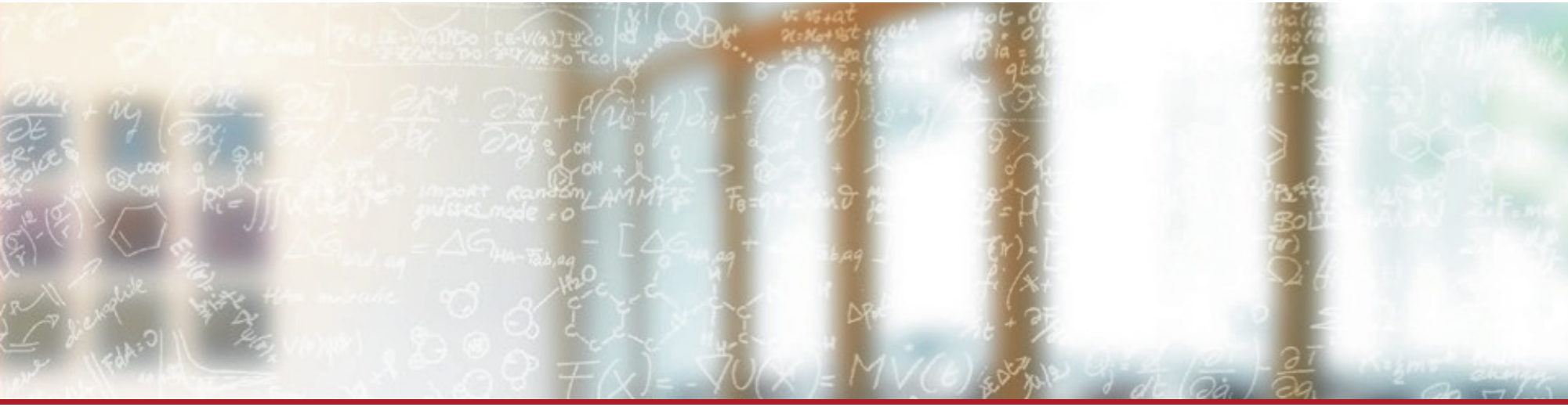
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



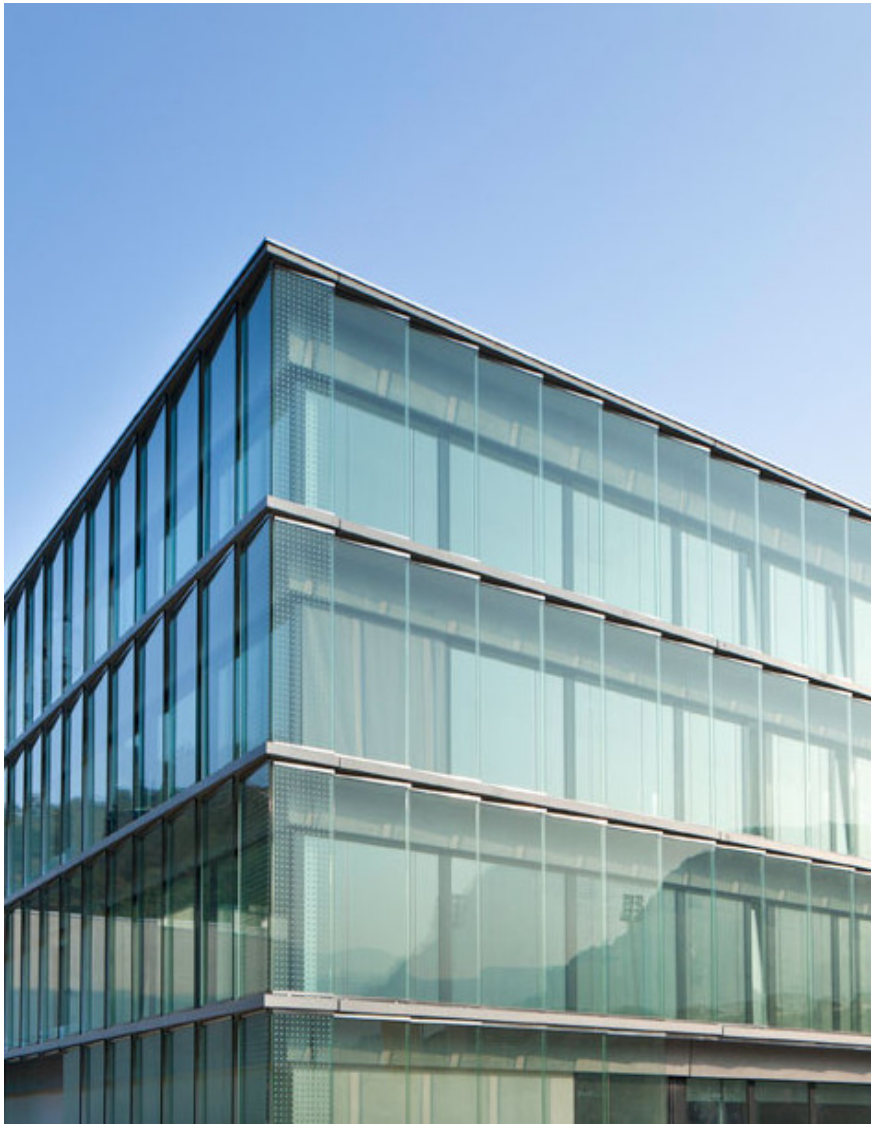
Continuous Integration with Jenkins

CSCS Software Management Course

Theofilos Manitaras - ETH Zurich/CSCS

May 14, 2019

Continuous Integration with Jenkins



- Continuous Integration & Jenkins
  - Jenkins in a Nutshell
  - Piepeline-as-code
  - Pipeline Steps
  - Jenkins Plugins
  - Jenkins Credentials
  - Triggering a build
  - Live Demonstration
- Jenkins at CSCS
  - Accessing the CSCS CI
  - Using the CSCS CI service
  - Best practices
  - Github pull request builder
  - Polling a remote repository

Jenkins in a Nutshell

What is CI and why Jenkins?

- Continuous integration(CI) is a software development practice according to which code changes are continuously integrated into a shared code basis.
- Each contribution to the main code repository is tested by an automated build.
- CI helps to catch errors sooner and avoid code conflicts in the future.
- Jenkins is an open source automation server written in Java which is used extensively for CI.
- It's ease of use and extensibility via plugins make it ideal for various development scenarios.
- In this presentation when talking about Jenkins, we refer to Jenkins 2, the latest major version.

Typical CI Workflow

In order for CI to make sense, the software project has to be version controlled (e.g via git). The typical CI workflow is described in the following steps:

1. Each developer works on his/her own fork of the software project.
2. A change to the code base is submitted via a Pull/Merge Request to the master repository/branch.
3. The above change triggers an automated CI job (e.g via Jenkins) which runs the unittests and/or performs further testing to make sure that there are no bugs introduced.
4. The results of the CI are reported. Optionally, the developers are notified regarding the CI job output.
5. The proposed change is merged to the master branch either automatically or by one of the repository administrators.

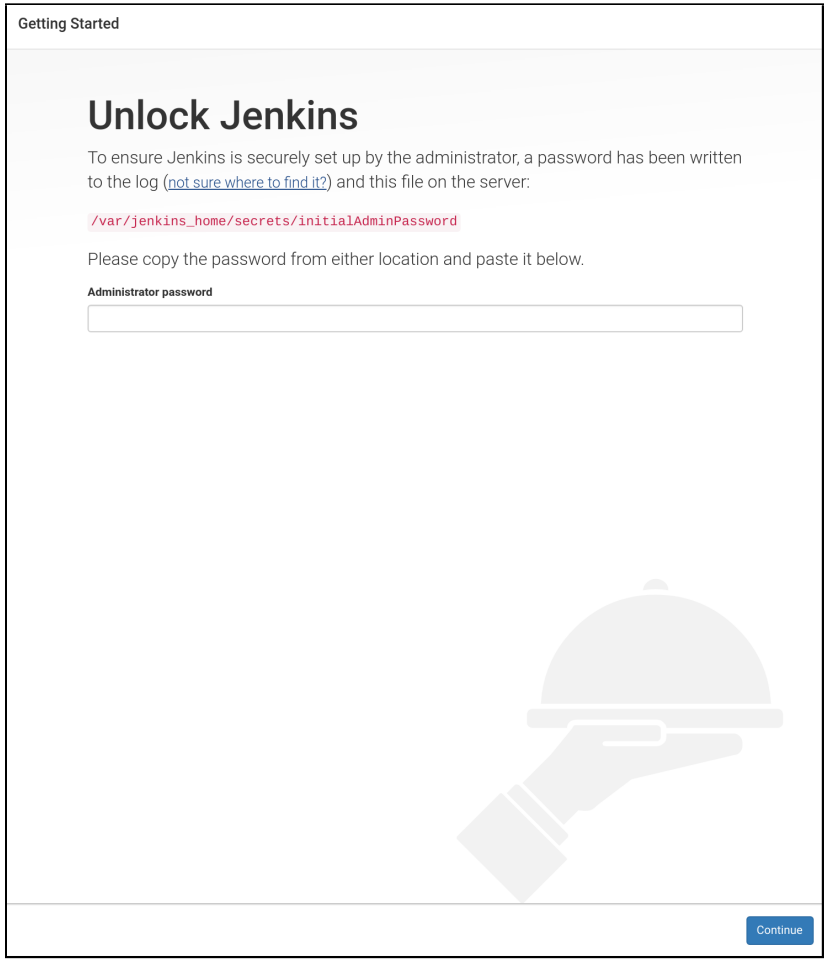
Installing/Running Jenkins

The various ways to install/run Jenkins on Linux which are described here. The three easier alternatives are the following:

1. Pull the jenkins image from Dockerhub and run using docker.
2. Download the Jenkins Web Application Archive (WAR) and run directly (Java 8 is needed).
3. Use the package manager of your Linux distribution.

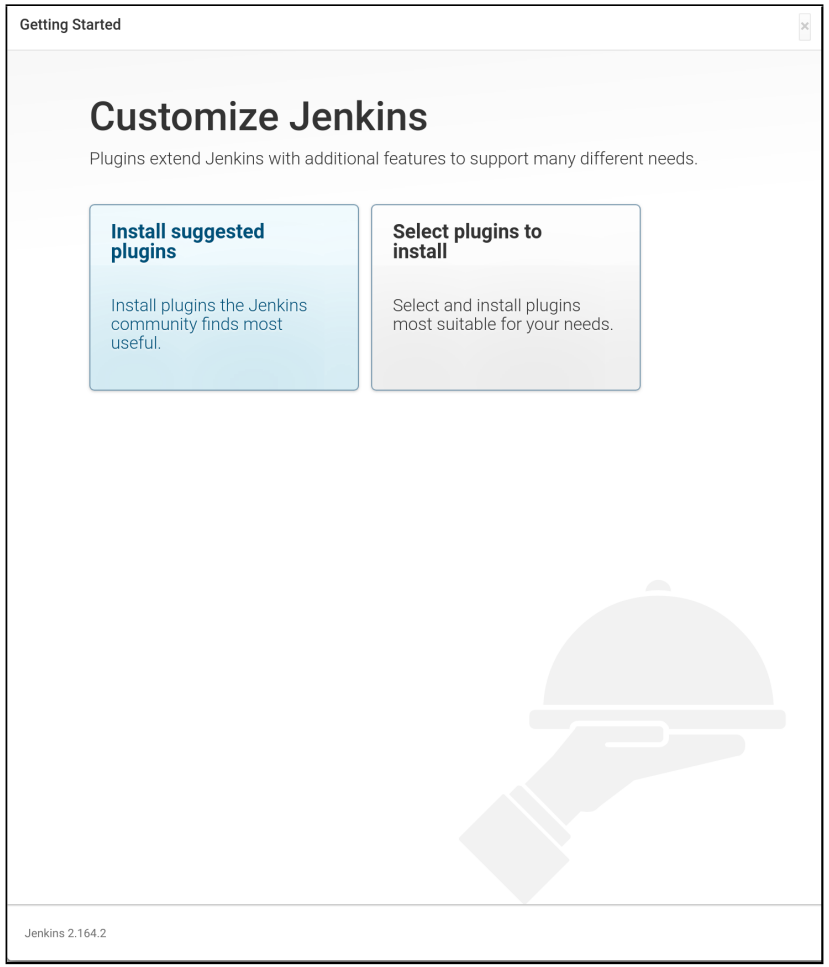
Accessing the Web UI for the 1st time

When Jenkins is installed, and the web interface is accessed for the first time, the following screen appears:



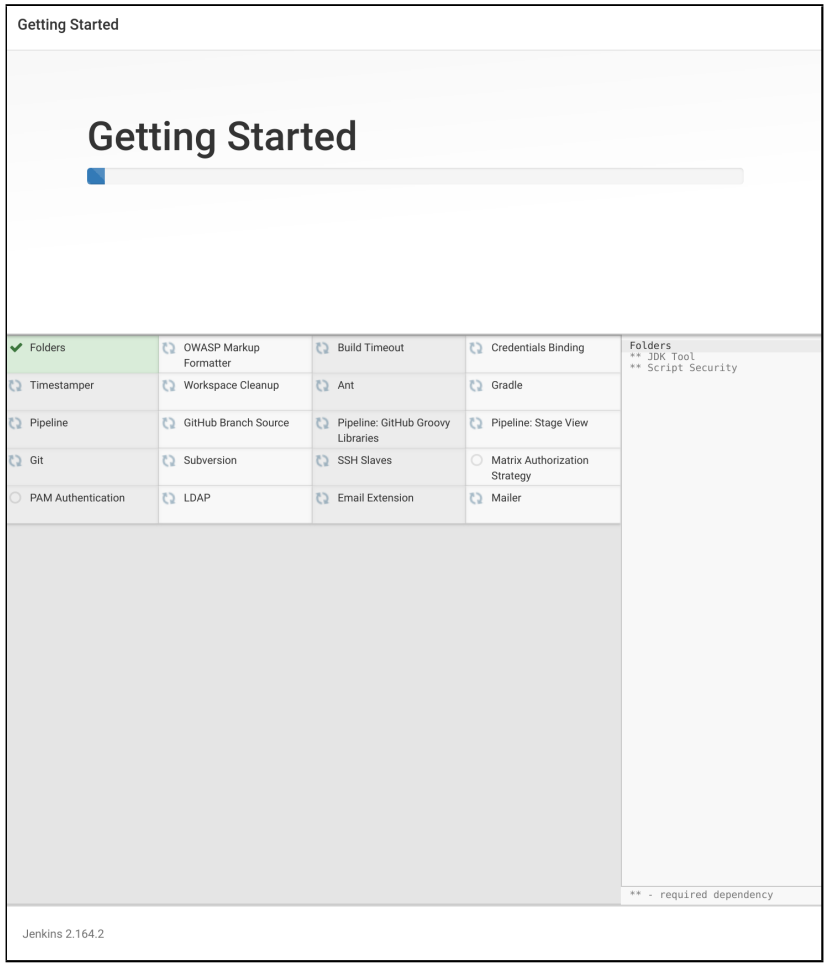
Installing the suggested plugins(1/2)

After giving the unlock password, Jenkins asks for installation of suggested plugins:



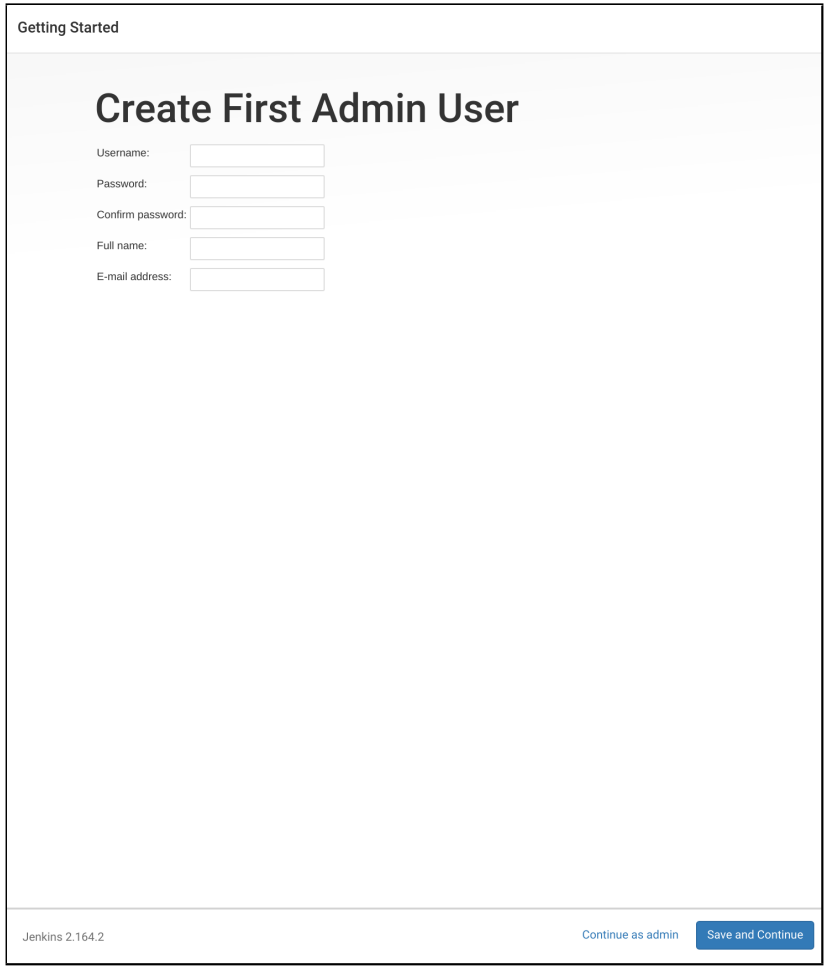
Installing the suggested plugins(2/2)

The plugin installation should look as follows:



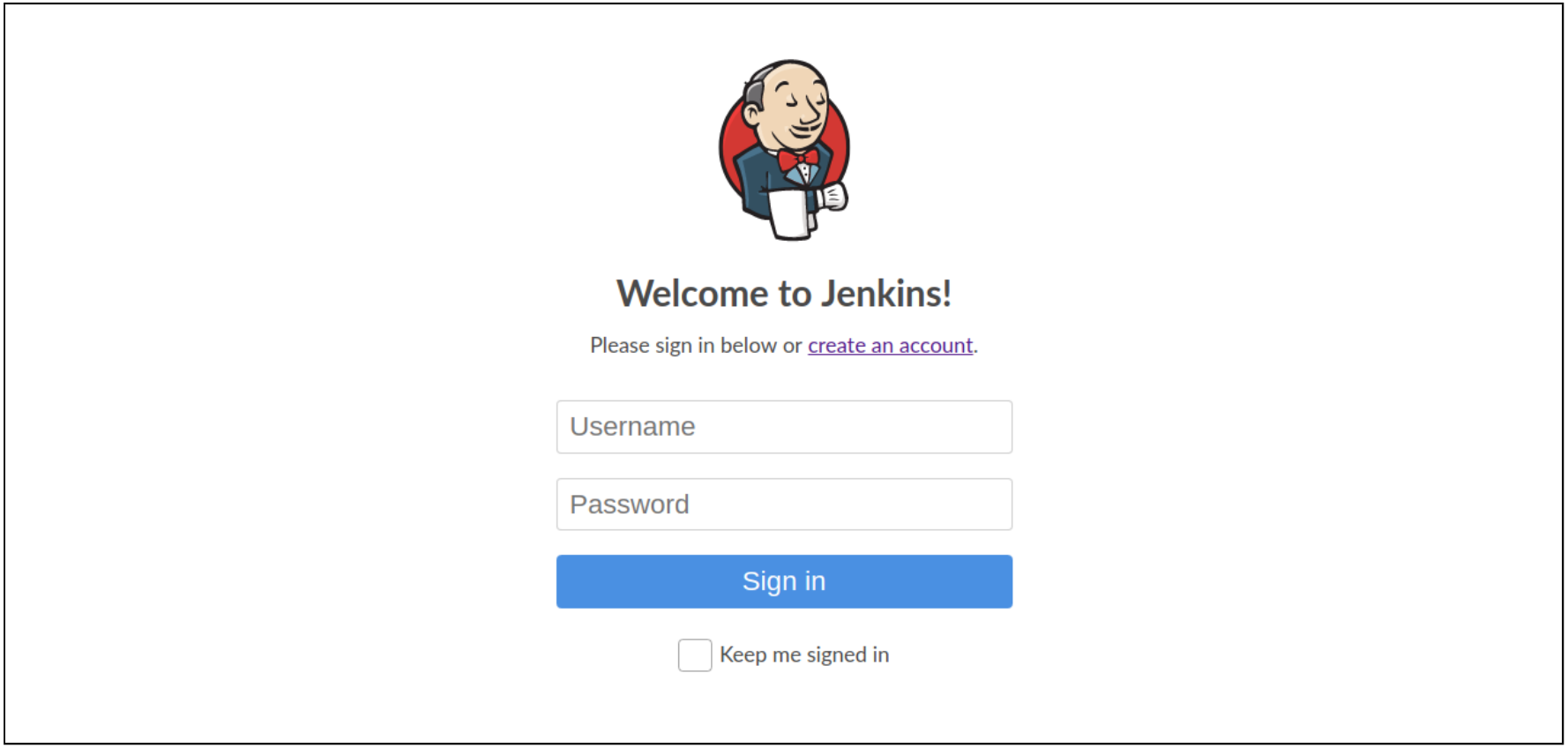
Creating an admin account

Finally an admin account has to be created:

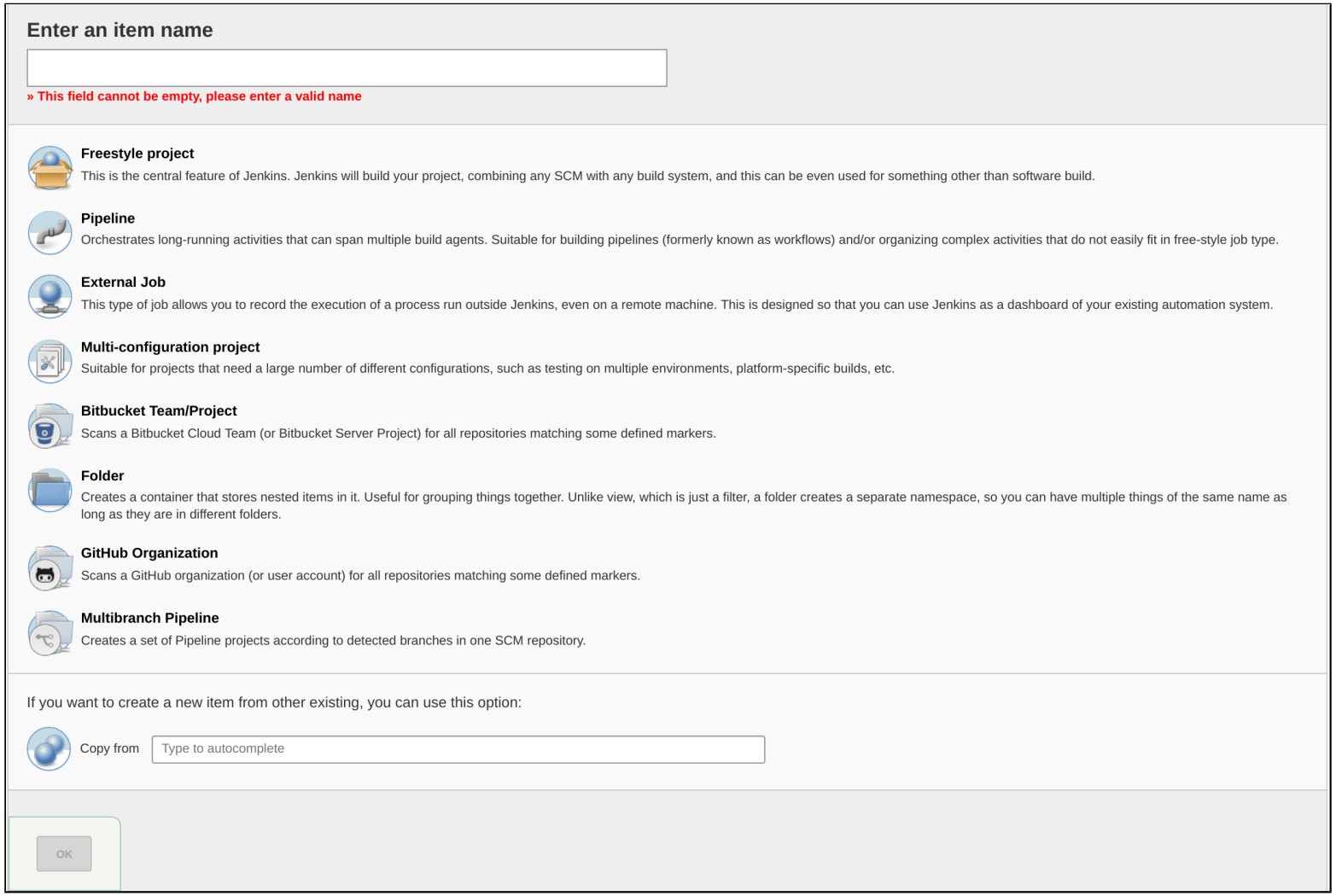


Logging to Jenkins

When connecting to Jenkins you first have to login:



Types of Jenkins Projects



Types of Jenkins Projects

In this course, we are going to cover the following Jenkins project types:

- **Folder:** Allows to group Jenkins projects into a directory type hierarchy.
- **Pipeline:** A general pipeline-as-code based project. A Jenkinsfile can be used from SCM or the pipeline can be directly written in the corresponding box.
- **Github Organization:** This project type allows to easily incorporate Github projects from a Github Organization which already contain Jenkinsfiles. Several options controlling the repository/branch discovery are available.

Pipeline-as-code

The Jenkinsfile

The **Jenkinsfile** is at the core of the Jenkins pipeline-as-code approach. It serves a double purpose:

1. It describes in a Domain Specific Language (DSL) the necessary actions to be performed by Jenkins once a build is triggered.
2. The existence of a Jenkinsfile in a code repository and/or branch makes Jenkins aware that this repo/branch can be run by it.

A Jenkinsfile describing a pipeline can be written using two alternative syntaxes:

1. **Scripted Pipeline**: basically a Groovy based script describing the actions to be taken by Jenkins.
2. **Declarative Pipeline**: A more easy to follow declarative approach based on simple constructs specific to the Jenkins actions.

Structure of a Jenkinsfile

Depending of the flavour of the syntax you choose, the structure of the Jenkinsfile differs as follows:

Scripted Pipeline

```
stage("Stage 1"} {
    node("<mynode>") {
        <step 1>
          .
          .
          .
        <step n>
    }
}
  .
  .
  .
stage("Stage N"} {
    node("<mynode>") {
        <step 1>
           .
           .
           .
        <step n>
    }
}
```

Declarative Pipeline

```
pipeline {
    agent any
    stages {
        stage("Stage 1") {
            steps {
                <step 1>
                   .
                   .
                <step n>
            }
        }
        .
        .
        stage("Stage N") {
            steps {
                <step 1>
                   .
                   .
                <step n>
            }
        }
    }
}
```

Nodes/Agents

- The Jenkins **Master** is the system on which the Jenkins instance is running. It is not intended to run any jobs since this poses a security risk.
- A **Node** in Jenkins terms is any system that can execute Jenkins jobs. A remote machine, a VM, a Docker container are examples of Nodes.
- An **Agent** is in Declarative pipeline terminology any nonmaster system which executes Jenkins jobs.
- Nodes can be managed either by the Jenkins admin or users with the corresponding permissions via **Manage Jenkins** $\rightarrow$ **Manage Nodes**.

Pipeline Steps

The dir step

- The **dir** command can be used both in scripted/declarative pipelines as follows:

```
dir("<mydir>") {
    <step 1>
       .
       .
       .
    <step N>
}
```

- The steps inside the curly braces will be executed inside the ”mydir” directory which will be created by Jenkins if it does not exist.
- A useful command is also **deleteDir** which recursively deletes the current working directory contents. Using is inside a **dir** block, it is going to recursively delete the corresponding directory.

The withEnv step

- The **withEnv** step can be used both in scripted/declarative pipelines as follows:

```
withEnv(["ENV1=myenv1", "ENV2=myenv2"]) {
    <step 1>
       .
       .
       .
    <step N>
}
```

- The corresponding environment variables **ENV1, ENV2** are going to be defined for all the steps inside the curely braces of the **withEnv** block.
- The environment variables are given to **withEnv** using the Groovy syntax which defines a list as follows:

```
["ENV1=myenv1", ... , "ENVN=myenvn"]
```

The sh step

- The **sh** step is used to execute shell commands as follows:

Scripted Pipeline

```
node("<mynode>") {
    .
    .
    .
    sh """#!/usr/bin/bash -l
         <sh command 1>
         <sh command 2>"""
}
```

Declarative Pipeline

```
steps {
    .
    .
    .
    sh """#!/usr/bin/bash -l
         <sh command 1>
         <sh command 2>"""
}
```

- The first line of the **sh** script can be a shebang line to indicate the shell used to execute the corresponding commands

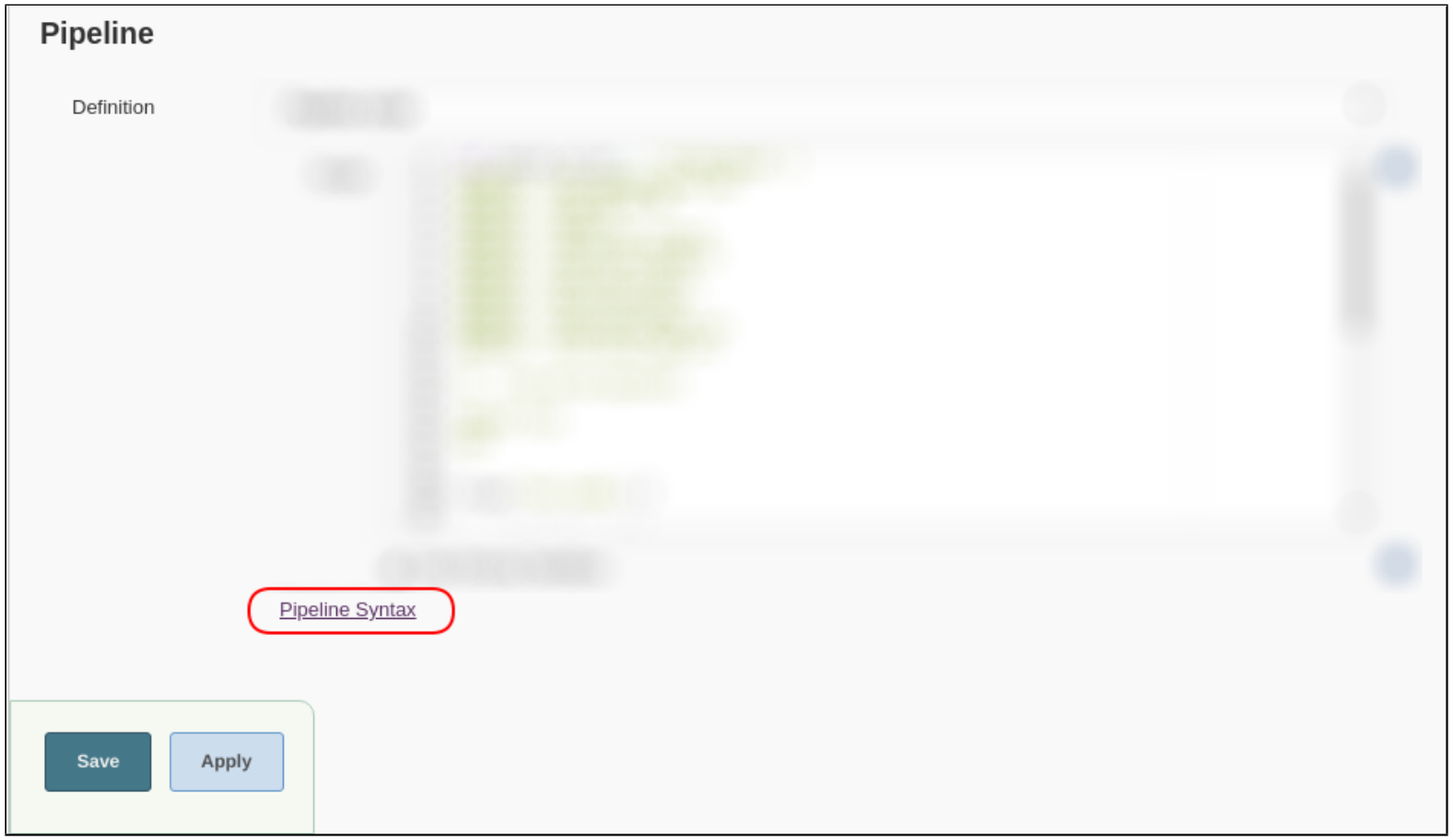
Post-processing

- In the declarative pipeline syntax post-build actions can be defined using the **post** section. Based on the build status, different actions can be performed as follows:

```
// End of the stages block
post {
    always {
        // Actions to perform always
    }
    changed {
        // Execute if the build status has changed from the previous build
    }
    success {
        // Execute in case of success
    }
    failure {
        // Execute in case of failure
    }
    unstable {
        // Execute in case of an unstable build
    }
}
```

- In scripted pipelines, the **post** functionality is achieved with a **try-catch-finally** block based on the value of the **currentBuild.result** variable.

Pipeline Snippet Generator (1/2)



Pipeline Snippet Generator (2/2)

Back
Snippet Generator
Declarative Directive Generator
Declarative Online Documentation
Steps Reference
Global Variables Reference
Online Documentation
IntelliJ IDEA GDSL

Overview

This **Snippet Generator** will help you learn the Pipeline Script code which can be used to define various steps. Pick a step you are interested in from the list, configure it, click **Generate Pipeline Script**, and you will see a Pipeline Script statement that would call the step with that configuration. You may copy and paste the whole statement into your script, or pick up just the options you care about. (Most parameters are optional and can be omitted in your script, leaving them at default values.)

Steps

Sample Step: archiveArtifacts: Archive the artifacts

Files to archive: test.out,test.err

Advanced...

Generate Pipeline Script

```
archiveArtifacts 'test.out,test.err'
```

Global Variables

There are many features of the Pipeline that are not steps. These are often exposed via global variables, which are not supported by the snippet generator. See the Global Variables Reference for details.

Jenkins Plugins

Useful Plugins

Jenkins offers it's great success in part because of it can be extended via plugins. Many of them are installed during the Jenkins installation. The following list contains some very useful ones:

- Role Strategy Plugin: Adds a new role-based strategy to manage users' permissions.
- Blue Ocean: Adds the new Blue Ocean interface.
- Github pull request builder plugin: Useful plugin to trigger & build Github Pull requests?
- Mock Agent Plugin: Creates dummy agents/nodes that run on the localhost. Very useful for testing and getting familiar with pipelines.

Plugin management is performed by navigating to **Manage Jenkins** $\rightarrow$ **Manage Plugins**.

Jenkins Credentials

Credential Types

Jenkins allows the creation and management of credentials through the Credentials Plugin (included with Jenkins installations). The following credential types are provided:

- Username with password
- Docker Host Certificate Authentication
- SSH username with private key
- Secret file
- Secret text
- Certificate

Credentials can be grouped together under a **Credential Domain**. The **Global** domain is available by default.

Trigerring a Jenkins Build

Build Triggers

The ways that a Jenkins build can be triggered are based on the project type and can be also changed based on the installed plugins. The most useful build triggers are the following:

- **Build after other projects are built:** This allows triggering a build based on the completion status of another project.
- **Build periodically:** This option triggers build under a specified schedule.
- **Poll SCM:** This option polls the SCM for code changes and triggers a build.
- **Trigger by Webhooks:** Trigger a build using webhooks (e.g Github webhooks).

Live Demonstration of a Jenkins Workflow

Jenkins at CSCS

CSCS CI service

- CSCS has recently started offering a CI service based on Jenkins to its users, as annouced at the PASC 2018 conference.
- Using the CSCS CI service, it is possible to test your applications on the HPC environment of Piz Daint.
- Jenkins slaves are configured so that the builds take place on the compute nodes of Piz Daint and therefore software is tested on the actual hardware/software.

Accessing to the CSCS CI

In order to be granted access to the CSCI service, a principal investigator (PI) responsible for a project including software development has to open a ticket at `help@cscs.ch` and make the request.
From the CSCS Jenkins instance side, the following apply:

- Each project is assigned a Jenkins folder with the same name on the Jenkins instance.
- The Jenkins jobs related to the project have to be created in the above folder.
- Credentials can be added to be used with version control systems, etc.
- Each project is assigned a Jenkins node(slave) to run the corresponding Jenkins jobs.
- A Jenkins user which is going to be used by the Jenkins node to access Piz Daint also added.

Using the CSCS CI service

Logging to the Jenkins Web interface (1/2)

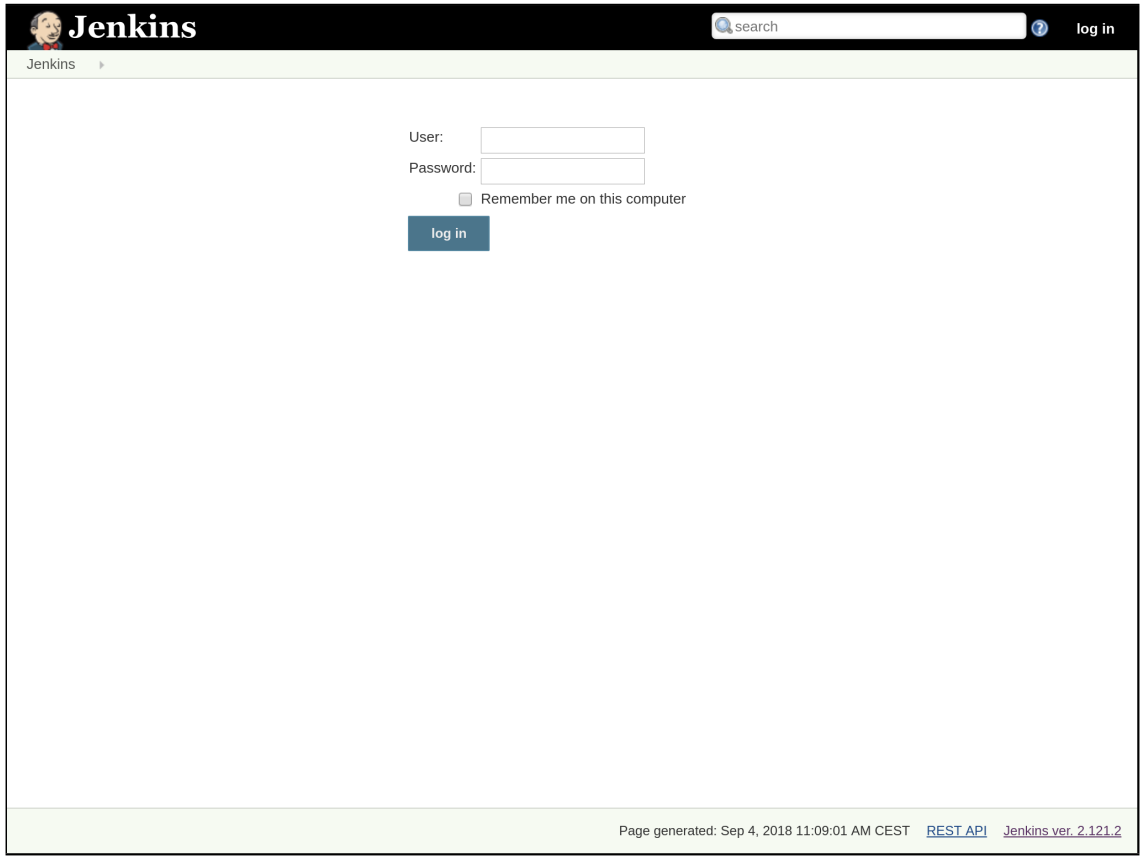
- The Jenkins web interface provided by CSCS is not accessible from the public web. In order to be able to access it, a local port forwarding must be performed with ssh. Thus the user has to forward a local port to **lisone.cscs.ch:443** via **ela.cscs.ch**.
- For Linux/Mac users this can be performed from the shell with the following command:

```
$ ssh -L 7000:lisone.cscs.ch:443 ela.cscs.ch
```

  This way, the user can access the web interface from a web browser visiting `https://ci.cscs.ch:7000/`. Note that the number of the local port (here 7000) is chosen by the user.

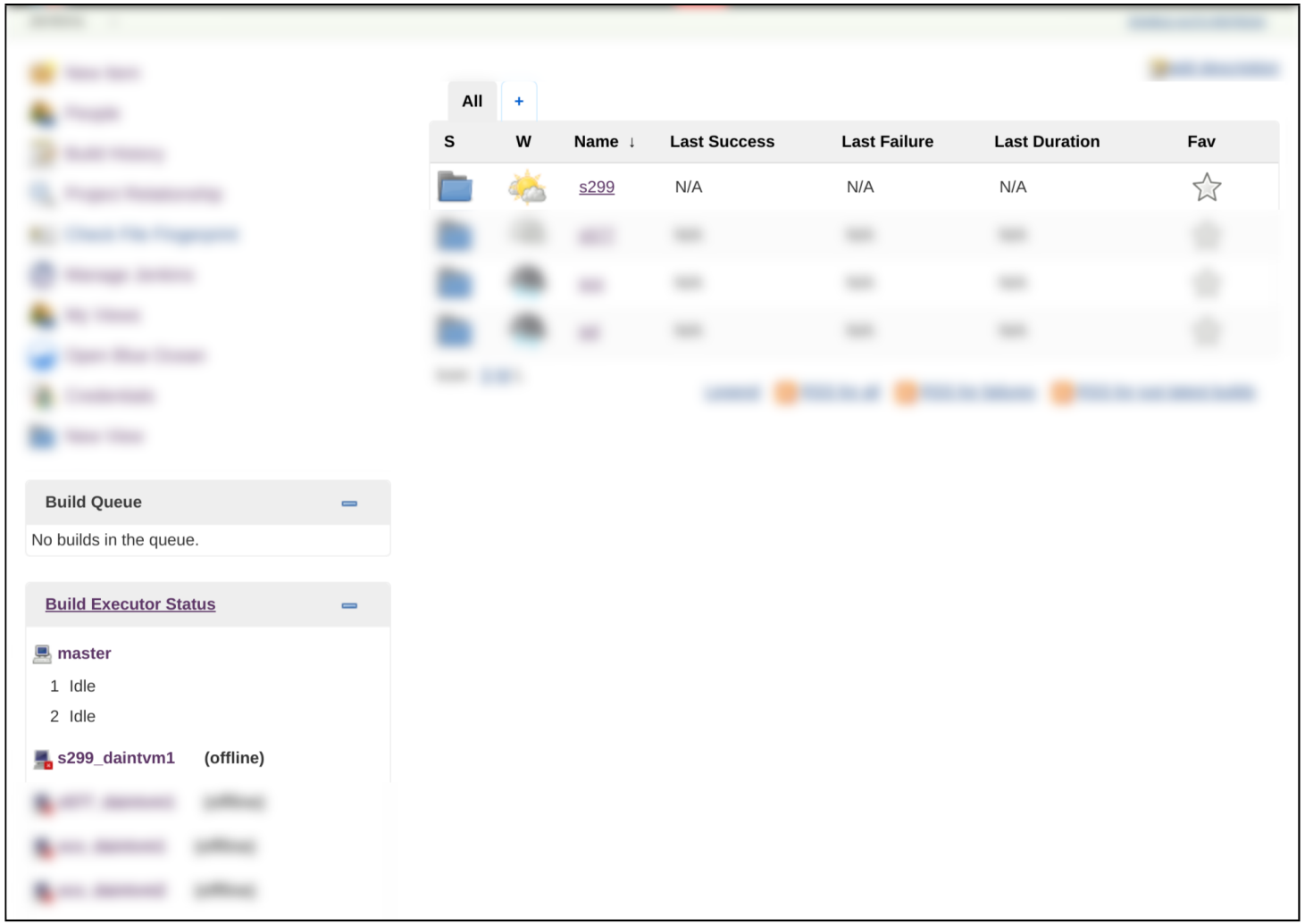
Logging to the Jenkins Web interface (2/2)

Use your CSCS account credentials to login.



Dedicated folder and Jenkins slave

Every project has access to a dedicated folder and a corresponding Jenkins slave which is a virtual machine and not a login node of Piz Daint.



Creating a new Jenkins project

Enter an item name

» This field cannot be empty, please enter a valid name

Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

Pipeline
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

External Job
This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system.

Multi-configuration project
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

Bitbucket Team/Project
Scans a Bitbucket Cloud Team (or Bitbucket Server Project) for all repositories matching some defined markers.

Folder
Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

GitHub Organization
Scans a GitHub organization (or user account) for all repositories matching some defined markers.

Multibranch Pipeline
Creates a set of Pipeline projects according to detected branches in one SCM repository.

Best practices

- Adopt the pipeline-as-code modern approach of Jenkins 2 and include the **Jenkinsfile** in your git remote.
- The actual build jobs have to be submitted via sbatch in the job queue. Use of srun is not allowed.
- The –wait option should be used when submitting sbatch jobs, else sbatch returns immediately after job submission. Using:

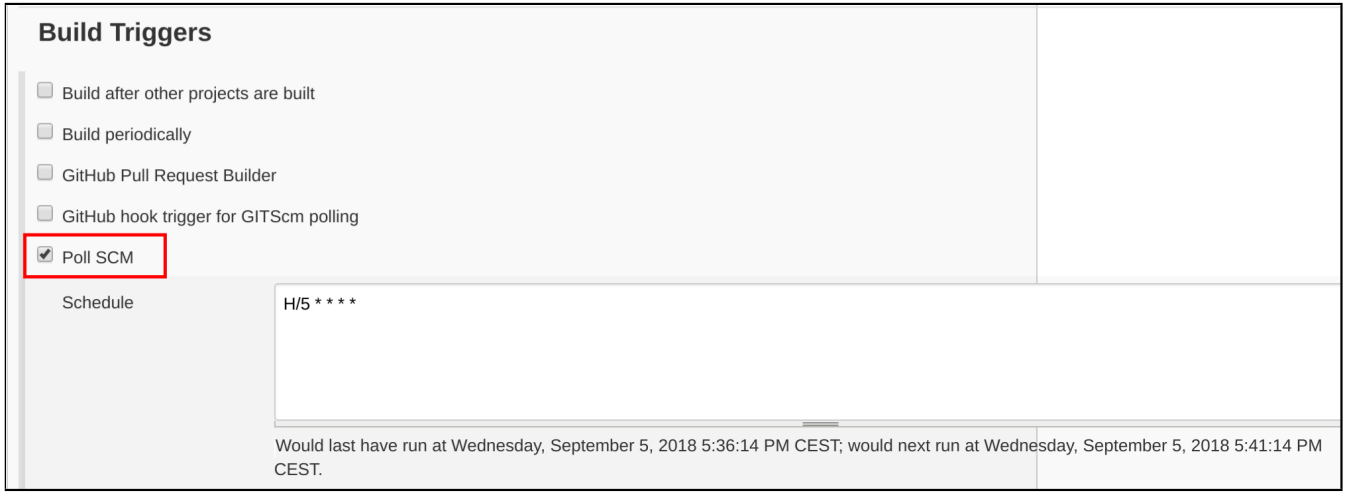
  ```
  $ sbatch --wait <batch_script>
  ```

  forces sbatch to wait for the submitted job to complete before returning.
- For 1-node jobs, it is good practice to use the **cscsci** partition which offers higher priority and is suitable for ci jobs.
- Copy the build output/errors on **SCRATCH** to have access to it.
- Make use of artifacts to store the output/error files.

Polling a remote repository

Polling a remote repository

- Since the CSCS Jenkins instance is not accessible from the public web, Jenkins has to poll the source control repository to be made aware of any changes made to the specified remote and branch and start a job.
- The above option is enabled by using the **Poll SCM** option under Build Triggers:



Github pull request builder (ghprb) plugin

Enabling the ghprb (1/2)

1. Invite the **jenkins-cscs** Github user which belongs to the CSCS UES group and is used by the CSCS jenkins instance. The above user has to be invited with Read & Write privileges.
2. Enable ghprb in your project:

Enabling the ghprb (2/2)

3. Set the advanced settings according to your needs:

Use github hooks for build triggering	☐
Trigger phrase	.*test\W+this\W+please.*
Only use trigger phrase for build triggering	☐
Close failed pull request automatically?	☐
Skip build phrase	.*\[skip\W+ci\].*
Display build errors on downstream builds?	☐
Crontab line	*/1 * * * *

A build is triggered from a new pull request

Code | Issues 0 | Pull requests 1 | Projects 0 | Wiki | Insights | Settings

Add hostname command #4

Open **teojgo** wants to merge 1 commit into `master` from `feature/add_hostname`

Conversation 0 | Commits 1 | Checks 0 | Files changed 1

teojgo commented 6 days ago — Member

No description provided.

`Add hostname command` — 22ffbd5

teojgo self-assigned this 6 days ago

Add more commits by pushing to the **`feature/add_hostname`** branch on **eth-cscs/UserLabDay**.

All checks have passed — Hide all checks

1 successful check

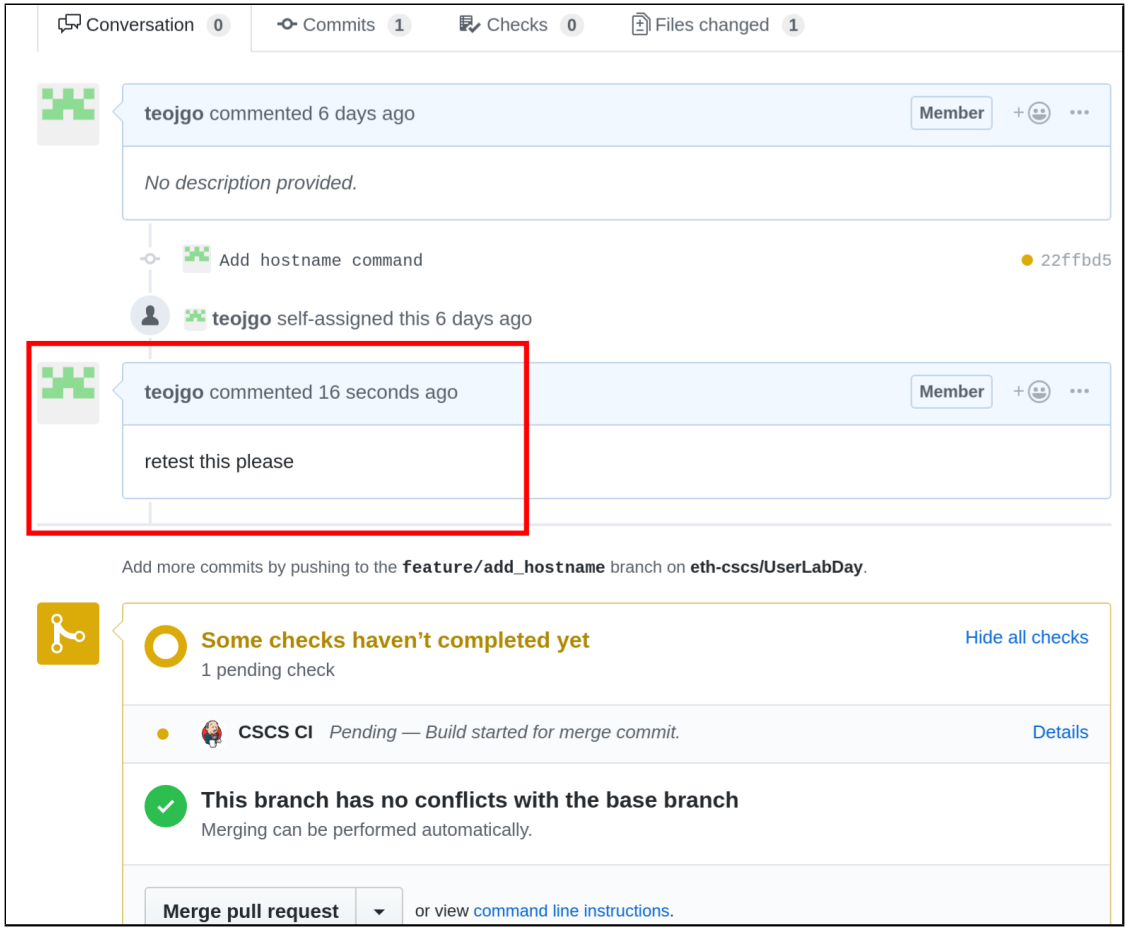
CSCS CI — Details

This branch has no conflicts with the base branch

Merging can be performed automatically.

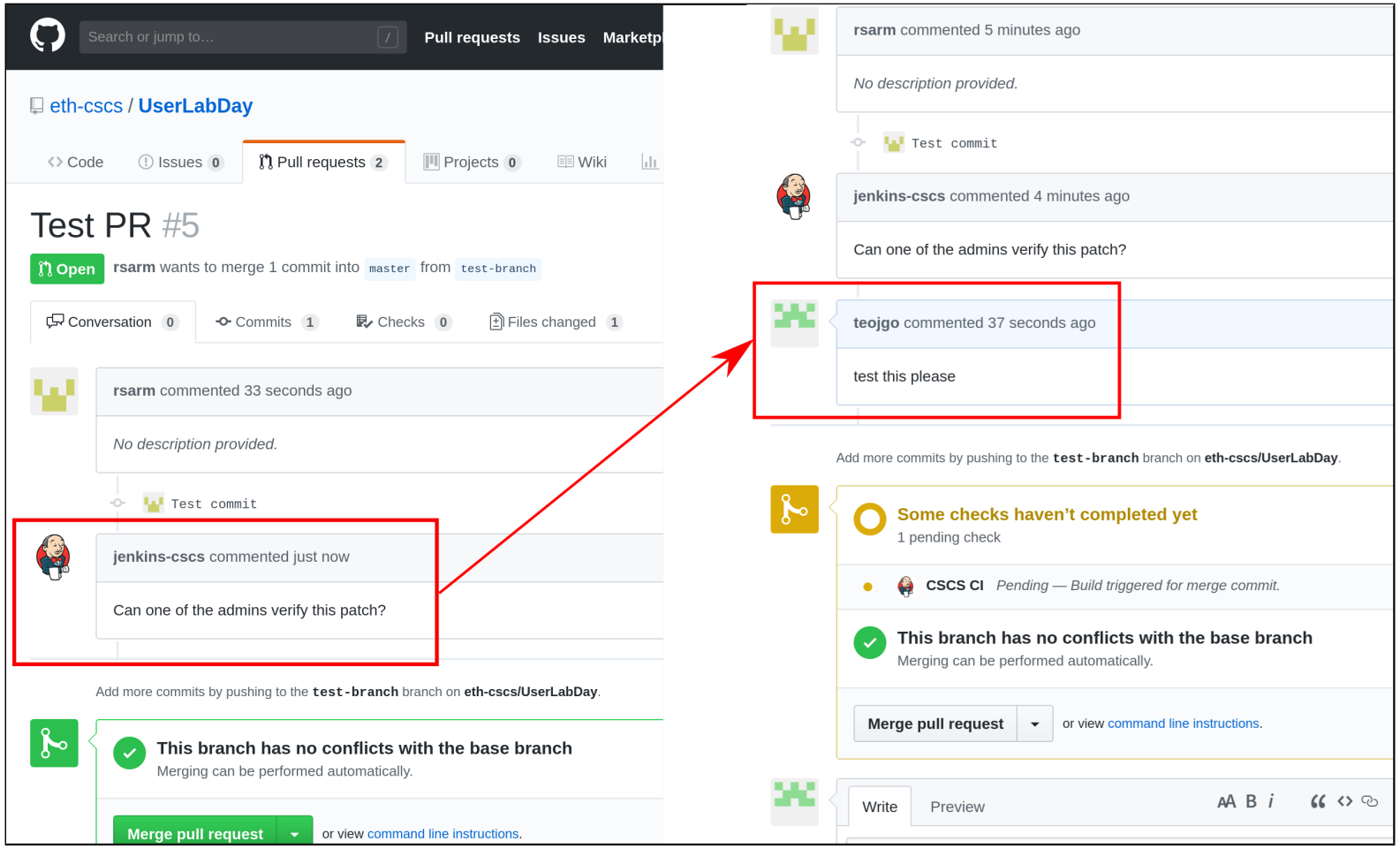
Retriggering a build

To retrigger a build for an already submitted pull request, an admin or whitelisted user has to make a comment matching a predefined pattern. In this case ”retest this please”.



Pull request of a non-whitelisted user

When a user who is not whitelisted in ghprb submits a pull request, an admin verification is needed to trigger the build.



Additional Topics/Bibliography

Further Jenkins Topics

The following advanced Jenkins topics are listed here for reference:

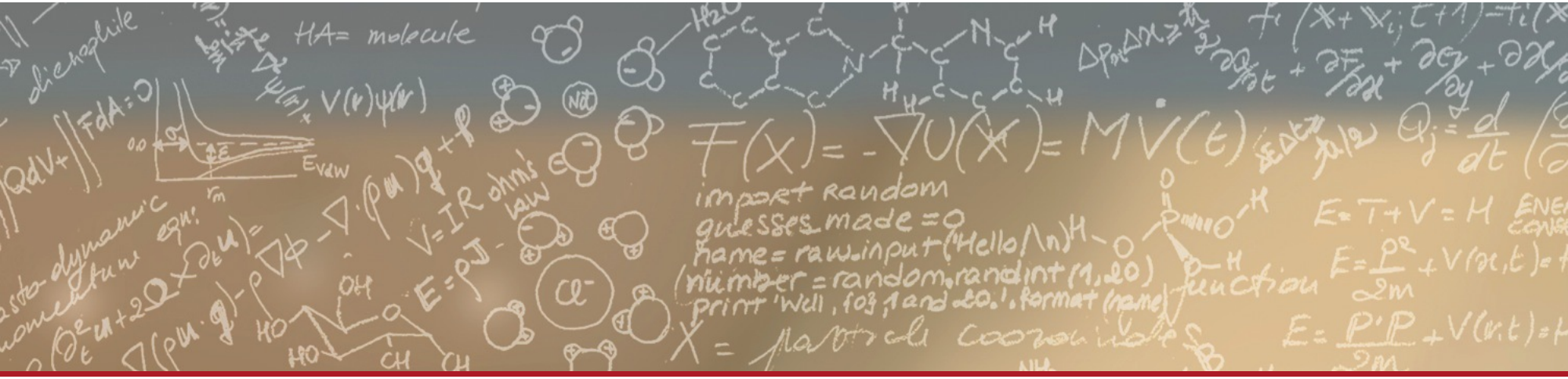
- Using Jenkins for Continuous Deployment (CD)
- Jenkins & Containers
- Jenkins Command Line Interface (CLI)
- Jenkins Rest API
- Securing Jenkins
- Performing Jenkins backups

Further Reading

- Jenkins User Documentation
- Jenkins Handbook
- Brent Laster
  Jenkins 2: Up and Running.
  O'Reilly Media, 2018.
- Joseph Muli, Arnold Okoth
  Jenkins Fundamentals.
  Packt Publishing, 2018.

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Thank you for your attention.