4science.org/gitea-webhook/post -d "{}"
```

Correct response should be just plain "Processed" string. If you get something else, post it here.

3.5.5 Testing the Gitea/Jenkins round trip

1) We push a new commit with a message "Update README.md" in the Gitea repository:



gCubeSystem / gxRest

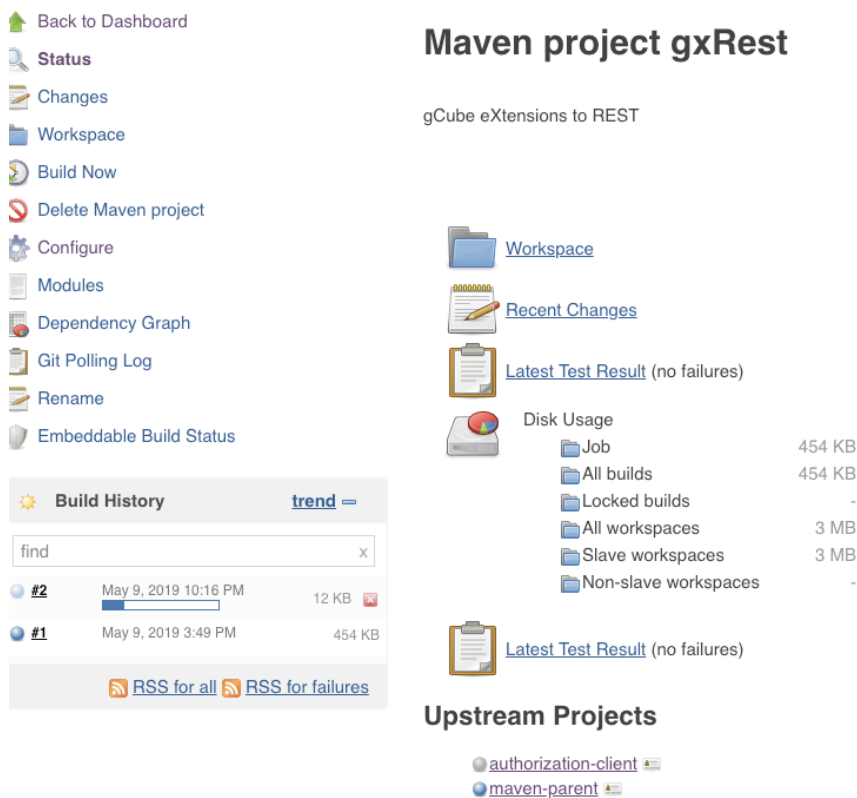
<> Code Issues Pull Requests 0 Releases 1 Wiki Activity

16 Commits

Branch: master Commit Graph

Author	SHA1	Message
Manuele Simi	29b6095d54	Update 'README.md'

2) If the webhook has been properly set up, on the Jenkins interface we should see that the build of the linked project has started:



Back to Dashboard

Status

Changes

Workspace

Build Now

Delete Maven project

Configure

Modules

Dependency Graph

Git Polling Log

Rename

Embeddable Build Status

Maven project gxRest

gCube eXtensions to REST

[Workspace](#)

[Recent Changes](#)

[Latest Test Result](#) (no failures)

Disk Usage

Job	454 KB
All builds	454 KB
Locked builds	-
All workspaces	3 MB
Slave workspaces	3 MB
Non-slave workspaces	-

[Latest Test Result](#) (no failures)

Build History

[trend](#)

find

#2	May 9, 2019 10:16 PM	12 KB
#1	May 9, 2019 3:49 PM	454 KB

[RSS for all](#) [RSS for failures](#)

Upstream Projects

[authorization-client](#)

[maven-parent](#)

3) Clicking on the ongoing build, we can appreciate that it has been triggered by the commit:

Build #2 (May 9, 2019 10:16:26 PM)



Changes

1. Update 'README.md' ([detail](#))



[Started by an SCM change](#)



Revision: 29b6095d54fa403840f95c6e50130b63e7cd5ba1

- refs/remotes/origin/master

Module Builds

[gCube eXtensions to REST with HTTP](#) 11 sec and counting

[gCube eXtensions to JAX-RS](#) Not started yet

 [gCube eXtensions to REST](#) 2.5 sec

4) Projects configured downstream will be also built if the build completes successfully.

4 Software management: Continuous Delivery

Continuous Delivery is producing software faster rather than in big release cycles over weeks or months, so that there exists at any time a stream of potential production grade versions of gCube artifacts that are ready to be released.

The difference between integration builds and delivery builds is that delivery builds publish artifacts as software versions. Typically, projects have many integration builds for verifications, validation etc. but artifacts are published only regularly or manually during a formal release cycle.

4.1 Principles

gCube applies the following principles to Continuous Delivery:

- A regular build is no different than a release build. What makes a release special is how we see it.
- No human intervention should be required. All decisions can be calculated automatically
- If you have multiple branches for some reason, have a dedicated build job for each one of them, in order to see and manage the current branch status easily.
- Branch builds must enforce building from the top, never to be parameterized for building custom changesets.
- Avoid having release branches.
- Avoid having continuous delivery before making code reviews enforced by the build system.
- Block artifact deployments except for the build user
- Make it possible to start everything completely from scratch
- Do not have any snapshot dependency during releases
- Do not use version ranges in dependencies, because it prevents reproducible builds
- Keep most logic inside Maven and Git to keep it reusable everywhere, without need of a build server.

4.2 Jenkins Release Pipeline

gCube uses the declarative pipelines of Jenkins to define the build flow. The pipelines allow to use basic but common building blocks to define the whole build/release process.

Jenkins Pipeline (<https://jenkins.io/doc/book/pipeline/>) is a combination of plugins that support the integration and implementation of continuous delivery pipelines using Jenkins. A pipeline has an extensible automation server for creating simple or complex delivery pipelines "as code" via pipeline DSL (Domain-specific Language).

In gCube we use a Pipeline to trigger the builds of jobs forming a gCube Release. The pipeline project is available at: <https://jenkins.d4science.org/job/gCube-Release/>

These are the parameters of the pipeline:

☒ This project is parameterized

Choice Parameter

Name

Type

Choices

SNAPSHOT-DRY-RUN
 SNAPSHOT
 RELEASE-DRY-RUN
 RELEASE

Description

The type of artifacts the build is expected to generate

[Plain text]

[Preview](#)

Add Parameter

No triggers are defined because the pipeline is expected to be manually launched by the Release Manager (see section 4.3.2):

Build Triggers

- ☐ Build after other projects are built
- ☐ Build periodically
- ☐ Build whenever a SNAPSHOT dependency is built
- ☐ GitHub hook trigger for GITScm polling
- ☐ Poll SCM
- ☐ Disable this project
- ☐ Quiet period
- ☐ Trigger builds remotely (e.g., from scripts)

This behaviour can be changed according to the release needs and the availability of a sufficient number of dedicate agents in Jenkins.

The pipeline definition is maintained in a Git repository; therefore the Jenkins pipeline project is configured to poll that repository when a change occurs:

Pipeline

Definition: Pipeline script from SCM

SCM: Git

Repositories:

- Repository URL:
- Credentials: (Git user that can access the gCube/D4Science private)
- Name:
- Refspec:
-

Branches to build:

- Branch Specifier (blank for 'any'):
-

Repository browser: (Auto)

Additional Behaviours:

Script Path:

The Jenkins file in the Git repository defines a Declarative Pipeline (see <https://jenkins.io/doc/book/pipeline/#declarative-pipeline-fundamentals>).

These are the requirements to run the pipeline:

- Jenkins ver. 2.164.2 or later;
- Pipeline Plugin;
- Pipeline: Basic Steps;
- Pipeline: Maven;
- Jenkins configured with a JDK named 'OpenJDK 8';
- One or more Jenkins agents labelled as 'pipeline-agent'.

The Pipeline's code defines the entire build process of a gCube Release. This is the skeleton of the gCubeRelease pipeline (see <https://jenkins.io/doc/book/pipeline/syntax/> for references on pipeline syntax):

```
// manage options and settings
.....
// pipeline
pipeline {
    // run only on agents with the label
    agent { label 'pipeline-agent' }

    environment {
        JOB_OPTIONS = "${options}"
        MAVEN_LOCAL_REPO = "${maven_local_repo_path}"
        GCUBE_RELEASE_NUMBER = "${params.gCube_release_number}"
    }

    parameters {
        choice(name: 'Type',
            choices: ['SNAPSHOT-DRY-RUN', 'SNAPSHOT', 'RELEASE-DRY-RUN',
                'RELEASE'],

```

```

        description: 'The type of artifacts the build is expected to
generate')
string(name: 'gCube_release_number',
    defaultValue: '1',
    description: 'The number of the gCube release to build.
Ignored by the SNAPSHOT builds. Sample values: 4.14, 4.15,
etc.')
}
stages {
    stage('preliminary steps') {
        //prepare the environment for the builds
        steps {
            //execute steps here if needed
        }
    }
    stage('build group 1') {
        steps {
            withMaven(..maven settings here..) {
                build 'job name 1'
                build 'job name 2'
                build 'job name 3'
                build 'job name 4'
            }
            echo 'Done with group 1'
        }
    }
    stage('build group 2') {
        steps {
            ...
        }
    }
}
// post-build actions
post {
    always {
        echo 'This will always run'
    }
    success {
        echo 'This will run only if successful'
    }
    failure {
        echo 'This will run only if failed'
    }
    unstable {
        echo 'This will run only if the run was marked as unstable'
    }
    changed {
        echo 'This will run only if the state of the Pipeline has changed'
        echo 'For example, if the Pipeline was previously failing but is
now successful'
    }
}
}

```

4.3 Software Releases

Two types of actors are involved in the continuous delivery process:

- Developers: gCube software component developers which follow the procedure described in this deliverable;
- Release Manager(s): a manager responsible for orchestrating the release process.

4.3.1 Developer Activity

The developer **MUST** follow all the procedures described in this deliverable. He/She **MUST** guarantee that the master branch **MUST** be releasable at any time.

When the Release Manager declares a gCube release open and until it is declared close, the artifact version in the POM on master **MUST NOT** have the -SNAPSHOT suffix.

When the Release Manager declares the gCube release close and if the release includes a new version of a component, the developer must tag the master branch.

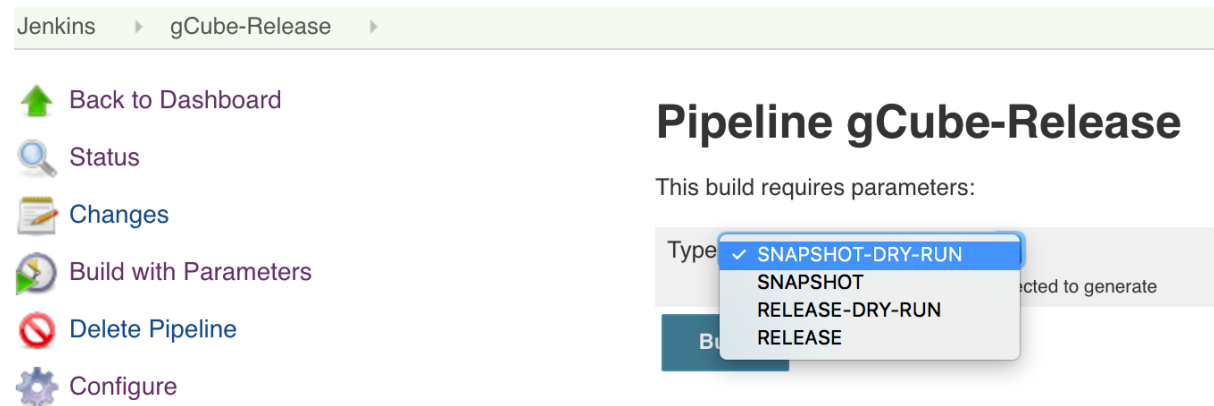
4.3.2 Release Manager Activity

The Release Manager is in charge of:

- Declaring when a gCube release is open and when it is closed
- Maintaining the Pipeline Definition (see section 4.1)
- Launching the build of the gCube Pipeline

To launch the build of the gCube in the Jenkins GUI, the release manager has to:

- select the gCubeRelease project;
- on the left side, click on 'Build with Parameters';
- select the type of build we want to generate in the choice menu;
- click on the 'Build' button;



The Pipeline project can be launched in 4 different ways (Type parameter):

- SNAPSHOT-DRY-RUN (default): build snapshot artifacts, install the artifacts in a local repo, do not deploy;
- SNAPSHOT: build snapshot artifacts, install the artifacts in a local repo, deploy the artifacts to the gcube-snapshots Maven Repository;
- RELEASE-DRY-RUN: build release artifacts, install the artifacts in a local repo, do not deploy
- RELEASE: build snapshot artifacts, install the artifacts in a local repo, deploy the artifacts to the gcube-releases Maven Repository

The idea behind these builds is that the Release Manager can test the full pipeline execution with the DRY-RUN builds. Once all the projects in the build set work, the SNAPSHOT or RELEASE build can be launched to effectively deploy the artifacts on the remote Maven Repository (see section 3.3.2).

As any other job execution, the pipeline can be monitored step-by-step in the Console Output page of the ongoing build:

The screenshot shows the Jenkins interface for build #29 of the 'gCube-Release' project. On the left, a sidebar contains navigation links: 'Back to Project', 'Status', 'Changes', 'Console Output' (highlighted), 'View as plain text', 'Edit Build Information', 'Delete build '#29'', 'Parameters', 'Embeddable Build Status', 'Restart from Stage', 'Replay', 'Pipeline Steps', 'Workspaces', and 'Previous Build'. The main area is titled 'Console Output' and displays the following log:

```

Started by user Manuele Simi
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] Start of Pipeline
[Pipeline] echo
Configure Maven for SNAPSHOT-DRY-RUN artifacts
[Pipeline] echo
Use settings file at null/.m2/settings.xml
[Pipeline] echo
Use local repo at /var/lib/jenkins/local-snapshots
[Pipeline] node
Running on jenkins-worker1-ctl.garr.d4science.org :
[Pipeline] {
[Pipeline] withEnv
[Pipeline] {
[Pipeline] stage
[Pipeline] { (clean up before starting)
[Pipeline] sh
+ echo 'Create a fresh local repository'
Create a fresh local repository
+ rm -rf /var/lib/jenkins/local-snapshots
+ mkdir -p /var/lib/jenkins/local-snapshots
+ echo 'Done with local repository'
Done with local repository
[Pipeline] }
[Pipeline] // stage

```

In addition, the Pipeline steps page (also accessible from the build page) shows the various steps and links to the other builds executed within the pipeline:

The screenshot shows the Jenkins 'Pipeline Steps' page for build #31. The left sidebar is identical to the previous screenshot, with 'Pipeline Steps' highlighted. The main area displays a table of pipeline steps:

Step	Arguments	Status
Start of Pipeline - (1 min 11 sec in block)		
Print Message - (6 ms in self)		
Print Message - (4 ms in self)	Use settings file at /var/lib/jenkins/.m2/jenkins-snapshots-dry-run-settings.xml	
Print Message - (0.11 sec in self)	Use local repo at /var/lib/jenkins/local-snapshots	
Allocate node : Start - (1 min 11 sec in block)	pipeline-agent	
Allocate node : Body : Start - (1 min 11 sec in block)		
Stage : Start - (0.88 sec in block)	Declarative: Checkout SCM	
Declarative: Checkout SCM - (0.86 sec in block)		
Check out from version control - (0.85 sec in self)		
Set environment variables : Start - (1 min 10 sec in block)	GIT_BRANCH, GIT_COMMIT, GIT_PREVIOUS_COMMIT, GIT_PREVIOUS_SUCCESSFUL_COMMIT, GIT_URL	
Set environment variables : Body : Start - (1 min 10 sec in block)		
Set environment variables : Start - (1 min 10 sec in block)	JOB_OPTIONS, MAVEN_LOCAL_REPO	
Set environment variables : Body : Start - (1 min 10 sec in block)		

At least one “stage” section must be defined on the “stages” section of the pipeline. Each stage contains the work that the pipeline will execute. Stages must be named accordingly since Jenkins will display each of them on its interface, as shown here:

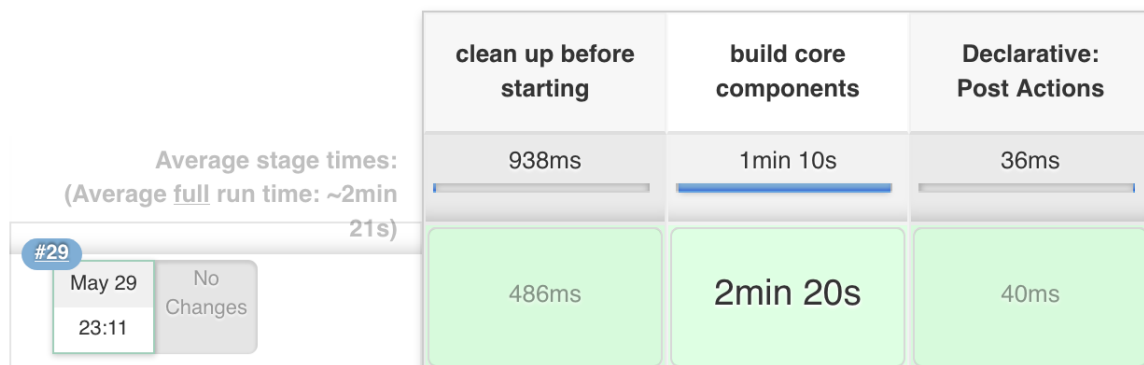
Pipeline gCube-Release

Generate gCube releases by building all the jobs.



[Recent Changes](#)

Stage View



The Release Manager has to work on the Jenkins Pipeline definition (see section 4.1) according to each release's requirements. Here are some expected activities:

- add/remove groups;
- add/remove jobs to/from groups;
- keep the agent label in sync with Jenkins config;
- keep Maven JDK in sync with Jenkins config.

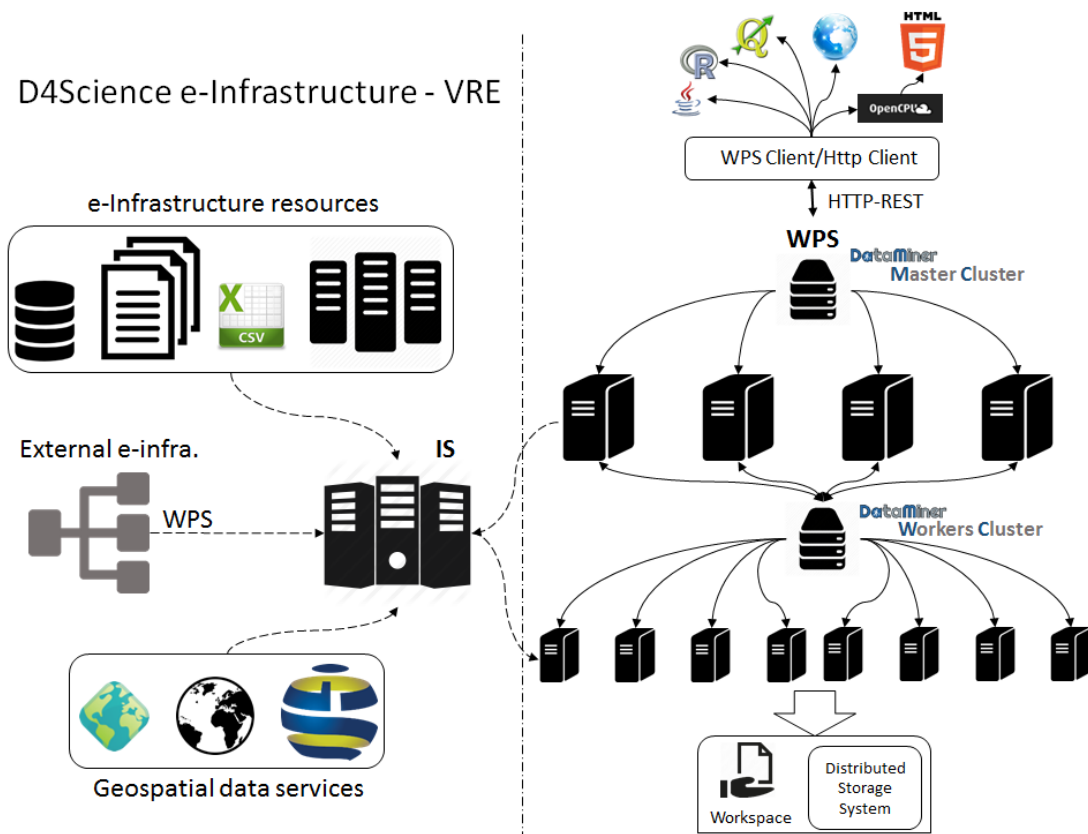
4.3.3 Maven Release plugin

Unlike other projects, gCube does not use the Maven Release plugin to release its artifacts. The simple explanation is that the plugin is not of any use to gCube. The plugin is designed for **SINGLE** artifact releases, not for multi-project releases like gCube. It also assumes that a human intervention during the release process (for instance, it prompts for the tagname) for each component, obviously not feasible in a gCube release.

At the time we are setting up this release process, it would be an over-complication to configure each Jenkins project with an invocation to the plugin and parametrize each build process to activate the plugin during the release build. This can be investigated in the future.

5 Software Methods Provision and Integration

The data processing platform (named DataMiner) is an open-source computational system based on the gCube system. This platform is fully integrated with the D4Science e-Infrastructure, and has been conceived to meet the Open Science paradigm requirements, thus to promote collaborative experimentation and open publication of scientific findings, while tackling Big Data challenges. DataMiner is able to interoperate with the services of the D4Science e-Infrastructure, and uses the Web Processing Service (WPS) standard to publish the hosted processes. Further, it saves the computational provenance of an executed experiment using the Prov-O standard. DataMiner implements a Cloud computing Map-Reduce approach and is able to process Big Data and save outputs onto a collaborative experimentation space. This space allows users to share computational information with other colleagues. This service was conceived to execute processes provided by communities of practice in several domains, leveraging integration effort at the same time. The DataMiner deployment is fully automatic through ANSIBLE scripts and is spread across different machines providers, including the Italian Garr network.



Architecture of the DataMiner data processing system

The DataMiner (DM) [2] architecture is made up of two sets of machines (clusters) that operate in a Virtual Research Environment [3]: the Master and the Worker cluster. In a typical deployment scenario, the Master cluster is made up of a number of powerful machines (e.g. Ubuntu 18 x86 64 with 16 virtual CPUs, 32 GB of random-access memory, 100 GB of disk) managed by a load balancer that distributes the requests uniformly to the machines. Each machine is endowed with a DM service that communicates with the D4Science Information

System (IS), i.e. the central registry of the e-Infrastructure resources, to notify its presence and capabilities. The balancer is indexed on the IS and is the main access point to interact with the DMs. The machines of the Worker cluster have the same computational power and serve Map-Reduce computations. DM is based on the 52North WPS implementation, but extends it to meet D4Science e-Infrastructure requirements. It is developed with Java and the Web service runs on an Apache Tomcat instance endowed with gCube system libraries. Further, it offers a development framework to integrate new algorithms and interact with the e-infrastructure.

Using the WPS standard in a Cloud computing system allows a number of thin clients to use the processes. The DataMiner services use the security services of the D4Science e-Infrastructure and require a user token to be provided for each operation. This token is passed via basic HTTPS-access authentication, which is supported by most WPS and HTTP(S) clients. The token identifies both a user and a Virtual Research Environment and this information is used by DM to query the IS about the capabilities to be offered in that VRE, i.e. the processes the user will be able to invoke with that authorization.

The DataMiner computations can take inputs from the D4Science Workspace. Inputs can also come from Workspace folders shared among several users. This fosters collaborative experimentation already at the input selection phase. Inputs can also come from external repositories, because a file can be provided either as a HTTP link or embedded in a WPS execution request. The outputs of the computations are written onto the D4Science Distributed Storage System and are immediately returned to a client at the end of the computation. Afterwards, an independent thread also writes this information on the Workspace. Indeed, after a completed computation, a Workspace folder is created which contains the input, the output, the parameters of the computation, and a provenance document summarizing this information. This folder can be shared with other people and used to execute the process again. Thus, the complete information about the execution can be shared and reused.

DataMiner can also import processes from other WPS services. If a WPS service is indexed on the IS for a certain VRE, its processes descriptions are automatically harvested, imported, and published among the DM capabilities for that VRE. During a computation, DM acts as a bridge towards the external WPS systems. Nevertheless, DM adds provenance management, authorization, and collaborative experimentation to the remote services.

5.1 Functional specifications

DataMiner satisfies functional specifications related to the processing of a large variety of data types (including geospatial data) in the wide context of Big Data processing and Open Science. Indeed, several computational systems exist, also used by e-Infrastructures, that typically parallelise the computation on several available cores/processors or machines of the e-Infrastructure. Nevertheless, DataMiner also satisfies new requirements requested by new Science paradigms, which include:

- Publishing local-machine processes, provided by a community of practice (e.g. scripts, compiled programs etc.), as-a-Service;
- Managing several programming languages;
- Interoperate with other services of an e-Infrastructure, possibly through a standard representation of the processes and of their parameters;

- Saving the “provenance” of an executed experiment, i.e. the set of input/output data, parameters, and metadata that would allow to reproduce and repeat the experiment;
- Supporting data and parameters sharing through collaborative experimental spaces;
- Being economically sustainable, e.g. easy to install and deploy on several partners machines;
- Supporting security and accounting facilities;
- Managing and analysing Big Data;
- Designing and executing Workflows that combine different processes published as services.

DataMiner was conceived to satisfy the reported requirements. One major advantage is that all the DM services publish their capabilities using a standard, which enhances the interoperability with other external services and software, with respect to using custom clients. Further, the DM clusters are managed by fast load balancers that are able to dynamically add machines and to ignore them when offline. Since the Worker nodes are exact replicas of the Master nodes, the Worker cluster can be used directly from clients and fosters alternative usages of the Cloud computing system. For example, external users of D4Science (authorised with proper tokens) may also implement their own Cloud computations by invoking the Worker cluster in custom workflows. Another DM feature is that it can interact with data preparation and harmonisation services. This speeds up the typical time consuming phase of data preparation for an experiment. Further, providing a shared experimentation area allows reusing the results of processes and also fosters multidisciplinary experiments. Users could also be services or external machines (e.g. sensors) that produce experimental data at different frequencies and time scales, while other processes analyse these data and take decisions. The same facilities are automatically offered to desktop software supporting WPS. Further, generating and storing provenance information improves the possibility to repeat and reproduce an experiment executed by other scientists. Finally, since processes and service installation is fully automatic through ANSIBLE scripts, it is easy to deploy DataMiner on a number of machines providers. The hosted processes currently hosted by DataMiner are written with the R, Java, Fortran, Linux-compiled, .Net, Octave, Knime, and Python programming languages and have been provided by developers with heterogeneous expertise (e.g. biologists, mathematicians, agronomists, physicists, data analysts etc.).

a. The 'Feed Forward' process configuration window. It shows parameters for training a generic Feed Forward Artificial Neural Network. Key fields include TrainingDataset (A2Dport.csv), TargetColumns (1), LearningRate (0.01), and HiddenLayers (20). A 'Start Computation' button is visible at the bottom.

b. The 'List of Computations' table, showing a summary of executed processes:

Name	Created	operator_name	start_date	end_date	status
FEED_FORWARD_A_N_N_DISTRIBUT	16 Nov 15:21 PM 2016	FEED_FORWARD_A_N_N_DISTRIBUT	16/11/2016 15:20:53	16/11/2016 15:20:58	completed
FEED_FORWARD_A_N_N_DISTRIBUT	16 Nov 15:20 PM 2016	FEED_FORWARD_A_N_N_DISTRIBUT	16/11/2016 15:20:45	16/11/2016 15:20:49	completed
FEED_FORWARD_A_N_N_DISTRIBUT	16 Nov 12:04 PM 2016	FEED_FORWARD_A_N_N_DISTRIBUT	16/11/2016 12:03:53	16/11/2016 12:03:58	completed
FEED_FORWARD_A_N_N_DISTRIBUT	16 Nov 12:03 PM 2016	FEED_FORWARD_A_N_N_DISTRIBUT	16/11/2016 12:03:01	16/11/2016 12:03:06	completed

c. The 'Computation Report' for a specific computation, showing detailed parameters and results. It includes fields like Input Dataset, Output Dataset, and a list of parameters used in the computation.

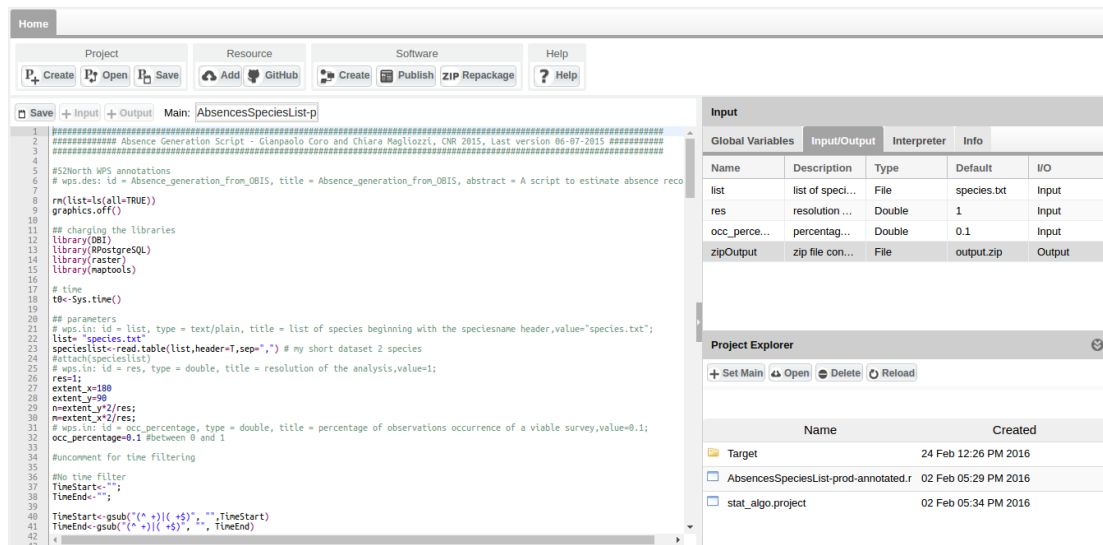
d. The 'List of Datasets' table, showing the datasets used in the computations:

Name	Created	computation_id	data_description	data_type	operator_name	VRE
Dataset_FEED_FORWARD_A_N_N_DISTRIBUT	16 Nov 15:21 PM 2016	FEED_FORWARD_A_N_N_DISTRIBUT	Output table for http link to a table in UTF-8 encoding following this template: (TESTSET) http://goo.gl/2HxK6	text/csv	FEED_FORWARD_A_N_N_DISTRIBUT	Infrastructure: evo/gCubeApps/Biodivers
Dataset_FEED_FORWARD_A_N_N_DISTRIBUT	16 Nov 15:20 PM 2016	FEED_FORWARD_A_N_N_DISTRIBUT	Output table for http link to a table in UTF-8 encoding following this template: (TESTSET) http://goo.gl/2HxK6	text/csv	FEED_FORWARD_A_N_N_DISTRIBUT	Infrastructure: evo/gCubeApps/Biodivers

Interface of the gCube DataMiner system

DataMiner offers a Web GUI to the users of a VRE. On the left panel (a), the GUI presents the list of capabilities available in the VRE, which are semantically categorised (the category is indicated by the process provider). For each capability, the interface calls the WPS *DescribeProcess* operation to get the descriptions of the inputs and outputs. When a user selects a process, in the right panel the GUI on-the-fly generates different fields corresponding to the inputs. Input data can be selected from the Workspace (the button associated to the input opens the Workspace selection interface). The “Start Computation” button sends the request to the DM Master cluster, which is managed as explained in the previous section. The usage and the complexity of the Cloud computations are completely hidden to the user, but the type of the computation is reported as a metadata in the provenance file. In the end, a view of the Workspace folders produced by the computations is given in the “Check the Computations” area (b), where a summary sheet of the provenance of the experiment can be obtained (“Show” button, c). From the same panel, the computation can be also re-submitted. In this case, the Web interface reads the Prov-O XML information associated to a computation and rebuilds a computation request with the same parameters. The computation folders may also include computations executed and shared by other users. Finally, the “Access to the Data Space” button allows the user to obtain a list of the overall input and output datasets involved in the executed computations (d), with provenance information attached that refers to the computation that used the dataset.

5.2 Software and Algorithms Integration

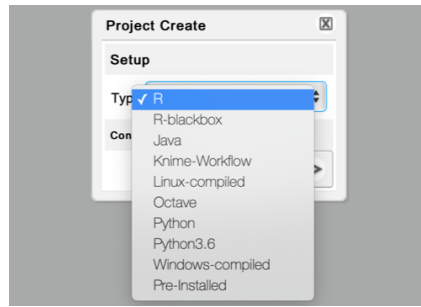


Interface to import an R process on DataMiner

Prototype scripting is the basis of most models in environmental sciences. Scientists making prototype scripts (e.g. using R and Matlab) often need to share results and make their models used also by other scientists on new data. To this aim, one way is to publish scripts as-a-Service, possibly under a recognized standard (e.g. WPS). The Statistical Algorithms Importer (SAI¹) is an interface that allows scientists to easily and quickly import scripts onto DataMiner. DataMiner in turn publishes these scripts as-a-Service and manages multi-tenancy and concurrency. Additionally, it allows scientists to update their scripts without following long software re-deploying procedures each time. In summary, SAI produces processes that run on the DataMiner Cloud computing platform and are accessible via the WPS standard.

The SAI interface for R scripts resembles the R Studio environment, a popular IDE for R scripts, in order to make it friendly to script providers, whereas the interface for software written in other programming languages does not allow to edit the main script. However, SAI provides support for scripts implemented in several languages as shown in the following picture.

¹ Described in Coro, G., Panichi, G., & Pagano, P. (2016). A Web application to publish R scripts as-a-Service on a Cloud computing platform. *Bollettino di Geofisica Teorica ed Applicata*, 57, 51-53. With a user guide available at [https://wiki.gcube-system.org/gcube/Statistical Algorithms Importer](https://wiki.gcube-system.org/gcube/Statistical_Algorithms_Importer)



The *Project* button allows creating, opening and saving a working session. A user uploads a set of files and data on the workspace area (lower-right panel). Upload can be done by dragging and dropping local desktop files. As next step, the user indicates the “main script”, i.e. the script that will be executed on DataMiner and that will use the other scripts and files.

For R scripts integration, after selecting the main script, the left-side editor panel visualises it with R syntax highlighting and allows modifying it.

Afterwards, the user indicates the input and output of the script by highlighting variable definitions in the script and pressing the *+Input* (or *+Output*) button. In the case of other programming languages than R, the Input and Output variables should be manually specified directly in the Input/Output panel.

For R scripts, SAI also supports WPS4R annotations inside the script to automatically generate inputs and outputs. Other tabs in this interface area allow setting global variables and adding metadata to the process. In particular, the *Interpreter* tab allows indicating the R interpreter version and the packages required by the script and the *Info* tab allows indicating the name of the algorithm and its description. In the *Info* tab, the user can also specify the algorithm’s name and category.

Once the metadata and the variables information has been fulfilled, the user can create one DataMiner as-a-Service version of the script by pressing the *Create* button in the Software panel. The term “software”, in this case indicates a Java program that implements an as-a-Service version of the user-provided scripts. The Java software contains instructions to automatically download the scripts and the other required resources on the server that will execute it, configure the environment, execute the main script and return the result to the user. The computations are orchestrated by the DataMiner computing platform that ensures the program has one instance for each request and user. The servers will manage concurrent requests by several users and execute code in a closed sandbox folder, to avoid damage caused by malicious code.

Based on the SAI Input/Output definitions written in the generated Java program, DataMiner automatically creates a Web GUI. By pressing the *Publish* button, the application notifies DataMiner that a new process should be deployed. DataMiner will not own the source code, which is downloaded on-the-fly by the computing machines and deleted after the execution.

This approach meets the policy requirements of those users who do not want to share their code. The *Repackage* button re-creates the software so that the computational platform will be using the new version of the script. The repackaging function allows a user to modify the script and to immediately have the new code running on the computing system. This approach separates the script updating and deployment phases, making the script producer completely

independent on e-Infrastructure deployment and maintenance issues. However, deployment is necessary again whenever Input/Output or algorithm's metadata are changed.

To summarise, the SAI Web application relies on the D4Science e-Infrastructure and enables a software (R, Java, Fortran, Linux-compiled, .Net, Octave, Knime, Python), provided by a community of practice working in a VRE, with as-a-Service features. SAI reduces integration time with respect to direct Java code writing. Additionally, it adds (i) multi-tenancy and concurrent access, (ii) scope and access management through Virtual Research Environments, (iii) output storage on a distributed, high-availability file system, (iv) graphical user interface, (v) WPS interface, (vi) data sharing and publication of results, (vii) provenance management and (viii) accounting facilities.

6 Tools Provision and Integration

The tools provision and integration of software deployed as service follows:

6.1 SmartGears

SmartGears is a set of Java libraries that turn Servlet-based containers and applications into gCube resources, transparently.

In this section, we introduce SmartGears² and explain how it is an improvement over existing gCube solutions. The discussion is relevant to node and infrastructure managers, who perform and maintain SmartGears installations, and to developers, who package or write software for a gCube infrastructure.

A piece of software is an infrastructure resource (the so-called Software as Resource, SaR) if we can manage it in the D4Science infrastructure. This means that we can do a number of things with the software, including:

- discover where it is deployed in the infrastructure, so as to use it without hard coded knowledge of its location. For this, we need to describe each and every software deployment, and publish these descriptions, or profiles, in the infrastructure;
- monitor and change the status of its deployments, so as to take actions when they are not in an operational status (e.g. redeploy the software, or at least prevent discovery and usage of the deployments). For this, we need to track their current status, report it in the profiles we publish, and republish the profiles when the status changes;
- dedicate its deployments to certain groups of users, in the sense that only users in those groups can use them. We can change the sharing policies of individual deployments at any time, i.e. share them across more or less groups. We can also grant different privileges to different types of users within given groups.

² More details about SmartGears can be found in the set of dedicated wiki pages at the address: <https://wiki.gcube-system.org/gcube/SmartGears>.

Publication, discovery, lifecycle management, controlled sharing are the pillars of the resource management. Yet relying on humans to compile deployment profiles, publish them in the infrastructure, keep track and change the status of deployments, or enforce sharing policies is all but practical. In some cases, it is downright impossible. We need instead automated solutions that live alongside each and every deployment and help us turn it into a resource we can manage. SmartGears is one such solution.

We focus on software that can be used over the network, such as distributed applications and network services. Software deployments then correspond to software endpoints.

Typically, software endpoints run within containers and, in D4Science, containers can be resources in their own right, the so-called gCube Hosting Nodes (gHNs).

Managing gHNs is a way to manage multiple endpoints simultaneously (e.g. to deactivate a gHN means to deactivate a set of endpoints at once). Equally, it is a way to manage underlying hardware resources (e.g. dedicate a gHN to selected groups of users).

This is a notion of "Container-as-Resource" (CaR), and it raises the same requirements as SaR, including publication and discovery, lifecycle management, and controlled sharing. SmartGears helps us meet these requirements too, i.e. turns containers as well as the endpoints therein into gCube resources.

SmartGears is not a development framework. Rather, SmartGears is invisible to the software, not part of its stack at all. As a result, any software can run in the infrastructure: SaR becomes a nature that software acquires at runtime.

Indeed, SmartGears has few requirements of the software. All we ask of software is that it is based on the Servlet specifications, which define the hooks that we need to track its lifecycle and its use. The software is thus a Web Application and may more specifically be a Soap service, a Rest service, or a generic Web Application. It may adopt different standards and technologies (e.g. JAX-RPC, JAX-WS, JAX-RS, but also Dependency Injection technologies, persistence technologies, etc.). And of course, it may run in any container that is Servlet-compliant (Web Containers, Application Servers).

Finally, the evolution of SmartGears is inconsequential for the software: most of the APIs of SmartGears remain private to SmartGears.

Containers and applications need a minimal set of requirements before SmartGears can turn them into gCube resources:

- containers must comply with version 3 of the Servlet specifications;
- applications must include at least one gcube-app.xml configuration file alongside their deployment descriptor (i.e. under /WEB-INF);

In addition:

- node managers must define a GHN_HOME environment variable that resolves to a location where SmartGears can find a container.xml configuration file.

Starting from version 3, the Servlet specifications allow SmartGears to intercept relevant events in the lifecycle of individual applications whilst being shared across all applications, in line with the deployment scheme of SmartGears. In particular, the specifications introduce a ServletContextInitializer interface that SmartGears implements to be notified of application startup. The specifications also allow programmatic registration of filters and servlets,

which SmartGears uses to transparently manage applications without the need of additional configuration in their web.xml descriptor.

Configuration is thus limited to WEB-INF/gcube-app.xml and \$GHN_HOME/container.xml, which provide the configuration of, respectively, the application and the container as gCube resources. Details about their contents are available in the gCube Wiki [Appendices](#).

Smartgears is distributed as a tarball that contains the libraries, scripts, and configuration files required to install Smartgears in a given container, and to maintain the installation over time. Instructions on how to download, install and maintain Smartgears are available in the [SmartGears_Web_Hosting_Node_\(wHN\)_Installation](#).

6.2 OAuth2.0

By means of the OAuth 2.0 protocol (authorised) third party applications can operate on a user's behalf over the D4Science infrastructure (while protecting the member's credentials). For more information about the OAuth authorization framework please visit the official OAuth site³. For technical details also see the OAuth 2.0 RFC⁴. In the following, the steps needed to authorize third party applications to operate on a user's behalf and the D4Science infrastructure are explained.

This exploitation case makes it possible to integrate in the infrastructure services, tools, and applications that are not deployed on SmartGears powered containers.

More details about how the OAuth 2.0 service work can be exploited in D4Science can be found in the set of dedicated wiki pages at the address:

<https://wiki.gcube-system.org/gcube/OAuth2.0>

³ <https://oauth.net/2/>

⁴ <https://tools.ietf.org/html/rfc6749#7C>

7 References

1. *Assante, M., Candela, L., Castelli, D., Cirillo, R., Coro, G., Frosini, L., Lelii, L., Mangiacrappa, F., Marioli, V., Pagano, P., Panichi, G., Perciante, C., Sinibaldi, F.* (2019). **The gCube System: Delivering Virtual Research Environments as-a-Service**. Future Generation Computer Systems, Vol. 95
2. *Coro, G., Panichi, G., Scarponi, P., & Pagano, P.* (2017). **Cloud computing in a distributed e-infrastructure using the web processing service standard**. Concurrency and Computation: Practice and Experience, 29(18), e4219
3. *L. Candela, D. Castelli, P. Pagano* (2013). **Virtual Research Environments: An Overview and a Research Agenda**. Data Science Journal, Vol. 12
4. *M. Assante, L. Candela, D. Castelli, R. Cirillo, G. Coro, L. Frosini, L. Lelii, F. Mangiacrappa, P. Pagano, G. Panichi, F. Sinibaldi.* (2019). **Enacting open science by D4Science**, Future Generation Computer System