CP-DCB

Continuous Build and Integration

DevOps++ Certified Professional Series

Continuous Build and Integration

DevOps++ Certified Professional Series

Schalk Cronjé and DevOps++ Alliance

This book is for sale at http://leanpub.com/continuousbuild

This version was published on 2019-08-12

Contents

Principles of Continuous Build and Role of Continuous Integration

Treat Everything as Code

The source code repository is the source of truth for any project or product. In this way recreating the environment somewhere else becomes an advtanage and not a constraint. Add all test setup to the repository. Add all CI configuration as source as well, or creste scripts which can automatically configuring CI and add that to to the repository.

Walking skeleton

Building a **walking skeleton** is the process of building a minimal pipeline at the start of a new product that will invoke each of the steps from building the product to eventually deploying it. The earlier this is done in the project the easier it is and the easier it is to detect issues that might arise during any part of the process. Using a process of continuous building this can lead to achieving the goal of going from a commit of source code to a deployment in production without any further interaction.

In this book the focus is on the setting up continuous builds for various environments. Deployments will not be discussed in detail, but some skeleton components will be put in place for deployments.

Bootstrap as much as possible

Minimize manual installation of tools.

Certain tools are harder to bootstrap. Typical examples of these are C++ compilers and the .NET runtime. Python environments can be managed with `virtualenv` and Ruby environments with `rvm`.

The most ideal situation is where the build tool can bootstrap everything. This allows finer control of versions of tools and can prevent the leakage of the tool environments across projects.

Maximize reproducibility outside CI

A common problem is that things might fail on CI. The more complex the pipelines on CI, the higher the probability of failure will be. A major time waster is that people will try to fix something in the

pipeline, but checking in untested pipeline orchestrating code, then wait to see the result. This leads to unnecessary builds, unnecessary notifications and frustration. it might also mean that pipelines are taken off-line whilst issues are being debugged. This again holds up other people from obtaining feedback from their changes. Therefore maximize the amount of build and deployment that can be done outside of a CI server.

CI should to a great extent purely confirm that you have already tested everything and communicate the message to other people working on the project.

There are cases where the full functional pipeline is too expensive to reproduce outside of CI, that is way the principle is to maximize as much that is cost-effective.

Build Tools

Build tools are the blue collar workers of the software industry. They are seldom revered, they are seldom considered sexy, yet they are the work horses of the industry. Without build tools, software will not easily get built.

In simple terms, build tools have a collection of recipes (rules) which can convert souce code into useable artifacts. In reality modern build tools do more and can be use as coordination tools for deployment, publication, event triggers and even application lifecycle control.

Classic build tools for native platforms are GMake[1], AMake[2], including various variants, and SCons[3]. On the JVM the most popular tools are Apache Ant[4], Apache Maven[5], Gradle Build Tool[6], Leiningen[7] and Scala Build Tool[8]. People working on the Windows operating system and mostly using .NET will be familiar with MSBuild[9] and some extent XBuild[10]. For JavaScript, there is Grunt[11] and Gulp[12] as the most popular choices. Some programming languages even have their own build tools, which are very language-specific and not really suited to building anything else. This includes Go and Rust. There are even tools which specialise in massive parallel builds such as Basel[13].

As a person setting up CI environments, you will no doubt encounter one of more of these tools. It is important that you study and know the capabilities of the build tools that you have to deal with in order to obtain maximum effectiveness out of the build pipeline.

In order to explore the capabilities that build tools can offer, the rest of this book will use one of the most powerful and flexible build tools available on the market today, namely the Gradle Build Tool.

[1] https://www.gnu.org/software/make/
[2] http://amake.sourceforge.net/
[3] http://scons.org/
[4] https://ant.apache.org/
[5] https://maven.apache.org/
[6] https://gradle.org/
[7] https://leiningen.org/
[8] http://www.scala-sbt.org/
[9] https://github.com/microsoft/msbuild
[10] http://www.mono-project.com/docs/tools+libraries/tools/xbuild/
[11] https://gruntjs.com/
[12] http://gulpjs.com/
[13] https://bazel.build/

Continuous Integration (CI)

The term **continuous integration**, can be tracked back as far as 1994 when Grady Booch used to describe the concept of developing micro processes and how the *internal releases represent a sort of continuous integration of the system, and exist to force closure of the micro process.* [Booch98]. The term really came to be well known with the advent of Extreme Programming [Beck99] and via Martin Fowler's famous article in 2006 [Fowler06].

Technology might have moved on, but the CI practices that Fowler layed out, is mostly as applicable today as it was then.

Automate the Build
: Source control is the source of truth for everything related to the project. Automating the build means that for the full system can be validated in an independent way down to every commit if you so wish.

Make Your Build Self-Testing
: The build executes all of the tests. A well designed build will have different levels of testing. Some of these, typically the unit tests, will execute first as they run the quickest and can provide feedback the quickest. For expensive tests that require a lot of additional infrastructure, pipelines can be set up to be run a further stage.

Everyone Commits To the Mainline Every Day
: To a great extent this practice depends on the source control system in use. At the time of Fowler's original article, Subversion[14] (or Apache Subversion as it is now official known as) was the darling source control system. Subversion uses a centralises model, with no real offline capability.[15] This means that the only safe way to keep code is to commit it to the centralized repository ebery day. These daily commits also meant that developers did not fall far behind other changes and that they could integrate other changes back into their own code every day. With the popularity of distributions source control systems growing, especially that of Git, the original reasoning has fallen by the wayside, but the practice is still common in many teams. It is a good practice to keep, but the decision needs to be done in the context of the project. With a distributed system like Git, it is possible to continually test integrations to the main branch without actually performing the integration. This allows for risky work to perform on branches for longer periods of time without physicially braking the stability of a main line. This approach also makes it easier to release a product from the main line at the proverbial wink of an eye.

Every Commit Should Build the Mainline on an Integration Machine
: This is the main purpose of continuous integration. Although every commit is an ideal, on systems that have more complex setups or limited resources i.e full hardware configurations

[14] https://subversion.apache.org/
[15] Strictly speaking it is possible to achieve some form of offline repository, but it is hard to set up for most users.

and installations, it might be best to run a polling system that will only start a new build once an existing one has finished. for smaller builds, i.e. web projects or JVM libraries, every commit should be the preferred approach.

Fix Broken Builds Immediately

A cultural pathology that exists within many teams is not to fix buil dfailures immediately. Build failures should be treated with higher priority than committing new code.

Keep the Build Fast

Longer builds lead to longer feedback cycles. Break complex builds into smaller repositories or add multiple steps to a pipeline by performing complex integration testing in later stages.

Test in a Clone of the Production Environment

Modern technology makes it a lot easier to have exact replicas of production systems. CI systems shoul dbe set up to perform inegration and functional tests in such environment. This becomes even more important when trying to achieve the goal of continuous delivery and deployment.

Make it Easy for Anyone to Get the Latest Executable

Make artifacts downloadable form CI or publish to binary repositories.

Everyone can see what's happening

Graphical feedback plus notifications are strong points of a CI system. It provides insight for anyone with access to see the state of code integrations and contributions.

Automate Deployment

Continuous integrations and build is a key cornerstone ro continuous delivery. Ensure that the CI system can automatically deploy artifacts to test environments and also to production environments as appropriate.

Maintain a Single Source Repository

Fowler originally stated that *everything you need to do a build should be in there including: test scripts, properties files, database schema, install scripts, and third party libraries*, but tgiven technology changes, this has changed slightly. Many projects use a single repository, but this is not strictly necessary. Certain CI system will allow you to specify a list of repositories and some source control systems will allow you to access multiple repositories as if a single unit. For instance Subversion using svn:externals[16] and Git uses submodules[17]. What is important if you are planning to use more than one repository is that you control the commit versions of the included repositories from the reference repository otherwise repeatable builds will not be achievable. Also with the advent of artifact repositories such as Artifactory[18] and Conan[19], it is no longer necessary to keep third party libraries in the source repository, but rather specify coordinates to artifacts which can then be retrieved from such binary repositories.

[16] http://svnbook.red-bean.com/en/1.7/svn.advanced.externals.html
[17] https://git-scm.com/book/en/v2/Git-Tools-Submodules
[18] https://www.jfrog.com/artifactory/
[19] https://conan.io/

CI Cloud services

There are many cloud services available that can help you build. The most popular at the time of writing include

Travis CI
: Travis CI[20] allows one to build projects on Linux and MacOS. it is strongly integrated with GitHub[21]. Both private and public repositories from GitHub are supported, although private repositories will require a paid Travis account.

Appveyor
: Appveyor[22] is a cloud offering that specialises on building on Windows. It integrates with a variety of cloud source reposiotyr offerings including GitHub, GitLab[23], BitBucket[24], Kiln[25] and Visual Studio Team Services[26], as well as any reachable Stash, Git, Mercurial and Subversion servers.

Azure
: Azure Pipelines[27] is relative new compared to the others listed here, but it is bound to become popular due to the weight of Microsoft and GitHub behind it.

Circle CI
: Circle CI[28] is Linux only, but offers much better workflow control than Appveyor and Travis CI. It also has first class support for Docker which is important for teams that need to use a collection of Docker images.

GitLab
: GitLab Pipelines[29] offers Linux runners as part of all projects on GitLab in its cloud offering. It is possible to add your own additional runners which can be on other platforms and on your own servers.

CI Self-hosted services

Cloud services will suffice as a CI tool for many projects that are not overly complex. Once your project require multi-platform integrations, multiple container and virtual images, the cloud offerings tend to fall short. This is when it is best to roll your own CI system that is customised.

There are many different solutions on the market, but it is important to select something that follow the following principles:

[20] https://travis-ci.org/
[21] https://github.com
[22] https://www.appveyor.com/
[23] https://gitlab.com/
[24] https://bitbucket.org/
[25] http://www.fogcreek.com/fogbugz/devhub/
[26] https://azure.microsoft.com/en-us/services/visual-studio-team-services/
[27] https://azure.microsoft.com/en-us/services/devops/pipelines/
[28] https://circleci.com
[29] https://gitlab.com/help/ci/quick_start/README

CI as code
: All of the CI configuration should be stored in source control.

Scriptability
: Be able to configure environments via a script interface.

Extensibility
: The ability to add and remove plugins depdning on your own context. It should also be possible to write your own plugins for your organisational context.

Multi-platform
: Run tasks on a variety of platforms.

Reporting and trending
: Keep track of build and test history.

API
: Configure jobs, kick off builds via an API (typically REST) without having to manually resort to a UI.

If you are already using GitLab you might be interested in simply extending that with your own custom runners or you might even host your own instance of GitLab. Another very popular alternative, and one which will be referred to in the rest of this book, is Jenkins CI. Both of these confirm to the above principles.

Bibliography

[**ul**] Beck, Kent. Extreme Programming Explained. ISBN 0-201-61641-6. 1999.

[**ul**] Booch, Grady. Object-Oriented Analysis and Design with applications (2nd edition), December 1998.

[**ul**] Fowler, Martin. Continuous Integration[30]. May 2006. (Retrieved 15 August 2017).

[30] https://martinfowler.com/articles/continuousIntegration.html