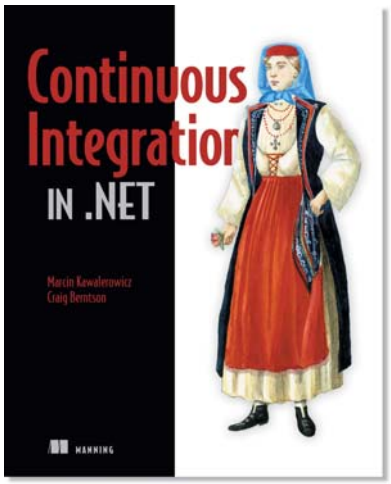
MANNING PUBLICATIONS

Introduction to Continuous Integration in .NET



Green Paper from

Continuous Integration in .NET

EARLY ACCESS EDITION

AUTHOR(S) Marcin Kawalerowicz and Craig Berntson

MEAP Release: October 2009
Softbound print: Summer 2010 (est.) | 375 pages
ISBN: 9781935182559

This green paper is taken from the book { HYPERLINK "http://manning.com/kawalerowicz/" } from Manning Publications. It introduces the essential concepts of Continuous Integration, and examines CI server possibilities. For the table of contents, the Author Forum, and other resources, go to {HYPERLINK "http://manning.com/kawalerowicz/"}

Have you ever heard the term “integration hell”? It is a situation where it takes more time to integrate the changes into the code base than to develop them. Normally the developer takes a repository snapshot to work with. While he is working on his copy the main code base changes because other developers submit their changes. But he has to eventually submit his work. That’s when all (integration) hell may break loose.

Continuous Integration (CI) is a process in software engineering aiming at minimizing this problem. It targets the best practices in software delivery: frequent integration, constant readiness, short and frequent build feedback, automated testing and so on. A full blown CI process will build, test, analyze and possibly deploy an application. This process should be triggered by every source code change. It should provide immediate feedback to the development team. The best way to accomplish all this is to set up a CI server.

If you are not using CI your process probably looks like the one in Figure 1. You are developing on your local machine and you use some kind of build tool to integrate your application. It may be simple manual compilation in Visual Studio or it might be a build script of some kind. No matter if your process is manual or automatic, the build process incorporates compilation, maybe some code generation, testing and code analysis. You push an imaginary level and it makes everything it was designed for and then stops. If the integration succeeds you pass the changes to a version control system of some sort. The whole process resembles a directed graph.

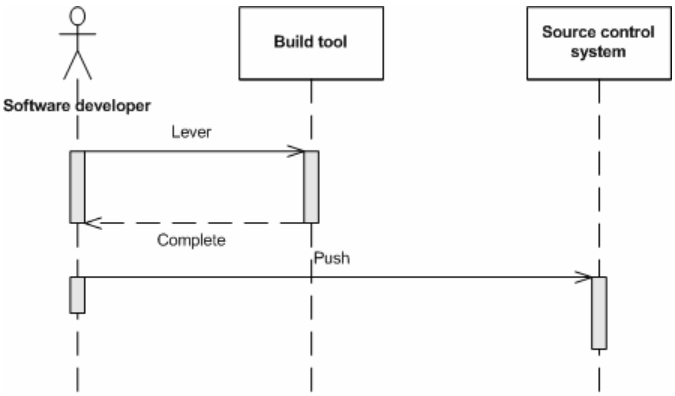


Figure 1. You push the imaginary “lever” and the build gets done. If it builds correctly you can push your code inside a source control system on a one-off basis. You get to do it all over again when you implement the next feature.

Our goal in this article is to introduce another player, a production-ready CI server, that we will choose, to work for us. The software developer still works the same as shown in Figure 1 but our new player gets more responsibility. Even if the software developer forgets to pull the “lever” to make a build and to check if everything is correct, the CI server will never forget as shown in Figure 2.

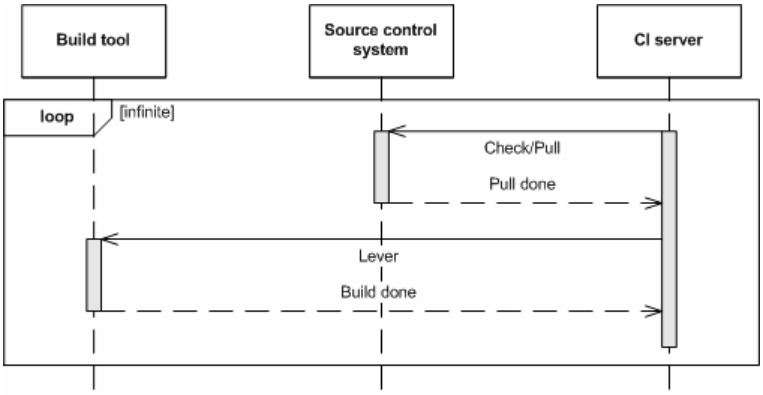


Figure 2. The CI server in action. It checks if anything new is on the source control server. If so it gets it and pulls the lever to start the build. The whole process is enclosed in an infinite loop. So whenever the software developer pushes anything into the source control repository, sooner or later it will be integrated.

There are various ways to accomplish this. First you can try to hire somebody to be a build master and act as a CI server. He can perform the build once a day or after every check-in. You can think of handcrafting a shell script (Listing 1). You can try to write your own task loop software to mimic the CI server and extend it later to a full blown tool. Or you can use one of the ready to use tools on the market. In this article we will show you the options and hopefully point you in the direction that is right for you.

Choosing the right CI server is not an easy task. You have to deal with the hardware and software aspects here. On the hardware side you have to determine if you have a separate physical machine for your CI server. Is it a full blown server with 99.9% uptime or only an old machine standing in the corner of your developers' room? If you don't have a physical machine can you get a virtualized server running somewhere where every team member has access? If you just want to check things out, it is all right to install it all together on your development machine but you have to be aware that it most likely wouldn't be a production setup.

It is strongly suggested to dedicate one separate machine for the CI server. Why? For various reasons:

- speed

- as few dependencies as possible
- clean configuration

You want your builds to run as quickly as possible. You don't want your developers waiting for your build machine. They need the feedback as soon as possible to jump in and fix a broken build.

A correctly created continuous integration process should have as few dependencies as possible. This means that your machine should be as vanilla as possible. For our .NET set-up it is the best to have only the operating system, .NET framework and probably a source control client. Some of the CI servers will need IIS or SharePoint Services to extend the functionality. This is ok too as long as the software you are installing on the CI machine is not taking part in the build process itself.

Another reason to use a dedicated continuous build machine is that you can keep it clear from any configuration changes you normally do on a development machine or on a machine that is used for something else. This way you are free from any assumptions. You can be sure that your vanilla machine stays vanilla. No topping, no icing, nothing spoiling the vanilla taste. In other words a machine brought to a know state every time the build occurs.

As for hardware requirements, it is as usual, the more the better. Your build should be fast. It should be as fast as possible. Every team member, after putting a new feature in the source control system, should wait for the CI build to finish and check it to be sure everything worked as expected. You don't want to hold the developer from his work for too long. A fast CPU and fast hard drive are in long run definitely cheaper than your developer's time.

Having a dedicated integration machine ready to host CI server you can now examine possible CI servers.

Examining the CI server possibilities

Before CI you would have had a build master or release engineer with his own build machine and integrating your software there. They would have received the code from the team and would have created working software ready for shipment. They would've done the same set of repetitive steps over and over again. The work was rather boring, error prone and took a lot of time. The release build was made rather infrequently. Quite often it led to integration problems. Then someone came up with an idea of manual integration build. With a build script in place you could make your developer personally integrate his new feature or user story into working software. But someone thought about changing the adjective to form and automated integration build. One thought lead to another and the CI server emerged. Let's try to walk this path. Maybe it will lead us to the right decision on how to establish our own CI server.

Manual integration build or your own CI server

The main reason to apply manual integration to builds is to prevent broken builds. A manual build is a technique that is losing its importance with the newest CI servers out there. Every developer has to manually run the build and integrate the software before he checks the software into repository. This task should be done in an environment as close as possible to the one that the customer will have. Running a manual integration build often involves a physical marker (often a toy) to indicate the developer that is currently holding up the build. Basically it relies on starting the same build script that the software developer starts on his machine in the integration environment to check if everything is still working. If you do this, you can be sure no code that can break the build gets to the repository. It can be of course used as a complement to a normal CI Server. Some modern CI servers are able to perform this kind of task out of the box. For example Microsoft Team Foundation Server and TeamCity, can be configured to not let code touch the source control repository without a prior integration build. But if will choose a server that does not yet have this feature, remember that a manual integration build will be made by humans and we tend to neglect, forget and make mistakes.

How about your own build server? Figure 3 shows an ordinary build run as a simple directed graph.

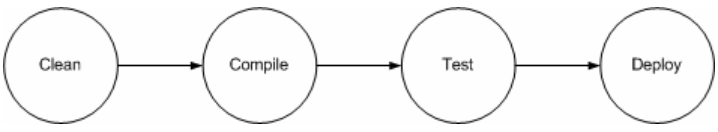


Figure 3. An ordinary build run. You clean the construction site, compile, test, and deploy. What can make this graph bend and become a task loop?

Try to think about what takes for the build process to run continuously. How about if we try squeezing it and bending it a little? Connect the end with the beginning like shown in Figure 4. What we get is something similar to task loop

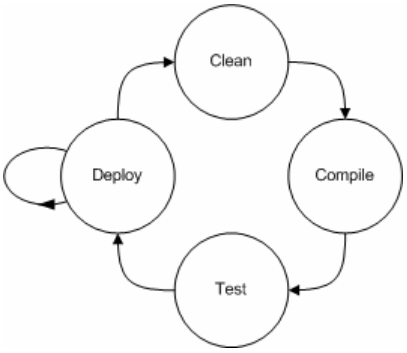


Figure 4. What is this? This graph bends and closes in a task loop. It's almost a homemade CI server diagram.

This approach has one little fault. If nothing changes we are still building the same software over and over again. What for? It's a pure waste of energy. How about adding one substantial element, something that will periodically poll the changes from the repository? If nothing interesting has changed the process will wait. But, if something did change, it will perform the build (Figure 5).

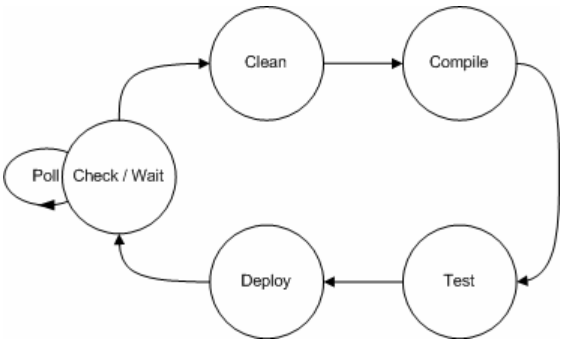


Figure 5. A simple CI server diagram. The build process bends to form a loop plus a poll/delay element.

Scripting this scenario or even writing a small program is not too hard. Let's give it a try and demonstrate the concept by creating a miniCI process in a command script. We will take this fine piece of software:

```
public class Program
{
     static int Main(string[] args)
     {
            return args[0].Equals(args[1]) ? 1 : 0;
     }
}
```

It takes two command line parameters and checks to see if they are equal. If so it returns 1. If they are not equal, it returns 0. Save it in a file called equals.cs. In the same directory create another file and call it miniCI.cmd. The code for this command file is in Listing 1.

Listing 1 The script for the miniCI demo system.

```
@echo off
cls
echo Setting up environment
if not exist work md work
if not exist deploy md deploy
if not exist equals.cs echo Dummy >> work\equals.cs
:Start
echo Checking for changes in files
fc equals.cs work\equals.cs /b > nul
 <#1 Checks for the changes in file>
if not errorlevel 1 goto :End
echo Compiling
copy equals.cs work\equals.cs
C:\Windows\Microsoft.NET\Framework\v3.5\Csc.exe work\equals.cs
 <#2 Compiles the source>
echo Testing
equals.exe test test
if errorlevel 0 goto :TestPassed
echo Test failed. Application not deployed
 <#3 Performs a test>
goto :End
:TestPassed
copy equals.exe deploy\equals.exe
 <#4 Deploys application>
echo Test passed. Application deployed.
:End
ping 1.1.1.1 -n 1 -w 5000 > nul
 <#5 Waits 5 seconds>
goto :Start
```

This simple command line script shows all the necessary steps that the CI process needs to do. It uses (or well abuses in this case) fc.exe <#1> to check if there were changes made to the source file since the last build. It uses the C# compiler <#2> to compile the source and uses the output exe to perform simple test <#3>. If the test is successful, the application is copied <#4> to a deployment folder. After that the script <#5> delays for 5 seconds (by abusing the ping command).

This script should give you a pretty good idea how a custom made CI server could be written. Our example is nothing but a funny example and has no production value. A complete CI server is a lot of work and the question is, do you really have the time to do it? It will take you a lot of time to get to the state where the production-ready CI servers are right now. They have evolved for years into quite feature rich applications.

CI servers for .NET

So you've decided not to write your own CI server. You made the right decision! You have a lot of options to choose from. There are several CI servers on the market and if you want to choose wisely you have many possible aspects to consider:

- How much money do you have to spend
- Do you want to pay the angle bracket tax (XML)
- Does it support the tools that you need
- How good is the documentation and support
- Does it do what you want it to do
- Does it do more than you need, not just now, but into the future
- Is it easy to use
- Is it cool and hip

Programs and scripts that performed a task similar to a CI server have been out there for a long time. For example, they were used in the Linux community for Kernel development ({HYPERLINK "http://test.kernel.org/tko/"}). But the era of CI servers started with CruiseControl in 2000-01. It is a tool from ToughtWorks and it emerged somewhere near the first article about continuous integration from Martin Fowler who works at ToughtWorks. CruiseControl is a Java based tool for performing continuous builds. It is widely adopted, mainly in Java community. It has a pluggable architecture and supports a wide range of source control systems. The build for CruiseControl is most often made using Ant as the build tool.

The first CI server aimed at .NET community was CruiseControl.NET (CCNet). It was a fork of the old CruiseControl made by ToughtWorks themselves. It has been on the market since 2003. It has everything the older brother has and it is written in .NET. It works as a Windows Service and has a web dashboard to present the current state. It has the ability to remotely manage the process using a system tray program called CCTray. Just like CruiseControl, CCNet is Open Source.

For year, CCNet was the automated integration server of choice for the .NET teams that didn't have enough resources for the commercial products, especially for Microsoft Team Foundation Server (TFS). One of its features is the ability to perform continuous integration builds. The coming TFS 2010 version will be more affordable for a small team (up to 5 developers). You will be able to use the TFS 2010 Basic configuration if you have purchased any MSDN Subscription.

Somewhere in between the free model of CCNet and costly model of TFS is another important player in the .NET CI server market, TeamCity from JetBrains, which was first released in 2006. The licensing scenario by TeamCity is a hybrid between free and propriety. You can start small without paying a penny for the professional license but if you grow and your needs expand you will have to pay for a license. It is written in Java, is very easy to set up and has a few features that make it very interesting to look at.

These three tools are not all the CI servers you can use in .NET. You can think of adopting Hudson, Bamboo, Electric Cloud, Anthill or one of many others. You can check a detailed CI feature matrix at the ToughtWorks wiki page ({HYPERLINK "http://confluence.public.thoughtworks.org/display/CC/CI+Feature+Matrix"}). We will look at our three players from our point of view.

Table 1 CI server matrix compares aspects of three CI servers.

	CruiseControl.NET	TFS 2010	TeamCity
Money	☺	😐	😐
XML	☹	☺	☺
Tools support	☺	😐	☺
Documentation	☺	☺	☺
CI functionality	☺	☺	☺
Additional Features	☹	☺	☺
Easy to use	☹	😐	☺
Cool (subjective)	😐	😐	☺

No matter if you choose to write your own CI server or to use open source tool or to buy something suitable, remember that the server's purpose is to help you establish a well thought out and designed CI process. A Working server will not help you if all the other pieces like version control, the build process or automated tests are not working properly.

Summary

You have to be well equipped if you want to fight integration hell. Continuous Integration is a prescription for how to win, but to do it, you will need good tools. One of them is a continuous integration server. A straightjacket you have to wear to achieve your goal. No matter if you are using a self made script, an Open Source tool or a complicated and expensive system, you need something that will get the constantly changing source code and integrate it according to a plan, to form working software. All the time and always - continuously!