Continuous Integration and Delivery

Learning Objectives

After this lesson, students will be able to:

- ➔ Explain what CI/CD are and the benefits of using these practices and tools.
- ➔ Understand what a CI/CD pipeline is and what it does.
- ➔ Use some common tools, plugins, and approaches to build, test, and deploy full-stack applications with a CI/CD pipeline.

Opening

Who doesn't want to save time and money?

You can use continuous integration (CI) and continuous delivery (CD) to build, test, and deploy software quickly and safely, ultimately delivering value to your users more quickly.

Generally, companies begin with continuous integration (CI). Continuous delivery goes a step further than CI by automating the deployment of releases, including infrastructure and configuration changes. Continuous deployment goes a step further than continuous delivery by deploying every change to production automatically.

Continuous Integration

Don't be Disaster Girl. You can avoid disasters with continuous integration, which means that individual developers integrate their work into a main repository multiple times a day. That way, you can catch integration bugs early and collaborate more quickly. Integrating work from multiple developers involves, at a minimum, building and testing the application whenever a change is made by a developer (i.e., continuously).

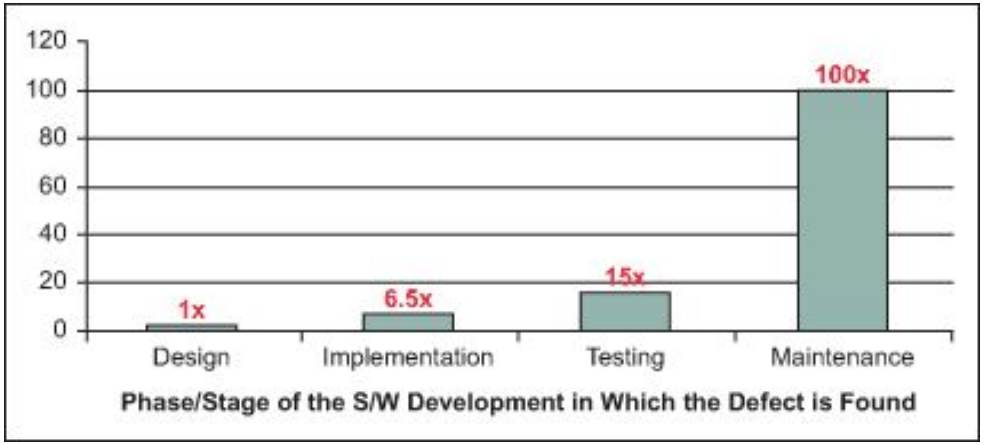
Commit Small Changes Often

This is one of the cardinal rules of CI. Developers should be committing to the mainline branch often – at least every day – and changes should be as small as possible so that, when a change breaks the build, anyone can quickly pinpoint the specific issue.

With many changes happening all the time, a fast build process is critical so that developers get feedback about broken builds right away.

How Will Continuous Integration Save Us Time and Money?

Like your mom always said, the early bird catches the worm. Finding bugs earlier in the software development life cycle (SDLC) is much less costly than finding bugs later.



Relative Costs to Fix Software Defects (Source: IBM Systems Sciences Institute)

Real Cases:

Samsung Note 7

This concept is fundamental to why we do CI/CD, so let's discuss a real-world example:

Ever heard of the Samsung Note 7 fiasco? Note 7 phones had a faulty battery management system that caused them to catch on fire! How much do you think Samsung would have saved if it had caught the bug in an early stage of development?

Answer: This bug cost Samsung nearly $17 billion! If it had caught it earlier, it also could've saved its reputation and many headaches.

Build Pipeline

So, how do we find bugs early? By testing early and often. You really don't want to have to think too much about *when* and *how* to run tests. Most modern software development teams would rather run tests automatically when there's any kind of change to the code, specifically in the mainline branch where all developers integrate their code for the next release.

A build pipeline is the automated process that watches for changes. When a change is detected, the build pipeline builds the application, tests it, and possibly does other stuff, too. Pipelines can have stages, and each stage can have steps. Steps in your pipeline can execute Maven, Gradle, shell scripts, and more.

Discussion: Build Pipeline

What do you think are the three characteristics of an ideal build pipeline?

1. Fast: You want your build pipeline to give you near-instant feedback on whether your tests pass or fail. This is so that you can fix problems right away while you are actively thinking and working on the feature.
2. Repeatable: You always want to run the same process to build and test your application, and you always want to run that process in the same build environment. That will reduce the number of things that change from one build to the next so that, when bugs come up and you need to troubleshoot, there's less to worry about.
3. Reliable: You want to set up your build pipeline and tests so that, if there's a failure, it's because of the application being built and tested rather than a problem in your test environment or the build pipeline itself. Common issues that lead to "false positives" include bad data in your database that weren't cleaned up from the last test run, or some other kind of contention for a shared resource (such as files on a shared file system).

Create a Build Pipeline with Jenkins

A very popular tool for building CI/CD pipelines (which we'll be using ourselves in this course) is Jenkins. Jenkins is an open-source automation server that can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software.

Other common tools include GitLab CI, Atlassian Bamboo, and JetBrains TeamCity.

Group Exercise:

Create a Build Pipeline with Jenkins

10 min

Let's do a bit of Jenkins research. With a partner, take a look at the Jenkins documentation (or other Jenkins resources). Answer the following questions:

- How does a Jenkins pipeline work?
- What is a Jenkinsfile?
- What's the difference between a declarative and scripted Jenkins pipeline?

Did the docs raise any other questions for you or leave you wanting more? Jot down those questions and we'll discuss them as a class!

Jenkins Plugins

A core feature of Jenkins are plugins, which allow you to extend Jenkins' basic functionality.

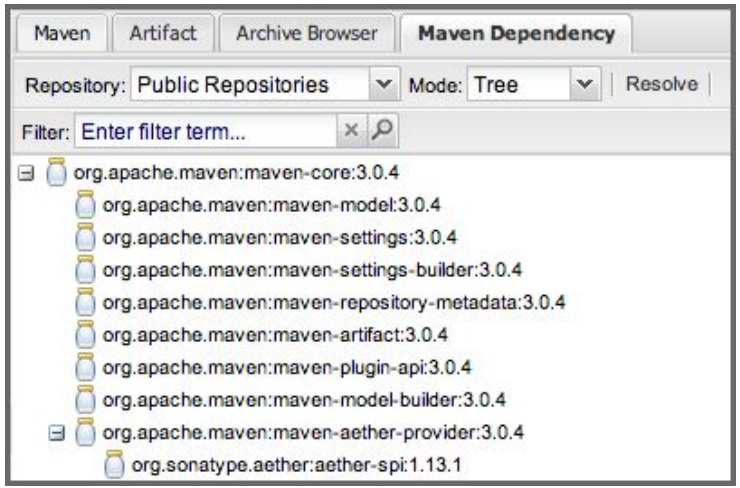
Jenkins plugins come in all shapes, sizes, and purposes. With a partner, research the following Jenkins plugins:

- Global Build Stats Plugin
- Job Generator Plugin
- Performance Plugin

Want to make sure your code is high quality? Jenkins pipelines will also often include plugins such as SonarQube and Find Bugs that scan code for bad coding practices and security vulnerabilities.

Moving to Nexus

Once your application is built (usually in the first stage of your pipeline), your application artifact (e.g., an rpm, war/jar, ear file, or image) will be deployed to a repository such as Nexus. By uploading the artifact to Nexus, the next stages in the pipeline can actually run on different servers and pull the artifact(s) from the repo to operate on them. Repositories like Nexus also offer other features for dependency management and audit controls.



View a Component's Dependencies in Nexus From Inspecting the Component Dependencies | Sonatype Nexus Documentation

Integration Testing

Discussion:

Integration Testing

Knowledge check: *What's the difference between unit tests and integration tests?*

Integration Testing

Unit tests with mocked dependencies usually run very quickly.

On the other hand, integration tests usually take longer to run in a pipeline. That's because it takes time for the pipeline to initialize all dependencies for integration tests. That's why you'll have staged pipelines, where unit tests run first to give you quick feedback. Integration tests run next, possibly in a completely different testing/QA environment.

Finding Bugs That Unit Tests Won't Find

Sometimes, the whole is less than the sum of its parts. Writing unit tests to test your components is great, but other kinds of bugs can pop up when changes from several developers working on a codebase are merged, or when components are integrated in the broader application. Even if all of the unit tests pass, when you bring the units together, they may not work together as intended.

Discussion:
Integration Testing

Knowledge check: *What might be some common sources of integration bugs?*

Finding Bugs That Unit Tests Won’t Find

- Merge conflicts: When merging code from several developers into a single branch, there are sometimes conflicts that can't be merged automatically. A human will need to step in and merge manually. This manual step can lead to bugs if not done perfectly. The larger the change, the more likely you are to encounter merge conflicts, so try to keep commits and PRs small.
- Using real components as dependencies: Integration tests will usually aim to use *real components as dependencies* instead of mocking the dependencies like we do in unit tests. By using real components, such as databases, application servers, containers, etc., integration tests are likely to surface problems that unit tests (with mocked components) simply wouldn't be capable of exposing.
- Incompatible interfaces or component behaviors: This is the broadest category of integration bugs. Unit tests may pass for individual components, but integration tests (operating across multiple components) may fail if the components simply weren't designed to play well with each other. For example, one component (a REST endpoint) may accept dashes/hyphens in a user's last name, while another (a database table) might only accept alphanumeric characters. Passing a user's last name with hyphens to the REST endpoint and then to the database would fail an integration test.

Integration Testing Frameworks and Tools

Thankfully, there are frameworks and tools to automate integration testing and, in particular, deal with the initialization of dependencies such as databases, application servers, containers, infrastructure, etc. Typical integration tests will rely on a number of these tools and frameworks to initialize a database with test data, initialize infrastructure and/or container(s) for your application, initialize your application (or sometimes just a subset of the components to test), and then run tests. After running the test suite, the dependencies should be shut down gracefully and/or cleaned up.

Integration Testing Frameworks and Tools

Common integration testing tools include:

- DbUnit
- TestNG
- Selenium
- Arquillian
- Spring Testing
- and others

We won't cover all of these in this course, but we wanted to make you're aware that they exist and provide you with links for further reading on your own time.

Continuous Delivery

Continuous Delivery

Continuous delivery (CD) is an approach that builds on continuous integration. CD ensures that our code is always in a deployable state, even with thousands of developers making changes regularly. Teams produce software in short cycles so they can release the software whenever necessary. CD helps teams save time and money and makes delivering changes less risky.

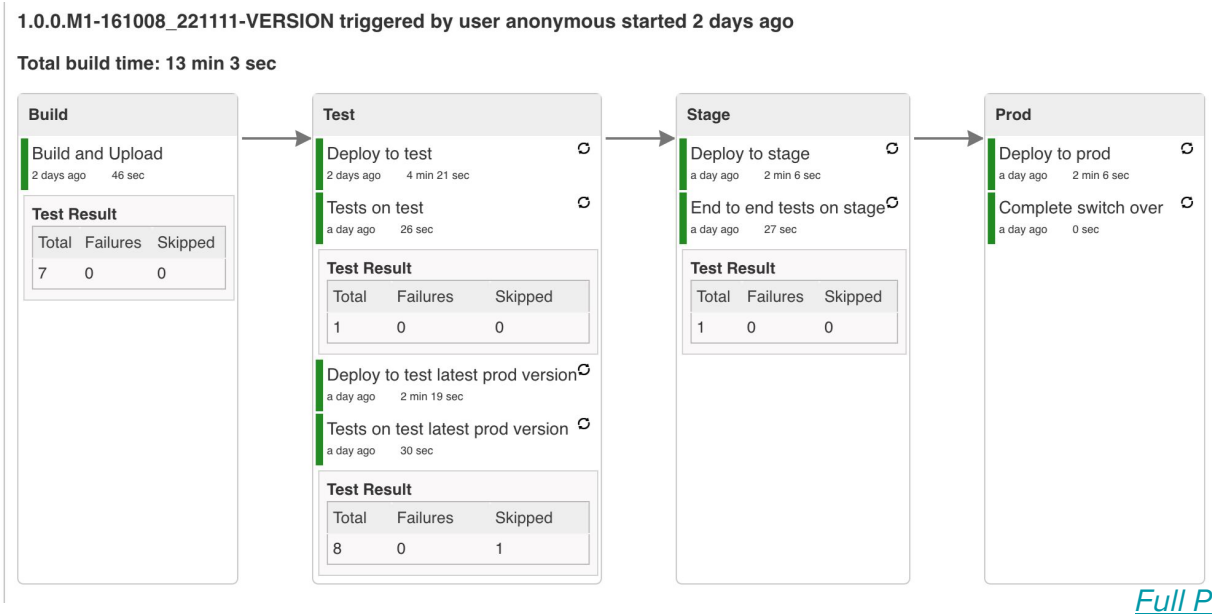
Development, Staging, and Production Environments

In most enterprises, there are several different environments where a new version of the application (a release candidate) needs to be built, tested, and/or reviewed before deployment to the production environment.

1. Build: Oftentimes, you'll see a build environment where the code is compiled and application artifacts are constructed (e.g., an rpm, war file, or image). Sometimes, unit tests will run here, too. Once built, the artifact is pushed to a repository and should never change.
2. Test: The artifact would then be pulled from the repository and installed in a test environment for integration testing, performance testing, and other testing purposes.
3. Stage: A staging environment is the last environment before promoting the artifact to production. That's where more tests can be run (automated or not) and/or human users may perform some kind of acceptance testing.

Development, Staging, and Production Environments

In CI/CD, this whole process — including deployments from build to test to stage and (optionally) to prod — should be automated by your build pipeline. Here's an example of what that pipeline might look like in Jenkins:



Full Pipeline View

The Other CD

In some companies, there will need to be a final approval or review by a human before software can actually be deployed to production. If that isn't the case, you can also fully automate that final deployment step (a practice known as continuous deployment).

Continuous delivery ensures that code can be rapidly and safely deployed to production and that applications function as expected through rigorous automated testing.

Continuous deployment is the next step of continuous delivery: Every change that passes the automated tests is deployed to production automatically.

Discussion:

Continuous Deployment

Knowledge check: *When is continuous deployment helpful? When might you not want to use continuous deployment?*

Wrap-ups

Partner Exercise:

Conclusion

5 min

Phew! Doesn't all of this information about DevOps make you grateful for all the operations that keep your favorite software (and memes) running smoothly?

With a partner, discuss the following prompts:

- Explain how continuous delivery and continuous integration help teams save time and money.
- Define:
  - The build pipeline
  - Build, test, and stage environments

Resources

- Continuous Delivery
- An Introduction to Continuous Integration, Delivery, and Deployment
- Defect Prevention: Reducing Costs and Enhancing Quality
- Continuous Integration
- Dev Leaders Compare Continuous Delivery vs. Continuous Deployment vs. Continuous Integration

Guided Walk-Through:
Jenkins

Follow along at this repo:

week5_lesson1/jenkins-in-docker.md