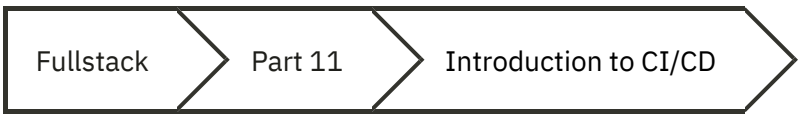


a Introduction to CI/CD

During this part, you will build a robust deployment pipeline to a ready made example project starting in exercise 11.2. You will fork the example project and that will create you a personal copy of the repository. In the last two exercises, you will build another deployment pipeline for some of *your own* previously created app!

There are 22 exercises in this part, and you need to complete *each* exercise for completing the course. Exercises are submitted via the submissions system just like in the previous parts, but unlike parts 0 to 9, the submission goes to a different "course instance".

This part will rely on many concepts covered in the previous parts of the course. It is recommended that you finish at least parts 0 to 5 before starting this part.

Unlike the other parts of this course, you do not write many lines of code in this part, it is much more about configuration. Debugging code might be hard but debugging configurations is way harder, so in this part, you need lots of patience and discipline!

Getting software to production

Writing software is all well and good but nothing exists in a vacuum. Eventually, we'll need to deploy the software to production, i.e. give it to the real users. After that we need to maintain it, release new versions, and work with other people to expand that software.

We've already used GitHub to store our source code, but what happens when we work within a team with more developers?

Many problems may arise when several developers are involved. The software might work just fine on *my computer*, but maybe some of the other developers are using a different operating system or different library versions. It is not uncommon that a code works just fine in one developer's machine but another developer can not even get it started. This is often called the "works on my machine" problem.

There are also more involved problems. If two developers are both working on changes and they haven't decided on a way to deploy to production, whose changes get deployed? How would it be possible to prevent one developer's changes from overwriting another's?

In this part, we'll cover ways to work together and build and deploy software in a strictly defined way so that it's clear *exactly* what will happen under any given circumstance.

Some useful terms

In this part we'll be using some terms you may not be familiar with or you may not have a good understanding of. We'll discuss some of these terms here. Even if you are familiar with the terms, give this section a read so when we use the terms in this part, we're on the same page.

Branches

Git allows multiple copies, streams, or versions of the code to co-exist without overwriting each other. When you first create a repository, you will be looking at the main branch (usually in git, we call this *master* or *main*, but that does vary in older projects). This is fine if there's only one developer for a project and that developer only works on one feature at a time.

Branches are useful when this environment becomes more complex. In this context, each developer can have one or more branches. Each branch is effectively a copy of the main branch with some changes that make it diverge from the master. Once the feature or change in the branch is ready it can be *merged* back into the main branch, effectively making that feature or change part of the main software. In this way, each developer can work on their own set of changes and not affect any other developer until the changes are ready.

But once one developer has merged their changes into the main branch, what happens to the other developers' branches? They are now diverging from an older copy of the main branch. How will the developer on the later branch know if their changes are compatible with the current state of the main branch? That is one of the fundamental questions we will be trying to answer in this part.

You can read more about branches e.g. from here.

Pull request

In GitHub merging a branch back to the main branch of software is quite often happening using a mechanism called pull request, where the developer who has done some changes is requesting the changes to be merged to the main branch. Once the pull request, or PR as it's often called, is made or *opened*, another developer checks that all is ok and *merges* the PR.

If you have proposed changes to the material of this course, you have already made a pull request!

Build

The term "build" has different meanings in different languages. In some interpreted languages such as Python or Ruby , there is actually no need for a build step at all.

In general when we talk about building we mean preparing software to run on the platform where it's intended to run. This might mean, for example, that if you've written your application in TypeScript, and you intend to run it on Node, then the build step might be transpiling the TypeScript into JavaScript.

This step is much more complicated (and required) in compiled languages such as C and Rust where the code needs to be compiled into an executable.

In part 7 we had a look at webpack that is the current de facto tool for building a production version of a React or any other frontend JavaScript or TypeScript codebase.

Deploy

Deployment refers to putting the software where it needs to be for the end-user to use it. In the case of libraries, this may simply mean pushing an npm package to a package archive (such as npmjs.com) where other users can find it and include it in their software.

Deploying a service (such as a web app) can vary in complexity. In part 3 our deployment workflow involved running some scripts manually and pushing the repository code to Heroku hosting service.

In this part, we'll develop a simple "deployment pipeline" that deploys each commit of your code automatically to Heroku *if* the committed code does not break anything.

Deployments can be significantly more complex, especially if we add requirements such as "the software must be available at all times during the deployment" (zero downtime deployments) or if we have to take things like database migrations into account. We won't cover complex deployments like those in this part but it's important to know that they exist.

What is CI?

The strict definition of CI (Continuous Integration) and the way the term is used in the industry are quite different. One influental but quite early (from year 2006) discussion of the topic is in Martin Fowler's blog.

Strictly speaking, CI refers to merging developer changes to the main branch often, Wikipedia even helpfully suggests: "several times a day". This is usually true but when we refer to CI in industry, we're usually talking about what happens after the actual merge happens.

We'd likely want to do some of these steps:

- Lint: to keep our code clean and maintainable
- Build: put all of our code together into software
- Test: to ensure we don't break existing features
- Package: Put it all together in an easily movable batch
- Upload/Deploy: Make it available to the world

We'll discuss each of these steps (and when they're suitable) in more detail later. What is important to remember is that this process should be strictly defined.

Usually, strict definitions act as a constraint on creativity/development speed. This, however, should usually not be true for CI. This strictness should be set up in such a way as to allow for easier development and working together. Using a good CI system (such as GitHub Actions that we'll cover in this part) will allow us to do this all automagically.

Packaging and Deployment as a part of CI

It may be worthwhile to note that packaging and especially deployment are sometimes not considered to fall under the umbrella of CI. We'll add them in here because in the real world it makes sense to lump it all together. This is partly because they make sense in the context of the flow and pipeline (I want to get my code to users) and partially because these are in fact the most likely point of failure.

The packaging is often an area where issues crop up in CI as this isn't something that's usually tested locally. It makes sense to test the packaging of a project during the CI workflow even if we don't do anything with the resulting package. With some workflows, we may even be testing the already built packages. This assures us that we have tested the code in the same form as what will be deployed to production.

What about deployment then? We'll talk about consistency and repeatability at length in the coming sections but we'll mention here that we want a process that always looks the same, whether we're running tests on a development branch or the master. In fact, the process may *literally* be the same with only a check at the end to determine if we are on the master branch and need to do a deployment. In this context, it makes sense to include deployment in the CI process since we'll be maintaining it at the same time we work on CI.

Is this CD thing related?

The terms *Continuous Delivery* and *Continuous Deployment* (both of which have the acronym CD) are often used when one talks about CI that also takes care of deployments. We won't bore you with the exact definition (you can use e.g. Wikipedia or another Martin Fowler blog post) but in general, we refer to CD as the practice where the master branch is kept deployable at all times. In general, this is also frequently coupled with automated deployments triggered from merges into the master/base branch.

What about the murky area between CI and CD? If we, for example, have tests that must be run before any new code can be merged to master, is this CI because we're making frequent merges to master, or is it CD because we're making sure that master is always deployable?

So, some concepts frequently cross the line between CI and CD and, as we discussed above, deployment sometimes makes sense to consider CD as part of CI. This is why you'll often see references to CI/CD to describe the entire process. We'll use the terms "CI" and "CI/CD" interchangeably in this part.

Why is it important?

Above we talked about the "works on my machine" problem and the deployment of multiple changes, but what about other issues. What if Alice committed directly to master? What if Bob used a branch but didn't bother to run tests before merging? What if Charlie tries to build the software for production but does so with the wrong parameters?

With the use of continuous integration and systematic ways of working, we can avoid these.

- We can disallow commits directly to master
- We can have our CI process run on all Pull Requests (PRs) against master and allow merges

only when our desired conditions are met e.g. tests pass

- We can build our packages for production in the known environment of the CI system

There are other advantages to extending this setup:

- If we use CD with deployment every time there is a merge to master then we know that master is always running in production
- If we only allow merges when the branch has an up to date master, then we can be sure that different developers don't overwrite each other's changes

Note that in this part we are assuming that *master* or *main* branch contains the code that is running in production. The numerous different workflows one can use with git, e.g. in some cases, it may be a specific *release branch* that contains the code which is running in production.

Important principles

It's important to remember that CI/CD is not the goal. The goal is better, faster software development with fewer preventable bugs and better team cooperation.

To that end, CI should always be configured to the task at hand and the project itself. The end goal should be kept in mind at all times. You can think of CI as the answer to these questions:

- How to make sure that tests run on all code that will be deployed?
- How to make sure that the master branch is deployable at all times?
- How to ensure that builds will be consistent and will always work on the platform it'd be deploying to?
- How to make sure that the changes don't overwrite each other?
- How to make deployments happen at the click of a button or automatically when one merges to master?

There even exists scientific evidence on the numerous benefits the usage of CI/CD has. According to a large study reported in the book Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations, the use of CI/CD correlate heavily with organizational success (e.g. improves profitability and product quality, increases market share, shortens the time to market). CI/CD even makes developers happier by reducing their burnout rate. The results summarized in the book are also reported in scientific articles such as this.

Documented behavior

There's an old joke that a bug is just an "undocumented feature". We'd like to avoid that. We'd like to avoid any situations where we don't know the exact outcome. For example, if we depend on a label on a PR to define whether something is a "major", "minor" or "patch" release (we'll cover the meanings of those terms later), then it's important that we know what happens if a developer forgets to put a label on their PR. What if they put a label on after the build/test process has started? What happens if the developer changes the label mid-way through, which one is the one

that actually releases?

It's possible to cover all cases you can think of and still have gaps where the developer will do something "creative" that you didn't think of, so it's important to have the process fail safely in this case.

For example, if we have the case mentioned above where the label changes midway through the build. If we didn't think of this beforehand, it might be best to fail the build and alert the user if something we weren't expecting happened. The alternative, where we deploy the wrong type of version anyway, could result in bigger problems, so failing and notifying the developer is the safest way out of this situation.

Know the same thing happens every time

We might have the best tests imaginable for our software, tests that catch every possible issue. That's great, but they're useless if we don't run them on the code before it's deployed.

We need to guarantee that the tests will run and we need to be sure that they run against the code that will actually be deployed. For example, it's no use if the tests are *only* run against Alice's branch if they would fail after merging to master. We're deploying from the master so we need to make sure that the tests are run against a copy of master with Alice's changes merged in.

This brings us to a critical concept. We need to make sure that the same thing happens every time. Or rather that the required tasks are all performed and in the right order.

Code always kept deployable

Having code that's always deployable (and provably so) makes life easier. This is especially so when the master branch contains the code running in the production environment. For example, if a bug is found and it needs to be fixed, you can pull a copy of master (knowing it is the code running in production), fix the bug, and make a pull request back to master. This is relatively straight forward.

If, on the other hand, master and production are very different and master is not deployable, then you would have to find out what code *is* running in production, pull a copy of that, fix the bug, figure out a way to push it back, then work out how to deploy that specific commit. That's not great and would have to be a completely different workflow from a normal deployment.

Knowing what code is deployed (sha sum/version)

It's often important to know what is actually running in production. Ideally, as we discussed above, we'd have master running in production. This is not always possible. Sometimes we intend to have master in production but a build fails, sometimes we batch together several changes and want to have them all deployed at once.

What we need in these cases (and is a good idea in general) is to know exactly *what code is running in production*. Sometimes this can be done with a version number, sometimes it's useful to have the commit SHA sum (uniquely identifying hash of that particular commit in git) attached to the code. We will discuss versioning further a bit later in this part.

It is even more useful if we combine the version information with a history of all releases. If, for example, we found out that a particular commit has introduced a bug, we can find out exactly

when that was released and how many users were affected. This is especially useful when that bug has written bad data to the database. We'd now be able to track where that bad data went based on the time.

Types of CI setup

To meet some of the requirements listed above, we want to dedicate a separate server for running the tasks in continuous integration. Having a separate server for the purpose minimizes the risk that something else interferes with the CI/CD process and causes it to be unpredictable.

There are two options: host our own server or use a cloud service.

Jenkins (and other self-hosted setups)

Among the self-hosted options, Jenkins is the most popular. It's extremely flexible and there are plugins for almost anything (except that one thing you want to do). This is a great option for many applications, using a self-hosted setup means that the entire environment is under your control, the number of resources can be controlled, secrets (we'll elaborate a little more on security in later sections of this part) are never exposed to anyone else and you can do anything you want on the hardware.

Unfortunately, there is a downside. Jenkins is quite complicated to set up. It's very flexible but that means that there's often quite a bit of boilerplate/template code involved to get builds working. With Jenkins specifically, it also means that CI/CD must be set up with Jenkins' own domain-specific language. There are also the risks of hardware failures which can be an issue if the setup sees heavy use.

With self-hosted options, the billing is usually based on the hardware. You pay for the server. What you do on the server doesn't change the billing.

GitHub Actions and other cloud-based solutions

In a cloud-hosted setup, the setup of the environment is not something you need to worry about. It's there, all you need to do is tell it what to do. Doing that usually involves putting a file in your repository and then telling the CI system to read the file (or to check your repository for that particular file).

The actual CI config for the cloud-based options is often a little simpler, at least if you stay within what is considered "normal" usage. If you want to do something a little bit more special, then cloud-based options may become more limited, or you may find it difficult to do that one specific task for which the cloud platform just isn't built for.

In this part, we'll look at a fairly normal use case. The more complicated setups might, for example, make use of specific hardware resources, e.g. a GPU.

Aside from the configuration issue mentioned above, there are often resource limitations on cloud-based platforms. In a self-hosted setup, if a build is slow, you can just get a bigger server and throw more resources at it. In cloud-based options, this may not be possible. For example, in GitHub Actions, the nodes your builds will run on have 2 vCPUs and 8GB of RAM.

Cloud-based options are also usually billed by build time which is something to consider.

Why pick one over the other

In general, if you have a small to medium software project that doesn't have any special requirements (e.g. a need for a graphics card to run tests), a cloud-based solution is probably best. The configuration is simple and you don't need to go to the hassle or expense of setting up your own system. For smaller projects especially, this should be cheaper.

For larger projects where more resources are needed or in larger companies where there are multiple teams and projects to take advantage of it, a self-hosted CI setup is probably the way to go.

Why use GitHub Actions for this course

For this course, we'll use GitHub Actions. It is an obvious choice since we're using GitHub anyway. We can get a robust CI solution working immediately without any hassle of setting up a server or configuring a third-party cloud-based service.

Besides being easy to take into use, GitHub Actions is a good choice in other respects. It might be the best cloud-based solution at the moment. It has gained lots of popularity since its initial release in November 2019.

Exercise 11.1

Before getting our hands dirty with setting up the CI/CD pipeline let us reflect a bit on what we have read.

11.1 warming up

Think about a hypothetical situation where we have an application being worked on by a team of about 6 people. The application is in active development and will be released soon.

Let us assume that the application is coded with some other language than JavaScript/TypeScript, e.g. in Python, Java, or Ruby. You can freely pick the language. This might even be a language you do not know much yourself.

Write a short text, say 200-300 words, where you answer or discuss some of the points below. You can check the length with https://wordcounter.net/ . Save your answer to the file named *exercise1.md* in the root of the repository that you shall create in exercise 11.2 .

The points to discuss:

- Some common steps in a CI setup include *linting*, *testing*, and *building*. What are the specific tools for taking care of these steps in the ecosystem of the language you picked? You can search for the answers by google.
- What alternatives are there to set up the CI besides Jenkins and GitHub Actions? Again, you can ask google!
- Would this setup be better in a self-hosted or a cloud-based environment? Why? What

information would you need to make that decision?

Remember that there are no 'right' answers to the above!

Propose changes to material

Part 10
Previous part

Part 11b
Next part

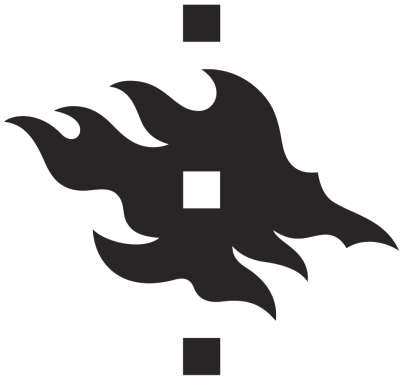
About course

Course contents

FAQ

Partners

Challenge



UNIVERSITY OF HELSINKI


HOUSTON