Corndel DevOps Engineering Programme

in association with Softwire

Module 7: Continuous Integration

Corndel Digital.

DAAG

Introduction

In the previous module we talked about wanting cycles of work; to reduce the time between new versions of our application being available so that we can iterate rapidly and deliver value to users more quickly. To accomplish this with a *team* of software developers requires individual efforts to regularly be merged together for release. This process is called **integration**.

In less agile workflows integration might happen weekly, monthly or even less often. Long periods of isolated development provide lots of opportunities for code to diverge and for integration to be a slow, painful experience. Agile teams want to release their code more regularly: at least once a sprint. In practice this means we want to be integrating our code much more often: twice a week, once a day, or even multiple times a day. Ideally new code should be **continuously integrated** into our main branch to minimise divergence.

In order to do this we need to make integration and deployment quick and painless; including compiling our application, running our tests, and releasing it to production. We need to automate all these process to make them quick, repeatable and reliable.

Continuous integration is the practice of merging code regularly and having sufficient automated checks in place to ensure that we catch and correct mistakes as soon as they are introduced. We can then build on continuous integration to further automate the release process. This involves the related concepts of **continuous delivery** and **continuous deployment**. Both of these processes involve automating all stages of building, testing and releasing an application, with one exception: continuous delivery requires human intervention to release new code to production; continuous deployment does not.

Continuous deployment sounds scary (and it is). It requires a high level of confidence in your automated systems to prevent bugs and other mistakes getting into your live environment. It's also not appropriate for all organisations and industries.

We'll cover continuous delivery and continuous deployment in the next module. For now we will concentrate on continuous integration, which is a prerequisite for the others and a very helpful concept in its own right.

Table of Contents

Chapter 1

Continuous Integration

In this section we introduce the concept of **continuous integration (CI)** and why we might want to use it. We also highlight some of the challenges of adopting it and look at some of the techniques that we can use to mitigate those challenges.

What is continuous integration?

The core principle of continuous integration is that code changes should be merged into the main branch of a repository as soon as possible. Under continuous integration all code branches should be short lived and their changes merged back into the main branch frequently. It favours small, incremental changes over large, comprehensive updates.

Why do we want it?

CI has many potential benefits, including reducing the risk of developers interfering with each others' work and reducing the time it takes to get code to production.

Having multiple authors work in parallel on the same codebase introduces the risk of different changes being made to the same area of code at the same time. Sometimes the changes are trivial and can be automatically merged by a tool like `git`. However, sometimes the two sets of changes are fundamentally inconsistent with each other. This results in a **merge conflict** which has to be manually resolved by the developers.

Regular code merges reduce the risk of two developers changing the same code at the same time. This reduces both the number and the severity of merge conflicts.

CI also speeds up development. Merging your code changes into the main branch regularly makes your code changes available to other developers as soon as possible so they can build upon your work. It also ensures that the main branch is as up-to-date as possible. This means that your application is, in theory, always ready to be released. Automated testing on the main branch is more meaningful; it is testing a release candidate, rather than each feature in isolation.

What makes it hard?

Merging code to `main` regularly isn't without problems. With a busy team your main branch will be in a constant state of flux. It is very likely that different sets of changes will not work together as expected; or worse yet, break existing functionality. QA will find some bugs but no manual process will be able to keep up with a team that may be merging new branches multiple times per day.

How do you keep on top of it? How can you be confident that the code still works? How can you be confident that a changeset didn't break existing functionality? It's not an easy problem.

How do we do it?

Fortunately, there are several practices that we can use to help mitigate the challenges of merging regularly while still accepting all the advantages!

Above all, communication is key. Coordinating the changes you are making with the rest of your team will reduce the risk of merge conflicts. This is why **stand ups** are so important in agile teams.

You should also have a comprehensive, automated test suite. Knowing that you have tests covering all your essential functionality will give you confidence that your new feature works and that it hasn't broken any existing functionality.

Being able to run your tests quickly every time you make a change reduces the risks of merging regularly. Some testing tools will run your test suite automatically on your dev machine in real time as you develop. This is really helpful to provide quick feedback (and helps stop you forgetting to run the tests).

Continuous integration servers

Automatically running tests locally is good; running them on a central, shared server is even better. At a minimum this ensures that the tests aren't just passing on your machine. At its best, a continuous integration server can provide a wide variety of useful processes and opens the door to automated deployment. More on this later.

Continuous integration is always a good idea. While it may be challenging to introduce on some legacy applications; the benefits on a greenfield project will almost always outweigh the costs. This isn't just limited to agile - even a waterfall based project can benefit from continuous code integration.

In the rest of this module we'll look at the practicalities of setting up a CI pipeline, and introduce some of the most commonly used tools.

Branch Protection

Some version control tools, like GitHub, have the concept of **protected branches**. One way of using protected branches is to prevent any developer from changing that branch directly. Instead an automated process will watch for any new merge requests; attempt to automatically merge the code in *private*; and run the automated tests. If and *only* if the tests pass will the branch be permitted to merge into the public history. This can ensure that the `main` branch is never in a broken state (so long as your test suite is comprehensive)!

Chapter 2

Pipelines

Pipelines crop up in many areas of software development, including data processing and machine learning. In this section we will be focusing on **build pipelines** and **release pipelines**. We will discuss what these pipelines are, their purpose, and why they might be helpful to us.

What is a pipeline?

In the last chapter we saw that automatically running tests was a key part of continuous integration. In practice, we cannot just "run the tests". While we try to make our projects easy to work with, even the simplest of codebases will require multiple steps to run the tests from scratch.

Let us look at a very simple example:

- Checkout the latest version of the code from a repository.
- Install dependencies.
- (Depending on the technology used) Compile or otherwise build the code.
- Run unit or integration tests.

These steps form our **pipeline**. They are executed in sequence and if any step is unsuccessful then the entire pipeline has failed. This means there is a problem with our code (or our tests)!

If all the steps in the pipeline complete successfully then the pipeline has succeeded. If you have good quality tests, with high code coverage then you can be *reasonably* confident that the latest code changes haven't broken anything.

In practice a pipeline may include more steps than just those in the example above. Depending on your goals you may want to run code quality checks; apply database migrations; set environment variables; load code onto a server; or check you haven't left any "TODOs" in your codebase. You can make pipelines as simple or complicated as necessary for your project.

Build vs release

As the names suggest, *build* pipelines and *release* pipelines serve slightly different purposes. A build pipeline tends to start with raw code and produces a **build artefact**, often an executable binary or other deployable object. A release pipeline takes built programs and deploys them to a given environment. Often a release pipeline is triggered by the successful completion of a build pipeline. Some tools, like Azure DevOps, draw a clear distinction between the two; other tools are more fluid. Separating the *build* from the *release* is a good idea; although in practice it may not always be that clear cut.

A continuous integration process will generally involve a build process. It may also involve deploying that to a test environment, perhaps to run some sort of end-to-end system test. This can blur the lines between continuous integration, continuous delivery and continuous deployment. The extent to which you do this will depend upon your application and the wider business context and culture.

Why not a script?

One might ask why bother with the extra overhead of a pipeline over a simple script? Decomposing your build and release process into the discrete steps of a pipeline increases reusability.

Automating the pipeline

While a pipeline *could* be a manual process, it is definitely better to automate it. One of the simplest approaches is with scripting. If your project is relatively simple and only needs a basic pipeline then a single shell script may suffice.

The most important thing is that we have a reliable, consistent, and repeatable approach to running our pipeline. Often, you will find that it is easier to configure a tool to do all this work for you. In many cases the tool will still be essentially just running those same scripts. However, it now does so in an easy, repeatable way. Ideally your tool should be configured to run in just one click (or even better, automatically when you push to version control).

Breaking the Build

If you have committed code to the project that results in either the tests failing or, worse, the code being unable to compile you are said to have **broken the build**. This is bad: the project is now in a broken, useless state. Fixing it quickly should be a matter of priority. More sophisticated build pipelines will prevent build breaking commits from being merged to `main`. This is definitely highly desirable, if it can be achieved.

Using a dedicated server-based build tool has many advantages. It means the build pipeline is a universal, shared resource and not reliant on any one developer's individual machine. This allows greater **scalability**: multiple builds can run in parallel if necessary. The server can also securely store any secret configuration values, like database connection strings, centrally, improving **security** by removing the need for developers to access them directly.

It also allows the team to set up additional automated behaviour; for example **sending alerts** to everyone on the team when the build breaks. Finally, it allows the results of build pipelines to be published and presented on **dashboards** to make the current state of the project visible to the entire team. This visibility is important as the entire team shares responsibility for the status of the build.

Chapter 3

Common CI Tools

There are a large number of CI tools available with a variety of features, limitations and costs. The selection of a tool will depend a lot upon your team's requirements. Generally a development team would prefer to use a single CI tool; although there are always going to be exceptions. These tools are often referred to as CI/CD tools or platforms, as they typically handle not only continuous integration, but also delivery and deployment.

In this section we'll cover the basic anatomy of CI tools and name-check some of the most common tools in the market today. This isn't a comprehensive guide and the list will be ever changing; this is just another reason DevOps engineers need to be constantly learning about developments in our field.

Anatomy of a CI Tool

While there are some substantial differences between different CI tools they do have some common elements.

Linking to a repository

The CI tool must be linked to your source code. Modern tools generally have built in support to easily link themselves to common providers (for example GitHub, GitLab and Bitbucket). If you're using a less common or privately hosted solution you may need to do some additional set up.

When linking to a repository you will also commonly define the conditions for triggering a build. These will vary from project to project, but common use-cases include:

- Running after every commit to a specific branch
- Preemptively building and testing every time a pull request is raised
    - Optionally, a failed pipeline can block merging of the PR
- Running on a regular timer
    - A "nightly" build of a rarely changed program can help identify issues with changing dependencies and help prevent *code rot.*

Configuration

There are two distinct bits of configuration to be aware of:

1. The instructions on how to run the pipeline for your specific project; and
2. Environment specific variables that are injected into the pipeline.

This allows you to define your pipeline once and then reuse the configuration for multiple environments. The environment specific variables may include **secrets**, such as passwords or connection strings. Some tools provide mechanisms to add extra layers of protection to secrets and prevent them being accidentally exposed in the UI or build logs.

Job configuration is usually stored in one or two ways: either as configuration file stored alongside the source code; or as separately created configuration in the CI tool. Where you have a choice we highly recommend the former: storing your configuration as a file alongside the source code. This enables you to take advantage of version control; making your build process much more maintainable.

Build Agent

When using a CI tool you still need some sort of computer to run your pipeline. This is usually called a **build agent**. Just as we saw with our discussion on infrastructure earlier in the course there are two broad methods of creating build agents: using a dedicated VM or creating ephemeral containers.

Most modern, cloud-based CI tools use the container approach. These function in a very similar manner to Docker: starting from a base image (usually Windows or Linux); installing any necessary

What is Code Rot?

When code is first written it is considered fresh. It uses the latest libraries, the most up-to-date syntax and current best practice. This will not last. Over time the app's dependencies will change, develop and eventually become unsupported; the language will evolve with new syntax and core features; and best practice will change as new technologies emerge. While the original code hasn't changed it has effectively "rotted" because the world around it has moved on.

An application *can* run for months, years or even decades without being updated. However, when you need to add a new feature or implement a security fix you will probably uncover lots of additional work: upgrading dependencies, adapting to breaking changes and new syntax. In the worst case the entire tech stack may now be unsupported, making it impossible to build a new version of the application!

Code rot can be avoided by regular, on-going maintenance. A nightly build can alert the team to some breaking changes; tools like GitHub's Dependabot can help highlight obsolete dependencies early and even automatically fix them.

tools or dependencies; then finally building and testing your code. This allows you to define the entire build environment as code.

Using a dedicated VM, the approach traditionally favoured by Jenkins, runs into many of the same issues we considered with "snowflake servers" in previous modules. It can also result in different jobs running on the same server; potentially allowing different jobs to interfere with each other, a problem solved by using containers. However, dedicated VMs have some advantages over containers in CI. When using containerised pipelines a new container will typically be have to be created for each new build. The overhead of container creation and set-up can be a significant part of the build time.

Docker

It is also possible to use Docker to containerise your entire CI/CD pipeline. This may be particularly useful if your production environment uses Docker. It will also allow you to run the same CI/CD pipeline locally, avoiding "it works for me" problems. Using Docker is an alternative to using a CI tool's native container, described above, and brings many of the same benefits.

Some tools support Docker natively but do be cautious - you may run into problems with subtle variations between different implementations of Docker.

Configuration File Language

Historically CI/CD tools have used a variety of **domain-specific languages** or **DSLs** for their configuration files. Recently most tools have been standardising on **YAML** for configuration.

This isn't the case for other tools in areas, such as configuration management or infrastructure provisioning tools, where DSLs are still more common. Any project involving these tools is likely to contain configuration files with varying syntaxes. You will need to be adaptable.

Choosing a tool

The choice of tool will be very dependent on the needs of your team. There are various things you will want to consider including the operating systems supported by the tool; what integrations it supports; the flexibility to support different build patterns; whether it allows self-hosting or is only available as SaaS; and finally cost.

There are no hard and fast rules: you will need to work out what works best for your team. It is always possible to change later on, although there will be some costs associated with this.

Common Tools

Jenkins

Link: https://jenkins.io/

Jenkins is a widely used, open-source automation server. It is exceedingly flexible and has a wide variety of community written plugins. Its versatility and longevity - it was first released in early 2005 - mean that Jenkins has acquired a lot of complexity and configurability. While powerful, it can be difficult to ramp up on.

Pipelines are supported through the *Pipeline* plugin, which has been enabled by default since 2016. Build instructions are defined in *Jenkinsfiles*, written in a language based on Groovy and stored alongside the project files in source control. Prior to this builds were configured through the UI and stored separately.

What's in a name?

The version of Jenkins released in 2005 wasn't actually called Jenkins. The project we now know as Jenkins started life at Sun Microsystems and was called Hudson. After Oracle acquired Sun Microsystems in 2010 disagreements between project contributors and Oracle resulted in the project splitting into two: Hudson, developed by Oracle; and Jenkins developed by the open source community.

While Hudson stopped development in 2016, Jenkins is still going strong. It does, however, still bear traces of its original name in its source code - over 2000 references at the time of writing.

Example Jenkinsfile:

```
pipeline {
    agent windows // Restricts the job to running on an agent
                  // with the 'windows' label
                  // We would be responsible for ensuring that agent
                  // had the correct dependencies before running the job

    stages {
        // Unlike the CircleCI example below, Jenkins performs
        // a code checkout automatically before the pipeline runs.
        stage('Build') {
            steps {
                bat 'dotnet build'
            }
        }
        stage('Test') {
            steps {
                bat 'cd DotnetTemplate.Web.Tests'
                bat 'dotnet test'
            }
        }
    }
}
```

CircleCI

Link: https://circleci.com/

CircleCI is a much simpler tool than Jenkins, in part because it is much more opinionated. It is strongly based around git branches, auto-running every time new code is pushed.

It is very easy to set-up and integrate with GitHub; it will actually generate some basic build configuration for you and add it directly to your repository. While it does allow shared configuration through packages called **orbs** it is much more limited than Jenkins; however, this means it is also much simpler and very easy to get started with.

CircleCI is configured using YAML, an example is shown:

```
version: 2.1

orbs:
 win: circleci/windows@2.2.0

jobs:
  build:
    executor: win/default

    steps:
      - checkout
      - run: dotnet build
      - run:
         command: |
           cd DotnetTemplate.Web.Tests
           dotnet test
```

Azure Pipelines

Azure Pipelines is part of Microsoft's Azure DevOps platform. While it is focused on integrating with Azure - it shares an authentication platform - it can also be used independently if required.

Unlike some other providers Azure makes a clear distinction between **build pipelines** and **release pipelines**, with the output of the former *generally* being the input to the later. This encourages the principle of "build once, deploy many"; where the same build artefact is deployed to each environment, differing only by config.

Azure Pipelines can be configured using YML. It also has a GUI and can be used to create reusable components, or use third party components from its built-in market place.

Other Cloud Providers

In addition to Azure, both AWS (Amazon) and GCP (Google) have their own CI/CD solutions. Like Azure, these also focus on deploying to their own environments.

AWS CodePipeline

- A fully managed continuous delivery service from AWS.

Cloud Build

- Google's CI/CD platform.

Other CI/CD Solutions

The CI/CD solutions mentioned above are just examples, not recommendations. Many different tools are available; you should pick the correct tool for your project and your organisation.

Additional CI/CD tools are listed in the Additional Material for this module.

Chapter 4

Exercise

The *theory* of why continuous integration is a Good Thing is pretty straightforward. The practicalities of implementing CI pipelines for different projects on different tools can be quite a bit more complicated.

Instead of reading a lot of additional material, you should spend some time experimenting with some of the CI tools mentioned in this chapter. Lots of them have good tutorials and onboarding documentation that should let you get started. See if you can configure various tools to build some code, run some tests and report the results.

This would also be a good opportunity to find out more about the CI tool used at your place of work as that will be the one you will use most in the immediate future.

Chapter 5

Additional Material

Other CI/CD Tools

Some additional CI/CD tools are listed below. These are examples and *not* recommendations.

Integrated Solutions

Both GitHub and Gitlab now include CI/CD solutions natively in their source control platforms.

GitLab CI/CD

1. A CI/CD tool built directly into the GitLab repository hosting service.

GitHub Actions

- A CI/CD tool built directly into Github.

Other Tools

- Travis CI
    - A very similar tool to CircleCI, widely used for open source projects.
- TeamCity
    - A CI/CD offering from JetBrains
- Bamboo
    - A CI/CD offering from Atlassian
- Concourse CI
    - An extremely opinionated but powerful "thing doer" with a hard learning curve
- Drone
    - A relatively new open-source CI platform
- Heroku Pipelines
    - Heroku is a cloud platform provider that allows developers to run their code in containers called **Dynos**.
    - Heroku Pipelines is a CI/CD tool built on Heroku, particularly helpful for projects on Heroku.