CONTINUOUS INTEGRATION

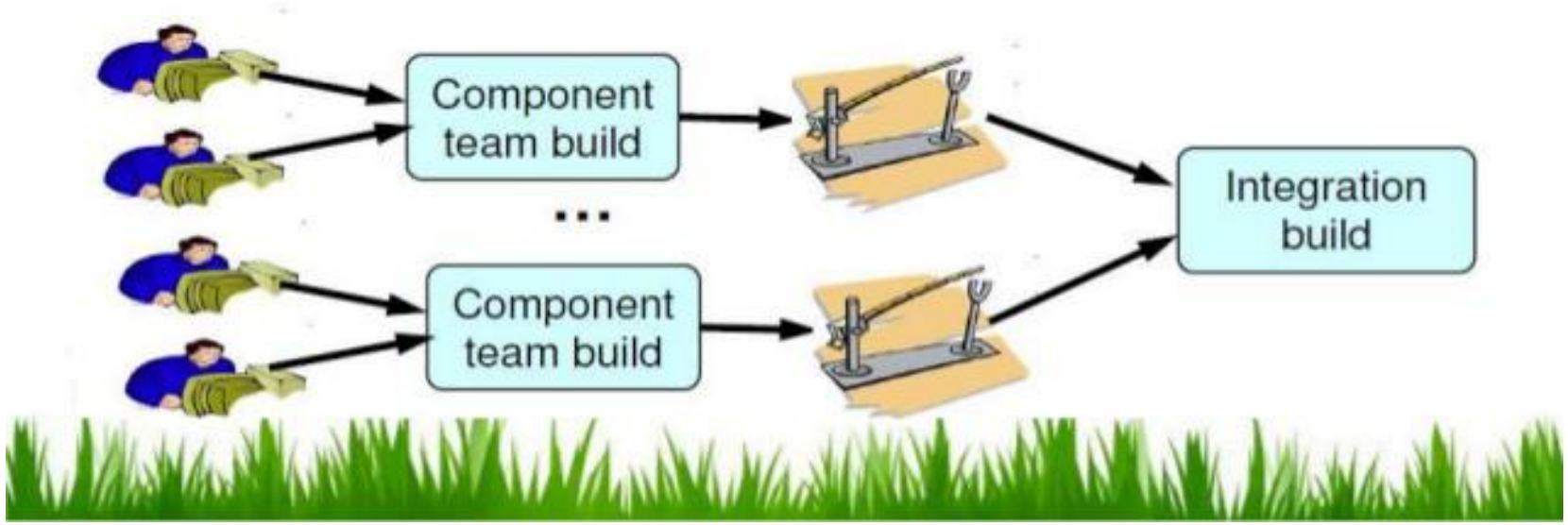
Continuous Integration (CI)

- Continuous Integration (**CI**) is the practice of merging all developer working copies to a shared code line several times a day, and validating each integration with an automated build.
- In practice, CI is often defined as having a build with unit tests that executes at every commit/ check-in to version control.

- Continuous Integration (CI) provides many benefits, including:
  - Improving code quality based on rapid feedback
  - Triggering for automated testing for every code change
  - Better managing technical debt and conducting code analysis
  - Reducing long, difficult and bug-inducing merges
  - Increasing confidence in code long before production

Continuous Integration

- Integrate the code changes by each developer so that the main branch remains up-to-date



- Continuous integration represents a paradigm shift.
- Without continuous integration, your software is broken until somebody proves it works, usually during a testing or integration stage.
- With continuous integration, your software is proven to work with every new change—and you know the moment it breaks and can fix it immediately.
- The teams that use continuous integration effectively are able to deliver software much faster, and with fewer bugs, than teams that do not.
- Bugs are caught much earlier in the delivery process when they are cheaper to fix, providing significant cost and time savings.
- Hence we consider it an essential practice for professional teams, perhaps as important as using version control.

Implementing Continuous Integration

- The practice of continuous integration relies on certain prerequisites (required as a prior condition) being in place.
- Most importantly, continuous integration depends on teams following a few essential practices.

What You Need Before You Start:-

- There are three things that you need before you can start with continuous integration.

  i. **Version Control:-** Everything in your project must be checked in to a single version control repository: code, tests, database scripts, build and deployment scripts, and anything else needed to create, install, run, and test your application.

ii. **An Automated Build:-** You must be able to start your build from the command line. You can start off with a command-line program that tells your IDE to build your software and then runs your tests, or it can be a complex collection of multistage build scripts that call one another. Whatever the mechanism, it must be possible for either a person or a computer to run your build, test, and deployment process in an automated fashion via the command line.

iii. **Agreement of the Team:-** Continuous integration is a practice, not a tool. It requires a degree of commitment and discipline from your development team. You need everyone to check in small incremental changes frequently to mainline and agree that the highest priority task on the project is to fix any change that breaks the application. If people don't adopt the discipline necessary for it to work, your attempts at continuous integration will not lead to the improvement in quality that you hope for.

A Basic Continuous Integration System :-

- *Continuous Integration* is a practice, not a tool.
- CI tools these days are extremely simple to install and get running.
- Once you have your CI tool of choice installed it should be possible to get started in just a few minutes by telling your tool where to find your source control repository, what script to run in order to compile, and run the automated commit tests for your application, and how to tell you if the last set of changes broke the software.
- The next step is for everybody to start using the CI server.

- Here is a simple process to follow.

1. Check to see if the build is already running. If so, wait for it to finish. If it fails, you'll need to work with the rest of the team to make it green before you check in.
2. Once it has finished and the tests have passed, update the code in your development environment from this version in the version control repository to get any updates.
3. Run the build script and tests on your development machine to make sure that everything still works correctly on your computer, or alternatively use your CI tool's personal build feature.

4. If your local build passes, check your code into version control.
5. Wait for your CI tool to run the build with your changes.
6. If it fails, stop what you're doing and fix the problem immediately on your development machine—go to step 3.
7. If the build passes, rejoice and move on to your next task.

If everybody on the team follows these simple steps every time they commit any change, you will know that your software works on any box with the same configuration as the CI box at all times.

Prerequisites for Continuous Integration

- For CI to be effective, the following practices will need to be in place before you start…

1. ***Check In Regularly***
2. ***Create a Comprehensive Automated Test Suite***
3. ***Keep the Build and Test Process Short***
4. ***Managing Your Development Workspace***
5. **Using Continuous Integration Software**

Check In Regularly

- The most important practice for *continuous integration* to work properly is frequent check-ins to trunk or mainline.
- Checking in your code at least a couple of times a day.
- Checking in regularly brings lots of other benefits:-
    1. It makes your changes smaller and thus less likely to break the build. It means you have a recent known good version of the software to revert to when you make a mistake or go down the wrong path.

2. It helps you to be more disciplined about your refactoring and stick to small changes that preserve behavior.
3. It helps to ensure that changes altering a lot of files are less likely to conflict with other people's work.
4. It allows developers to be more explorative, trying out ideas and discarding them by reverting back to the last committed version.
5. It forces you to take regular breaks and stretch your muscles.
6. It also means that if something catastrophic happens (such as deleting something by mistake) you haven't lost too much work.

Create a Comprehensive Automated Test Suite

- If you don’t have a comprehensive suite of automated tests, a passing build only means that the application could be compiled and assembled.
- While for some teams this is a big step, it’s essential to have some level of automated testing to provide confidence that your application is actually working.
- There are many kinds of automated tests. However, there are three kinds of tests that runs from continuous integration build: unit tests, component tests, and acceptance tests.
- These three sets of tests, combined, should provide an extremely high level of confidence that any introduced change has not broken existing functionality.

- **Unit tests** are written to test the behavior of small pieces of your application in isolation.
- They can usually be run without starting the whole application.
- They do not hit the database (if your application has one), the filesystem, or the network.
- They don’t require your application to be running in a production-like environment.
- Unit tests should run very fast—your whole suite, even for a large application, should be able to run in under ten minutes.

- **Component tests** test the behavior of several components of your application.
- Like unit tests, they don’t always require starting the whole application.
- They may hit the database, the filesystem, or other systems (which may be stubbed out).
- Component tests typically take longer to run.

- **Acceptance tests** test that the application meets the acceptance criteria decided by the business, including both the functionality provided by the application and its characteristics such as capacity, availability, security, and so on.
- Acceptance tests are best written in such a way that they run against the whole application in a production-like environment.
- Acceptance tests can take a long time to run—it's not unheard of for an acceptance test suite to take more than a day to run sequentially.

Keep the Build and Test Process Short

- If it takes too long to build the code and run the unit tests, you will run into the following problems:

1. People will stop doing a full build and running the tests before they check-in. You will start to get more failing builds.
2. The continuous integration process will take so long that multiple commits will have taken place by the time you can run the build again, so you won't know which check-in broke the build.
3. People will check in less often because they have to sit around for ages waiting for the software to build and the tests to run.

- Ideally, the compile and test process that you run prior to check-in and on your CI server should take no more than a few minutes.
- There are a number of techniques that you can use to reduce the build time.
- The first thing to consider is making your tests run faster.
- XUnit-type tools, such as JUnit and NUnit, provide a breakdown of how long each test took in their output.

Managing Your Development Workspace

- It is important for developers' productivity and sanity that their development environment is carefully managed.
- Developers should always work from a knowngood starting point when they begin a fresh piece of work.
- They should be able to run the build, execute the automated tests, and deploy the application in an environment under their control.
- This should be on their own local machine.

- Only in exceptional circumstances should you use shared environments for development.
- Running the application in a local development environment should use the same automated processes that are used in the continuous integration and testing environments and ultimately in production.
- The first prerequisite to achieve this is careful configuration management, not just of source code, but also of test data, database scripts, build scripts, and deployment scripts.
- The second step is configuration management of third-party dependencies, libraries, and components.
- The final step is to make sure that the automated tests, including smoke tests, can be run on developer machines.

Using Continuous Integration Software

- There are many products on the market that can provide the infrastructure for your automated build and test process.
- The most basic functionality of continuous integration software is to poll your version control system to see if any commits have occurred and, if so, check out the latest version of the software, run your build script to compile the software, run the tests, and then notify you of the results.

- At heart, continuous integration server software has two components.
  - The first is a long-running process which can execute a simple workflow at regular intervals.
  - The second provides a view of the results of the processes that have been run, notifies you of the success or failure of your build and test runs, and provides access to test reports, installers, and so on.
- Most CI servers include a web server that shows you a list of builds that have run and allows you to look at the reports that define the success or failure of each build.

- This sequence of build instructions should culminate in the production and storage of the resulting artifacts such as binaries or installation packages, so that testers and clients can easily download the latest good version of the software.
- Most CI servers are configurable using a web interface or through simple scripts.
- Today's advanced CI servers can distribute work across a build grid, manage the builds and dependencies of collections of collaborating components, report directly into your project management tracking system, and do lots of other useful things.

Essential Practices

- Continuous integration is a practice, not a tool, and it depends upon discipline to make it effective.
- Keeping a continuous integration system operating, particularly when dealing with large and complex CI systems, requires a significant degree of discipline from the development team as a whole.
- The objective of CI system is to ensure that software is working, all of the time.

- There are few practices that should be enforced on teams to ensure that the software is working, all of the time.
- These practices that are optional but desirable, but those listed here are mandatory for continuous integration to work.
- ***Don't Check In on a Broken Build:-*** If the build breaks, the developers responsible are waiting to fix it. They identify the cause of the breakage as soon as possible and fix it. work out what caused the breakage and fix it immediately. When this rule is broken, it inevitably takes much longer for the build to be fixed. People get used to seeing the build broken, and very quickly you get into a situation where the build stays broken all of the time.

- ***Always Run All Commit Tests Locally before Committing, or Get Your CI Server to Do It for You:-*** Running the commit tests locally is a sanity check before committing to the action. It is also a way to ensure that what we believe to work actually does.
- There are two reasons for this approach:
    1. Other people may have checked in before your last update from version control, and the combination of your new changes and theirs might cause tests to fail. If you check out and run the commit tests locally, you will identify this problem without breaking the build.
    2. A common source of errors on check-in is to forget to add some new artefact to the repository. If you follow this procedure, and your local build passes, and then your CI management system fails the *commit stage*, you know that it is either because someone checked in in the meantime, or because you forgot to add the new class or configuration file that you have just been working on into the version control system.

- ***Wait for Commit Tests to Pass before Moving On:-*** The CI system is a shared resource for the team. When a team is using CI effectively, any breakage of the build is a minor stumbling block for the team and project as a whole. Build breakages are a normal and expected part of the process. Aim is to find errors and eliminate them as quickly as possible, without expecting perfection and zero errors. If the commit succeeds, the developers can move on to their next task. If it fails, they are at hand to start determining the natureof the problem and fixing it.

- ***Never Go Home on a Broken Build:-*** It is 5:30 P.M. on Friday, you have just committed your changes. The build has broken. You have three options. You can resign yourself to the fact that you will be leaving late, and try to fix it. You can revert your changes and return to your check-in attempt next week. Or you can leave now and leave the build broken. Just to be absolutely clear, *we are not recommending that you stay late to fix the build after working hours*. Rather, we recommend that you check in regularly and early enough to give yourself time to deal with problems should they occur.

- ***Always Be Prepared to Revert to the Previous Revision:-*** we all make mistakes, so we expect that everyone will break the build from time to time. Whatever our reaction to a failed *commit stage*, it is important that we get everything working again quickly. If we can't fix the problem quickly, for whatever reason, we should revert to the previous change-set held in revision control and remedy the problem in our local environment. Airplane pilots are taught that every time they land, they should assume that something will go wrong, so they should be ready to abort the landing attempt and "go around" to make another try. Use the same mindset when checking in.

- ***Time-Box Fixing before Reverting:-*** Establish a team rule: When the build breaks on check-in, try to fix it for ten minutes. If, after ten minutes, you aren't finished with the solution, revert to the previous version from your version control system.
- ***Don't Comment Out Failing Tests:-*** Once you begin to enforce the previous rule, the result is often that developers comment out failing tests in order to get their changes checked in. This impulse is understandable, but wrong. When tests that have been passing for a while begin to fail, it can be hard to work out why. Commenting out tests that fail should always be a last resort, very rarely and reluctantly used, unless you are disciplined enough to fix it right away. It is OK to very occasionally comment out a test pending either some serious development work that needs to be scheduled or some extended discussions with the customer.

- ***Take Responsibility for All Breakages That Result from Your Changes :-*** If you commit a change and all the tests you wrote pass, but others break, the build is still broken. Usually this means that you have introduced a regression bug into the application. It is your responsibility—because you made the change—to fix all tests that are not passing as a result of your changes.
- To do CI effectively, everybody needs access to the whole codebase. If for some reasons you are forced into a situation where access to code cannot be shared with the whole team, you can manage around it through good collaboration with the people who have the necessary access. However, this is very much a second-best, and you should work hard to get such restrictions removed.

Test-Driven Development

- The idea is that when developing a new piece of functionality or fixing a bug, developers first create a test that is an executable specification of the expected behavior of the code to be written. Not only do these tests drive the application's design, they then serve both as regression tests and as documentation of the code and the application's expected behavior.