UNIT – I

Introduction: Introduction, Agile development model, DevOps, and ITIL. DevOps process and Continuous Delivery, Release management, Scrum, Kanban, delivery pipeline, bottlenecks, examples

Introducing DevOps

DevOps is, by definition, a field that spans several disciplines. It is a field that is very practical and hands-on, but at the same time, you must understand both the technical background and the nontechnical cultural aspects.

The word "DevOps" is a combination of the words "development" and "operation". This wordplay already serves to give us a hint of the basic nature of the idea behind DevOps. It is a practice where collaboration between different disciplines of software development is encouraged.

The origin of the word DevOps and the early days of the DevOps movement can be tracked rather precisely: Patrick Debois is a software developer and consultant.

The DevOps movement has its roots in Agile software development principles.

DevOps can be said to relate to the first principle, "Individuals and interactions over processes and tools."

DevOps, then, tends to emphasize that interactions between individuals are very important, and that technology might possibly assist in making these interactions happen and tear down the walls inside organizations.

The first principle favors interaction between people over tools. If we use the tools properly, they can facilitate all of the desired properties of an Agile workplace.

A very simple example might be the choice of systems used to report bugs. Quite often, development teams and quality assurance teams use different systems to handle tasks and bugs. This creates unnecessary friction between the teams and further separates them when they should really focus on working together instead. The operations team might, in turn, use a third system to handle requests for deployment to the organization's servers.

An engineer with a DevOps mindset, on the other hand, will immediately recognize all three systems as being workflow systems with similar properties.

Another core goal of DevOps is automation and Continuous Delivery. Simply put, automating repetitive and tedious tasks leaves more time for human interaction, where true value can be created.

For DevOps processes must be fast. We need to consider time to market in the larger perspective, and simply stay focused on our tasks in the smaller perspective. This line of thought is also held by the Continuous Delivery movement.

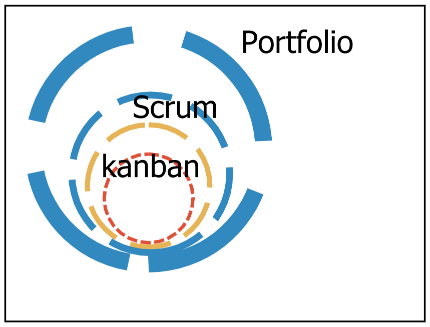
DevOps engineers work on making enterprise processes faster, more efficient, and more reliable. Repetitive manual labor, which is error prone, is removed whenever possible.

Agile development model

There are several different cycles in Agile development, from the Portfolio level through to the Scrum and Kanban cycles and down to the Continuous Integration cycle. The emphasis on which cadence work happens in is a bit different depending on which Agile framework you are working with. Kanban emphasizes the 24-hour cycle and is popular in operations teams. Scrum cycles can be between two to four weeks and are often used by development teams using the Scrum Agile process.

Longer cycles are also common and are called **Program Increments**, which span several Scrum Sprint cycles, in Scaled Agile Framework.

The Agile wheel of wheels



DevOps must be able to support all these cycles. This is quite natural given the central theme of DevOps: cooperation between disciplines in an Agile organization.

DevOps, and ITIL

DevOps fits well together with many frameworks for Agile or Lean enterprises. Scaled Agile Framework, or SAFe® , specifically mentions DevOps. There is nearly never any disagreement between proponents of different Agile practices and DevOps since DevOps originated in the Agile environments. The story is a bit different with ITIL, though.

ITIL, which was formerly known as Information Technology Infrastructure Library, is a practice used by many large and mature organizations.

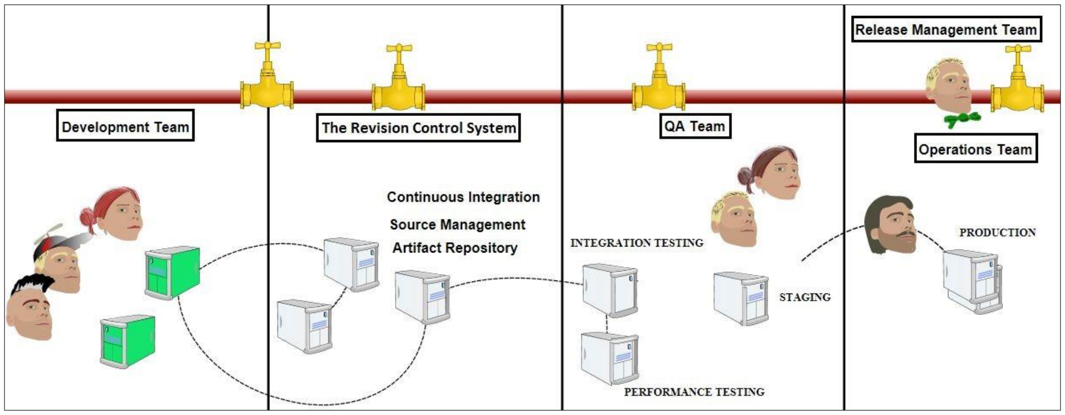
ITIL is a large framework that formalizes many aspects of the software life cycle. While DevOps and Continuous Delivery hold the view that the change sets we deliver to production should be small and happen often, at first glance, ITIL would appear to hold the opposite view. It should be noted that this isn't really true. Legacy systems are quite often monolithic, and in these cases, you need a process such as ITIL to manage the complex changes often associated with large monolithic systems.

In any case, many of the practices described in ITIL translate directly into corresponding DevOps practices. ITIL prescribes a configuration management system and a configuration management database. These types of systems are also integral to DevOps

DevOps process and Continuous Delivery

An overview

The following image is the overview of Continuous Delivery pipeline. For the time being, it is enough to understand that when we work with DevOps, we work with large and complex processes in a large and complex context.



The depending on the size of the organization and the complexity of the products that are being developed.

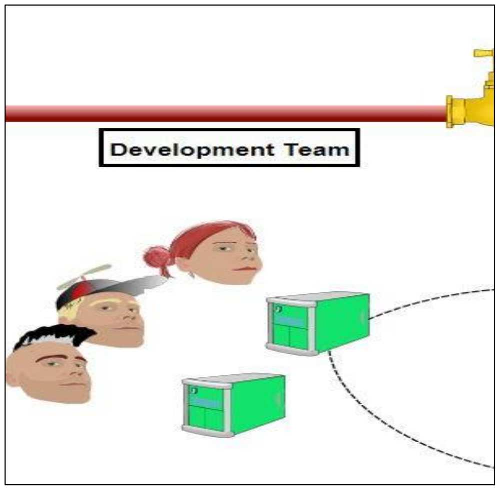
The early parts of the chain, that is, the developer environments and the Continuous Integration environment, are normally very similar. The number and types of testing environments vary greatly. The production environments also vary greatly.

The developers

The developers work on their workstations. They develop code and need many tools to be efficient.

The following detail from the previous larger Continuous Delivery pipeline overview illustrates the development team.

Ideally, they would each have production-like environments available to work with locally on their workstations or laptops. Depending on the type of software that is being developed, this might actually be possible, but it's more common to simulate, or rather, mock, the parts of the production environments that are hard to replicate. This might, for example, be the case for dependencies such as external payment systems or phone hardware.



When you work with DevOps, you might, depending on which of its two constituents you emphasized on in your original background, pay more or less attention to this part of the Continuous Delivery pipeline. If you have a strong developer background, you appreciate the convenience of a prepackaged developer environment.

For example, and work a lot with those. This is a sound practice, since otherwise developers might spend a lot of time creating their development environments. Such a prepackaged environment might, for instance, include a specific version of the Java Development Kit and an integrated development environment, such as Eclipse. If you work with Python, you might package a specific Python version, and so on

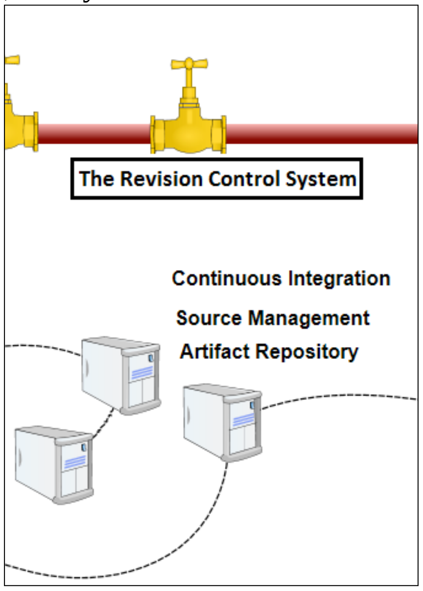
Keep in mind that we essentially need two or more separately maintained environments. The preceding developer environment consists of all the development tools we need. These will not be installed on the test or production systems. Further, the developers also need some way to deploy their code in a production-like way.
This can be a virtual machine provisioned with Vagrant running on the developer's machine, a cloud instance running on AWS, or a Docker container: there are many ways to solve this problem.

The revision control system

The revision control system is often the heart of the development environment. The code that forms the organization's software products is stored here. It is also common to store the configurations that form the infrastructure here. If you are working with hardware development, the designs might also be stored in the revision control system.

The following image shows the systems dealing with code, Continuous Integration, and artifact storage in the Continuous Delivery pipeline in greater detail:

For such a vital part of the organization's infrastructure, there is surprisingly little variation in the choice of product. These days, many use Git or are switching to it, especially those using



proprietary systems reaching end-of-life.

Regardless of the revision control system you use in your organization, the choice of product is only one aspect of the larger picture.

You need to decide on directory structure conventions and which branching strategy

The build server

The build server is conceptually simple. It might be seen as a glorified egg timer that builds your source code at regular intervals or on different triggers The most common usage pattern is to have the build server listen to changes in the revision control system. When a change is noticed, the build server updates its local copy of the source from the revision control system. Then, it builds the source and performs optional tests to verify the quality of the changes. This process is called Continuous Integration.

The artifact repository

When the build server has verified the quality of the code and compiled it into deliverables, it is useful to store the compiled binary artifacts in a repository. This is normally not the same as the revision control system.

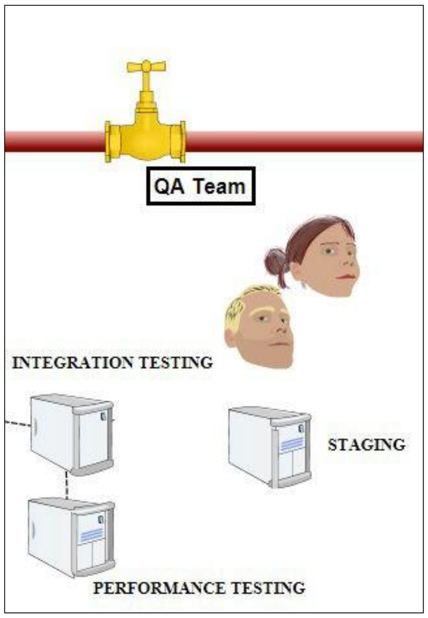
In essence, these binary code repositories are file systems that are accessible over the HTTP protocol. Normally, they provide features for searching and indexing as well as storing metadata, such as various type identifiers and version information about the artifacts.

Package managers

Linux servers usually employ systems for deployment that are similar in principle but have some differences in practice

Test environments

After the build server has stored the artifacts in the binary repository, they can be installed from there into test environments.



Test environments should normally attempt to be as production-like as is feasible. Therefore, it is desirable that the they be installed and configured with the same methods as production servers

Staging/production

Staging environments are the last line of test environments. They are interchangeable with production environments. You install your new releases on the staging servers, check that everything works, and then swap out your old production servers and replace them with the staging servers, which will then become the new production servers. This is sometimes called the blue-green deployment strategy.

Release management,

We have so far assumed that the release process is mostly automatic. This is the dream scenario for people working with DevOps.

This dream scenario is a challenge to achieve in the real world. One reason for this is that it is usually hard to reach the level of test automation needed in order to have complete confidence in automated deploys. Another reason is simply that the cadence of business development doesn't always the match cadence of technical development.

How this is done in practice varies, but deployment systems usually have a way to support how to describe which software versions to use in different environments.

The integration test environments can then be set to use the latest versions that havebeen deployed to the binary artifact repository. The staging and production servers have particular versions that have been tested by the quality assurance team.

Scrum, Kanban, and the Delivery pipeline,

Continuous Delivery pipeline support Agile processes such as Scrum and Kanban.

Scrum focuses on sprint cycles, which can occur biweekly or monthly. Kanban can besaid to focus more on shorter cycles, which can occur daily.

From a software-deployment viewpoint, both Scrum and Kanban are similar. Bothrequire frequent hassle-free deployments. From a DevOps perspective, a change starts propagating through the Continuous Delivery pipeline toward test systemsand beyond when it is deemed ready enough to start that journey.

Scrum is a framework for project management that emphasizes teamwork, accountability and iterative progress toward a well-defined goal. The framework begins with a simple premise: Start with what can be seen or known.

What is kanban? Kanban is a popular framework used to implement agile and DevOps software development. It requires real-time communication of capacity and full transparency of work. Work items are represented visually on a kanban board, allowing team members to see the state of every piece of work at any time.
Toyota has six rules for the effective application of Kanban: 1) Never pass on defective products; 2) Take only what is needed; 3) Produce the exact quantity required; 4) Level the production; 5) Fine-tune production; and 6) Stabilise and rationalise the process
Kanban is a process management tool that visualises the status of each job on a company's radar, and controls the flow of production from customer requests back to the warehouse

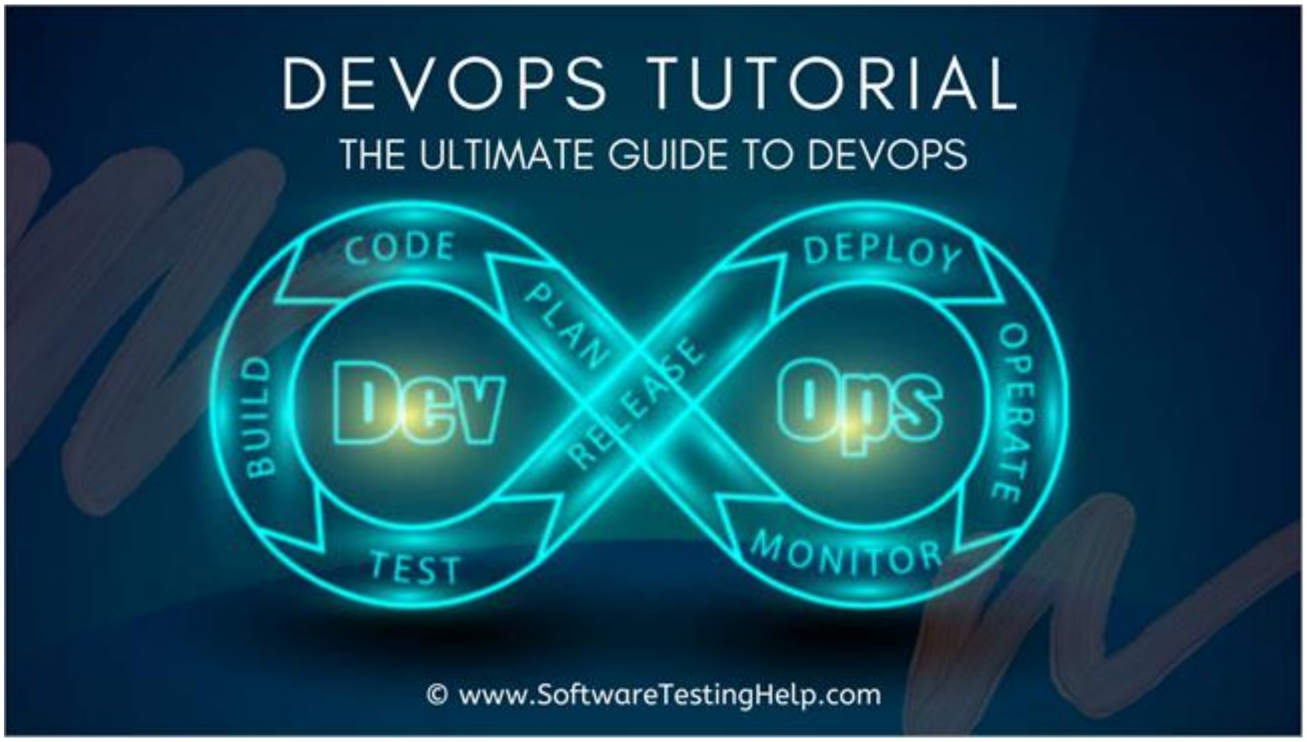
Kanban is a project management method that helps visualize tasks, while Scrum is a method that provides structure to the team and schedule. Kanban and Scrum are project management methodologies that complete project tasks in small increments and emphasize continuous improvement

The key difference between Agile and Scrum is that Agile is a philosophy about how to successfully deliver software to a customer, while Scrum is a proven methodology for software development teams to follow.
A team's success with scrum depends on five values: commitment, courage, focus, openness, and respect.

Our pipeline can manage both the following types of scenarios:

- The build server supports the generation of the objective code quality metricsthat we need in order to make decisions. These decisions can either be madeautomatically or be the basis for manual decisions.
- The deployment pipeline can also be directed manually. This can be handledwith an issue management system, via configuration code commits, or both.

So, again, from a DevOps perspective, it doesn't really matter if we use Scrum, Scaled Agile Framework, Kanban, or another method within the lean or Agile frameworks. Even a traditional Waterfall process can be successfully managed—DevOps serves all



UNIT-2

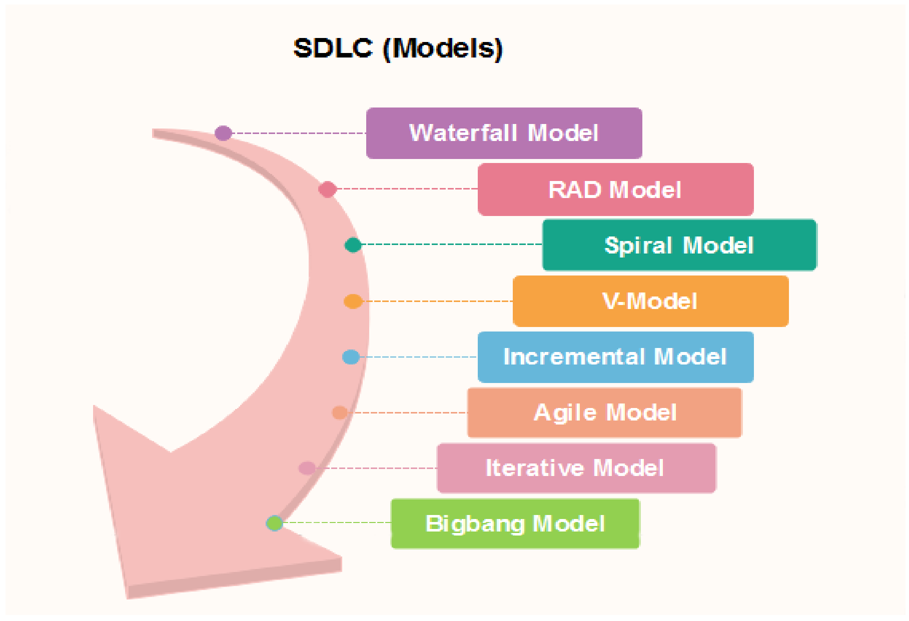
Software development models and DevOps: DevOps Lifecycle for Business Agility, DevOps, and Continuous Testing. DevOps influence on Architecture: Introducing software architecture, The monolithic scenario, Architecture rules of thumb, The separation of concerns, Handling database migrations, Micro services, and the data tier, DevOps, architecture, and resilience

Software development models and DevOps

Software Development life cycle (SDLC) is a spiritual model used in project management that defines the stages include in an information system development project, from an initial feasibility study to the maintenance of the completed application.

There are different software development life cycle models specify and design, which are followed during the software development phase. These models are also called "**Software Development Process Models**." Each process model follows a series of phase unique to its type to ensure success in the step of software development.

Here, are some important phases of SDLC life cycle:



Waterfall Model

The waterfall is a universally accepted SDLC model. In this method, the whole process of software development is divided into various phases.

The waterfall model is a continuous software development model in which development is seen as flowing steadily downwards (like a waterfall) through the steps of requirements analysis, design, implementation, testing (validation), integration, and maintenance.

Linear ordering of activities has some significant consequences. First, to identify the end of a phase and the beginning of the next, some certification techniques have to be employed at the end of each step. Some verification and validation usually do this mean that will ensure that the output of the stage is consistent with its input (which is the output of the previous step), and that the output of the stage is consistent with the overall requirements of the system.

RAD Model

RAD or Rapid Application Development process is an adoption of the waterfall model; it targets developing software in a short period. The RAD model is based on the concept that a better system can be developed in lesser time by using focus groups to gather system requirements.

- Business Modeling
- Data Modeling
- Process Modeling
- Application Generation
- Testing and Turnover

Spiral Model

The spiral model is a **risk-driven process model**. This SDLC model helps the group to adopt elements of one or more process models like a waterfall, incremental, waterfall, etc. The spiral technique is a combination of rapid prototyping and concurrency in design and development activities.

Each cycle in the spiral begins with the identification of objectives for that cycle, the different alternatives that are possible for achieving the goals, and the constraints that exist. This is the first quadrant of the cycle (upper-left quadrant).

The next step in the cycle is to evaluate these different alternatives based on the objectives and constraints. The focus of evaluation in this step is based on the risk perception for the project.

The next step is to develop strategies that solve uncertainties and risks. This step may involve activities such as benchmarking, simulation, and prototyping.

V-Model

In this type of SDLC model testing and the development, the step is planned in parallel. So, there are verification phases on the side and the validation phase on the other side. V-Model joins by Coding phase.

Incremental Model

The incremental model is not a separate model. It is necessarily a series of waterfall cycles. The requirements are divided into groups at the start of the project. For each group, the SDLC model is followed to develop software. The SDLC process is repeated, with each release adding more functionality until all requirements are met. In this method, each cycle act as the maintenance phase for the previous software release. Modification to the incremental model allows development cycles to overlap. After that subsequent cycle may begin before the previous cycle is complete.

Agile Model

Agile methodology is a practice which promotes continues interaction of development and testing during the SDLC process of any project. In the Agile method, the entire project is divided into small incremental builds. All of these builds are provided in iterations, and each iteration lasts from one to three weeks.

Any agile software phase is characterized in a manner that addresses several key assumptions about the bulk of software projects:

1. It is difficult to think in advance which software requirements will persist and which will change. It is equally difficult to predict how user priorities will change as the project proceeds.
2. For many types of software, design and development are interleaved. That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to think about how much design is necessary before construction is used to test the configuration.
3. Analysis, design, development, and testing are not as predictable (from a planning point of view) as we might like.

Iterative Model

It is a particular implementation of a software development life cycle that focuses on an initial, simplified implementation, which then progressively gains more complexity and a broader feature set until the final system is complete. In short, iterative development is a way of breaking down the software development of a large application into smaller pieces.

Big bang model

Big bang model is focusing on all types of resources in software development and coding, with no or very little planning. The requirements are understood and implemented when they come.

This model works best for small projects with smaller size development team which are working together. It is also useful for academic software development projects. It is an ideal model where requirements are either unknown or final release date is not given.

Prototype Model

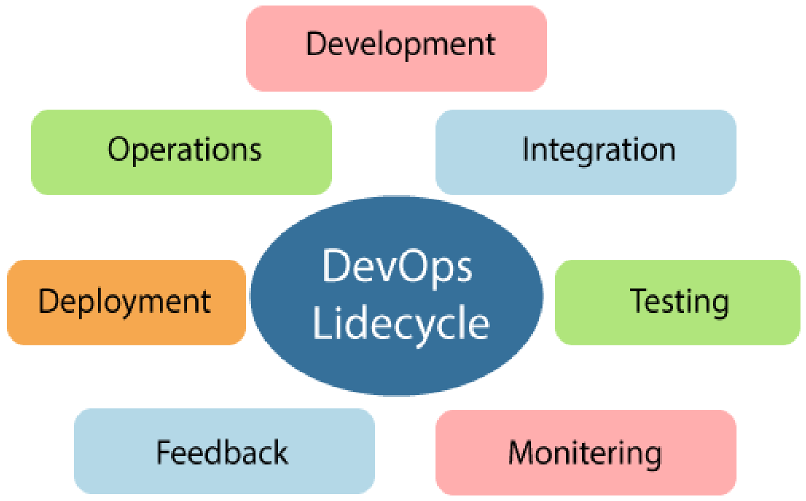
The prototyping model starts with the requirements gathering. The developer and the user meet and define the purpose of the software, identify the needs, etc.

A '**quick design**' is then created. This design focuses on those aspects of the software that will be visible to the user. It then leads to the development of a prototype. The customer then checks the prototype, and any modifications or changes that are needed are made to the prototype.

> Looping takes place in this step, and better versions of the prototype are created. These are continuously shown to the user so that any new changes can be updated in the prototype. This process continue until the customer is satisfied with the system. Once a user is satisfied, the prototype is converted to the actual system with all considerations for quality and security.

DevOps Lifecycle

DevOps defines an agile relationship between operations and Development. It is a process that is practiced by the development team and operational engineers together from beginning to the final stage of the product.



Learning DevOps is not complete without understanding the DevOps lifecycle phases. The DevOps lifecycle includes seven phases as given below:

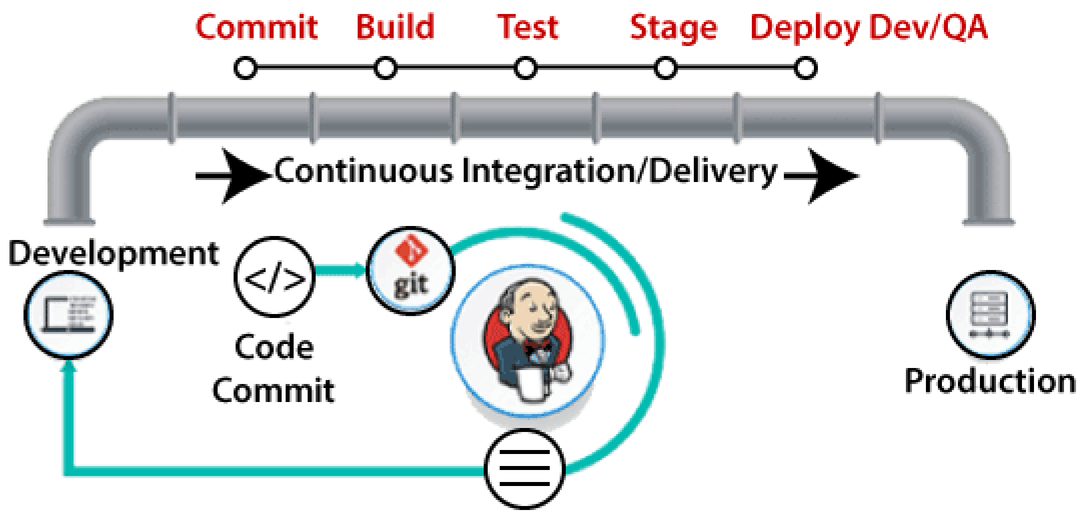
1) Continuous Development

This phase involves the planning and coding of the software. The vision of the project is decided during the planning phase. And the developers begin developing the code for the application. There are no DevOps tools that are required for planning, but there are several tools for maintaining the code.

2) Continuous Integration

This stage is the heart of the entire DevOps lifecycle. It is a software development practice in which the developers require to commit changes to the source code more frequently. This may be on a daily or weekly basis. Then every commit is built, and this allows early detection of problems if they are present. Building code is not only involved compilation, but it also includes **unit testing, integration testing, code review**, and **packaging**.

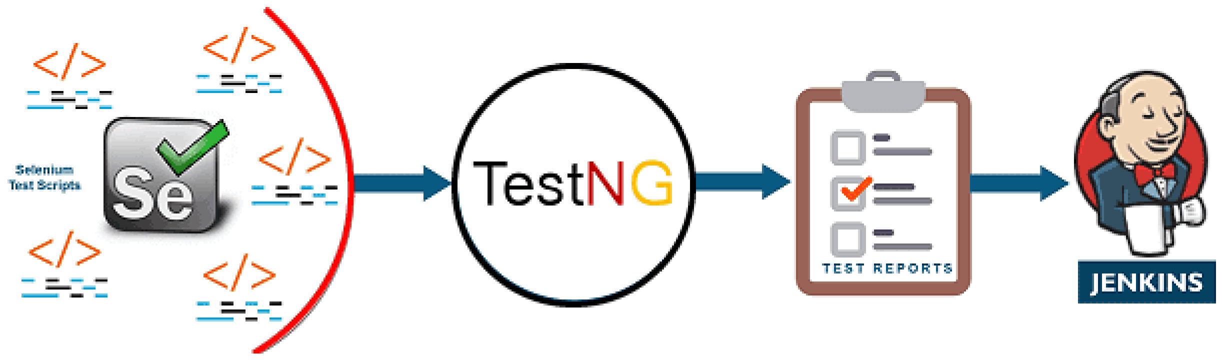
The code supporting new functionality is continuously integrated with the existing code. Therefore, there is continuous development of software. The updated code needs to be integrated continuously and smoothly with the systems to reflect changes to the end-users.



Jenkins is a popular tool used in this phase. Whenever there is a change in the Git repository, then Jenkins fetches the updated code and prepares a build of that code, which is an executable file in the form of war or jar. Then this build is forwarded to the test server or the production server.

3) Continuous Testing

This phase, where the developed software is continuously testing for bugs. For constant testing, automation testing tools such as **TestNG, JUnit, Selenium**, etc are used. These tools allow QAs to test multiple code-bases thoroughly in parallel to ensure that there is no flaw in the functionality. In this phase, **Docker** Containers can be used for simulating the test environment.



Selenium does the automation testing, and TestNG generates the reports. This entire testing phase can automate with the help of a Continuous Integration tool called **Jenkins**.

Automation testing saves a lot of time and effort for executing the tests instead of doing this manually. Apart from that, report generation is a big plus. The task of evaluating the test cases that failed in a test suite gets simpler. Also, we can schedule the execution of the test cases at predefined times. After testing, the code is continuously integrated with the existing code.

4) Continuous Monitoring

Monitoring is a phase that involves all the operational factors of the entire DevOps process, where important information about the use of the software is recorded and carefully processed to find out trends and identify problem areas. Usually, the monitoring is integrated within the operational capabilities of the software application.

It may occur in the form of documentation files or maybe produce large-scale data about the application parameters when it is in a continuous use position. The system errors such as server not reachable, low memory, etc are resolved in this phase. It maintains the security and availability of the service.

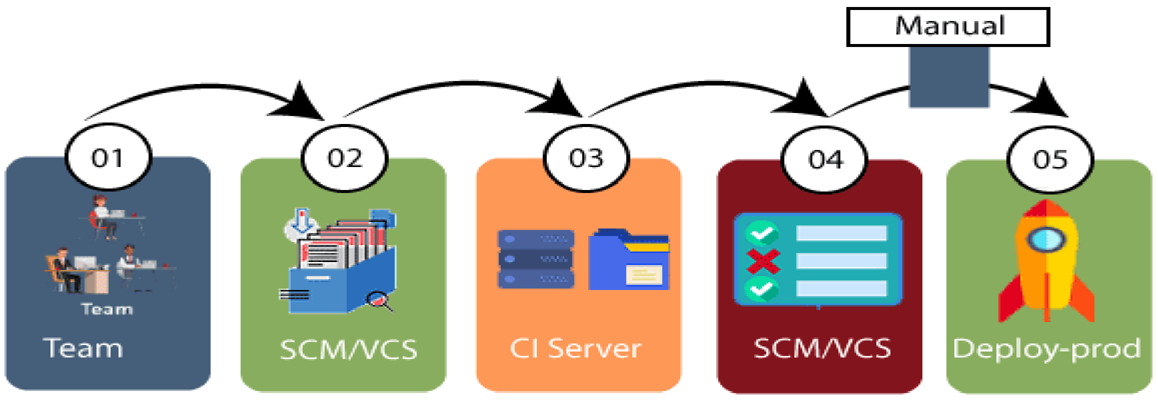
5) Continuous Feedback

The application development is consistently improved by analyzing the results from the operations of the software. This is carried out by placing the critical phase of constant feedback between the operations and the development of the next version of the current software application.

The continuity is the essential factor in the DevOps as it removes the unnecessary steps which are required to take a software application from development, using it to find out its issues and then producing a better version. It kills the efficiency that may be possible with the app and reduce the number of interested customers.

6) Continuous Deployment

In this phase, the code is deployed to the production servers. Also, it is essential to ensure that the code is correctly used on all the servers.



The new code is deployed continuously, and configuration management tools play an essential role in executing tasks frequently and quickly. Here are some popular tools which are used in this phase, such as **Chef, Puppet, Ansible**, and **SaltStack**.

Containerization tools are also playing an essential role in the deployment phase. **Vagrant** and **Docker** are popular tools that are used for this purpose. These tools help to produce consistency across development, staging, testing, and production environment. They also help in scaling up and scaling down instances softly.

Containerization tools help to maintain consistency across the environments where the application is tested, developed, and deployed. There is no chance of errors or failure in the production environment as they package and replicate the same dependencies and packages used in the testing, development, and staging environment. It makes the application easy to run on different computers.

7) Continuous Operations

All DevOps operations are based on the continuity with complete automation of the release process and allow the organization to accelerate the overall time to market continuingly.

It is clear from the discussion that continuity is the critical factor in the DevOps in removing steps that often distract the development, take it longer to detect issues and produce a better version of the product after several months. With DevOps, we can make any software product more efficient and increase the overall count of interested customers in your product.

Introducing software architecture

We will discuss how DevOps affects the architecture of our applications rather than the architecture of software deployment systems, which we discuss elsewhere in the book.

Often while discussing software architecture, we think of the non-functional requirements of our software. By non-functional requirements, we mean different characteristics of the software rather than the requirements on particular behaviors.

A functional requirement could be that our system should be able to deal with credit card transactions. A non-functional requirement could be that the system should be able to manage several such credit cards transactions per second.

Here are two of the non-functional requirements that DevOps and Continuous Delivery place on software architecture:

- We need to be able to deploy small changes often
- We need to be able to have great confidence in the quality of our changes

The normal case should be that we are able to deploy small changes all the way from developers' machines to production in a small amount of time. Rolling back a change because of unexpected problems caused by it should be a rare occurrence.

So, if we take out the deployment systems from the equation for a while, how will the architecture of the software systems we deploy be affected?

The monolithic scenario

One way to understand the issues that a problematic architecture can cause forContinuous Delivery is to consider a counterexample for a while.

Let's suppose we have a large web application with many different functions.
We also have a static website inside the application. The entire web application is deployed as a single Java EE application archive. So, when we need to fix a spellingmistake in the static website, we need to rebuild the entire web application archive and deploy it again.

While this might be seen as a silly example, and the enlightened reader would neverdo such a thing, I have seen this anti-pattern live in the real world. As DevOps engineers, this could be an actual situation that we might be asked to solve.

Let's break down the consequences of this tangling of concerns. What happens when we want to correct a spelling mistake? Let's take a look:

1. We know which spelling mistake we want to correct, but in which code basedo we need to do it? Since we have a monolith, we need to make a branch inour code base's revision control system. This new branch corresponds to the code that we have running in production.
2. Make the branch and correct the spelling mistake.
3. Build a new artifact with the correction. Give it a new version.
4. Deploy the new artifact to production.
5. We made a change in the monolith that our entire business critical systemcomprises. If something breaks while we are deploying the new version, we lose revenue by the minute. Are we really sure that the change affectsnothing else?
6. It turns out that we didn't really just limit the change to correcting a spellingmistake. We also changed a number of version strings when we produced the new artifact. But changing a version string should be safe too, right? Arewe sure?

The point here is that we have already spent considerable mental energy in making sure that the change is really safe. The system is so complex that it becomes difficultto think about the effects of changes, even though they might be trivial.

Now, a change is usually more complex than a simple spelling correction. Thus, weneed to exercise all aspects of the deployment chain, including manual verification,for all changes to a monolith.

We are now in a place that we would rather not be.

Architecture rules of thumb

There are a number of architecture rules that might help us understand how to dealwith the previous undesirable situation.

The separation of concerns

The renowned Dutch computer scientist Edsger Dijkstra first mentioned his idea of how to organize thought efficiently in his paper from 1974, *On the role ofscientific thought*.

He called this idea "the separation of concerns". To this date, it is arguably the single most important rule in software design. There are many other well-known rules, but many of them follow from the idea of the separation of concerns. The fundamental principle is simply that we should consider different aspects of a system separately.

The principle of cohesion

In computer science, cohesion refers to the degree to which the elements of a software module belong together.

Cohesion can be used as a measure of how strongly related the functions in a module are.

It is desirable to have strong cohesion in a module.

We can see that strong cohesion is another aspect of the principle of the separation of concerns.

Coupling

Coupling refers to the degree of dependency between two modules. We always want low coupling between modules.

Again, we can see coupling as another aspect of the principle of the separation of concerns.

Systems with high cohesion and low coupling would automatically have separation of concerns, and vice versa.

Back to the monolithic scenario

In the previous scenario with the spelling correction, it is clear that we failed with respect to the separation of concerns. We didn't have any modularization at all, at least from a deployment point of view. The system appears to have the undesirable features of low cohesion and high coupling.

If we had a set of separate deployment modules instead, our spelling correction would most likely have affected only a single module. It would have been more apparent that deploying the change was safe.

How this should be accomplished in practice varies, of course. In this particular example, the spelling corrections probably belong to a frontend web component. At the very least, this frontend component can be deployed separately from the backend components and have their own life cycle.

In the real world though, we might not be lucky enough to always be able to influence the different technologies used by the organization where we work. The frontend might, for instance, be implemented using a proprietary content management system with quirks of its own. Where you experience such circumstances, it would be wise to keep track of the cost such a system causes.

A practical example

Let's now take a look at the concrete example we will be working on for the remainder of the book. In our example, we work for an organization called Matangle. This organization is a software as a service (SaaS) provider that sells access to educational games for schoolchildren.

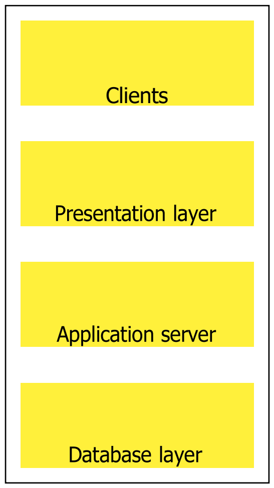
As with any such provider, we will, with all likelihood, have a database containing customer information. It is this database that we will start out with.

The organization's other systems will be fleshed out as we go along, but this initial system serves well for our purposes.

Three-tier systems

The Matangle customer database is a very typical three-tier, **CRUD** (**create, read, update,** and **delete**) type of system. This software architecture pattern has been in use for decades and continues to be popular. These types of systems are very common, and it is quite likely that you will encounter them, either as legacy systems or as new designs.

In this figure, we can see the separation of concerns idea in action:



The three tiers listed as follows show examples of how our organization has chosen to build its system.

The presentation tier

The presentation tier will be a web frontend implemented using the React web framework. It will be deployed as a set of JavaScript and static HTML files. The React framework is fairly recent. Your organization might not use React but perhaps some other framework such as Angular instead. In any case, from a deployment and build point of view, most JavaScript frameworks are similar.

The logic tier

The logic tier is a backend implemented using the Clojure language on the Java platform. The

Java platform is very common in large organizations, while smaller organizations might prefer other platforms based on Ruby or Python. Our example, based on Clojure, contains a little bit of both worlds.

The data tier

In our case, the database is implemented with the PostgreSQL database system. PostgreSQL is a relational database management system. While arguably not as common as MySQL installations, larger enterprises might prefer Oracle databases. PostgreSQL is, in any case, a robust system, and our example organization has chosen PostgreSQL for this reason.

From a DevOps point of view, the three-tier pattern looks compelling, at least superficially. It should be possible to deploy changes to each of the three layers separately, which would make it simple to propagate small changes through the servers.

Handling database migrations

Handling changes in a relational database requires special consideration.

A relational database stores both data and the structure of the data. Upgrading a database schema offers other challenges then the ones present when upgrading program binaries. Usually, when we upgrade an application binary, we stop the application, upgrade it, and then start it again. We don't really bother about the application state. That is handled outside of the application.

When upgrading databases, we do need to consider state, because a database contains comparatively little logic and structure, but contains much state.

In order to describe a database structure change, we issue a command to change the structure.

The database structures before and after a change is applied should be seen as different versions of the database. How do we keep track of database versions?

Generally, database migration systems employ some variant of the following method:

- Add a table to the database where a database version is stored.
- Keep track of database change commands and bunch them together in versioned change sets. In the case of Liquibase, these changes are stored in XML files. Flyway employs a slightly different method where the changesets are handled as separate SQL files or occasionally as separate Java classes for more complex transitions.
- When Liquibase is being asked to upgrade a database, it looks at the metadata table and determines which change sets to run in order to make the database up-to-date with the latest version.

As previously stated, many database version management systems work like this. They differ mostly in the way the changesets are stored and how they determine which change sets to run. They might be stored in an XML file, like in the case of Liquibase, or as a set of separate SQL files, as with Flyway. This later method is more common with homegrown systems and has some advantages. The Clojure ecosystem also has at least one similar database migration system of its own, called Migratus.

Rolling upgrades

Another thing to consider when doing database migrations is what can be referred to as rolling upgrades. These kinds of deployments are common when you don't want your end user to experience any downtime, or at least very little downtime.

Here is an example of a rolling upgrade for our organization's customer database.

When we start, we have a running system with one database and two servers. We have a load balancer in front of the two servers.

We are going to roll out a change to the database schema, which also affects the servers. We are going to split the customer name field in the database into two separate fields, first name and surname.

This is an incompatible change. How do we minimize downtime? Let's look at the solution:

1. We start out by doing a database migration that creates the two new name fields and then fills these new fields by taking the old name field and splitting the field into two halves by finding a space character in the name. This was the initial chosen encoding for names, which wasn't stellar. This is why we want to change it.

   This change is so far backward compatible, because we didn't remove the name field; we just created two new fields that are, so far, unused.

2. Next, we change the load balancer configuration so that the second of our two servers is no longer accessible from the outside world. The first server chugs along happily, because the old name field is still accessible to the old server code.
3. Now we are free to upgrade server two, since nobody uses it.

   After the upgrade, we start it, and it is also happy because it uses the two new database fields.

4. At this point, we can again switch the load balancer configuration such that server one is not available, and server two is brought online instead. We do the same kind of upgrade on server one while it is offline. We start it and now make both servers accessible again by reinstating our original load balancer configuration.

Now, the change is deployed almost completely. The only thing remaining is removing the old name field, since no server code uses it anymore.

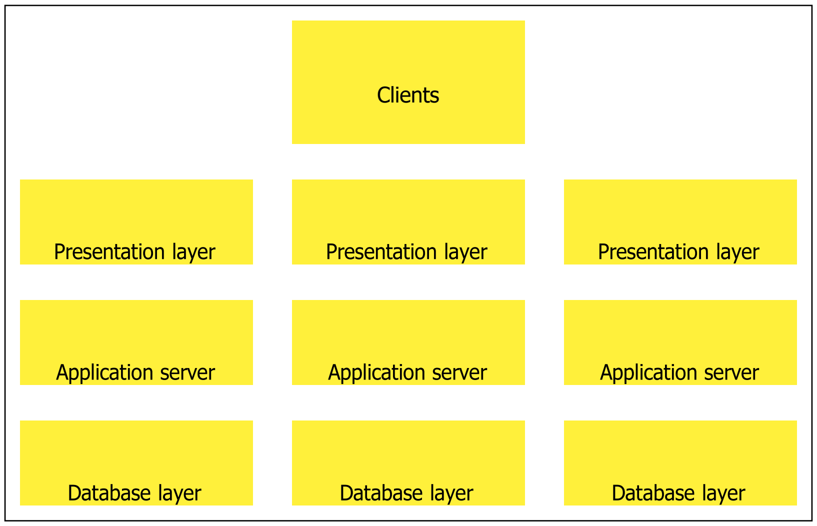
Microservices

Microservices is a recent term used to describe systems where the logic tier of the three-tier pattern is composed of several smaller services that communicate with language-agnostic protocols.

Typically, the language-agnostic protocol is HTTP based, commonly JSON REST, but this is not mandatory. There are several possibilities for the protocol layer.

This architectural design pattern lends itself well to a Continuous Delivery approach since, as we have seen, it's easier to deploy a set of smaller standalone services than a monolith.

Here is an illustration of what a microservices deployment might look like:



We will evolve our example towards the microservices architecture as we go along.

Interlude – Conway's Law
In 1968, Melvin Conway introduced the idea that the structure of an organization that designs software winds up copied in the organization of the software. This is called Conway's Law.

The three-tier pattern, for instance, mirrors the way many organizations' IT departments are structured:

- The database administrator team, or DBA team for short
- The backend developer team
- The frontend developer team
- The operations team

Well, that makes four teams, but we can see the resemblance clearly between the architectural pattern and the organization.

The primary goal of DevOps is to bring different roles together, preferably in cross-functional teams. If Conway's Law holds true, the organization of such teams would be mirrored in their designs.

The microservice pattern happens to mirror a cross-functional team quite closely.

How to keep service interfaces forwardcompatible

Service interfaces must be allowed to evolve over time. This is natural, since the organization's needs change over time, and service interfaces reflect that to a degree.
How can we accomplish this? One way is to use a pattern that is sometimes called **Tolerant Reader**. This simply means that the consumer of a service should ignore data that it doesn't recognize.

This is a method that lends itself well to REST implementations.

Microservices and the data tier
One way of viewing microservices is that each microservice is potentially a separate three-tier system. We don't normally implement each tier for each microservice though. With this in mind, we see that each microservice can implement its own data layer. The benefit would be a potential increase of separation between services.

There are pros and cons to both scenarios. It is easier to deploy changes when the systems are clearly separate from each other. On the other hand, data modeling is easier when everything is stored in the same database.

DevOps, architecture, and resilience
We have seen that the microservice architecture has many desirable properties from a DevOps point of view. An important goal of DevOps is to place new features in the hands of our user faster. This is a consequence of the greater amount of modularization that microservices provide.

Those who fear that microservices could make life uninteresting by offering a perfect solution without drawbacks can take a sigh of relief, though. Microservices do offer challenges of their own.

We want to be able to deploy new code quickly, but we also want our software to be reliable.

Microservices have more integration points between systems and suffer from a higher possibility of failure than monolithic systems.
Automated testing is very important with DevOps so that the changes we deploy are of good quality and can be relied upon. This is, however, not a solution to the problem of services that suddenly stop working for other reasons. Since we have more running services with the microservice pattern, it is statistically more likely for a service to fail.

We can partially mitigate this problem by making an effort to monitor the services and take appropriate action when something fails. This should preferably be automated.

In our customer database example, we can employ the following strategy:

- We use two application servers that both run our application
- The application offers a special monitoring interface via JsonRest
- A monitoring daemon periodically polls this monitoring interface
- If a server stops working, the load balancer is reconfigured such that the offending server is taken out of the server pool

This is obviously a simple example, but it serves to illustrate the challenges we face when designing resilient systems that comprise many moving parts and how they might affect architectural decisions.

Why do we offer our own application-specific monitoring interface though? Since the purpose of monitoring is to give us a thorough understanding of the current health of the system, we normally monitor many aspects of the application stack. We monitor that the server CPU is n't overloaded, that there is sufficient disk and memory space available, and that the base application server is running. This might not be sufficient to determine whether a service is running properly, though. For instance, the service might for some reason have a broken database access configuration. A service-specific health probing interface would attempt to contact the database and return the status of the connection in the return structure.
It is, of course, best if your organization can agree on a common format for the probing return structure. The structure will also depend on the type of monitoring.

UNIT - III
Introduction to project management: The need for source code control, The history of source code management, Roles and code, source code management system and migrations, Shared authentication, Hosted Git servers, Different Git server implementations, Docker intermission, Gerrit, The pull request model, GitLab.

The need for source code control

Terence McKenna, an American author, once said that everything is code.

While one might not agree with McKenna about whether everything in the universe can be represented as code, for DevOps purposes, indeed nearly everything can be expressed in codified form, including the following:

- The applications that we build
- The infrastructure that hosts our applications
- The documentation that documents our products

Even the hardware that runs our applications can be expressed in the form of software.

Given the importance of code, it is only natural that the location that we place code, the source code repository, is central to our organization. Nearly everything we produce travels through the code repository at some point in its life cycle.

The history of source code management

In order to understand the central need for source code control, it can be illuminating to have a brief look at the development history of source code management. This gives us an insight into the features that we ourselves might need. Some examples are as follows:

- Storing historical versions of source in separate archives.

  This is the simplest form, and it still lives on to some degree, with many free software projects offering tar archives of older releases to download.

- Centralized source code management with check in and check out.

  In some systems, a developer can lock files for exclusive use. Every file is managed separately. Tools like this include RCS and SCCS.
- A centralized store where you merge before you commit.
- Examples include Concurrent Versions System (CVS) and Subversion.

  Subversion in particular is still in heavy use. Many organizations have centralized workflows, and Subversion implements such workflows well enough for them.

- A decentralized store.

  On each step of the evolutionary ladder, we are offered more flexibility and concurrency as well as faster and more efficient workflows. We are also offered more advanced and powerful guns to shoot ourselves in the foot with, which we need to keep in mind!

Currently, Git is the most popular tool in this class, but there are many other similar tools in use, such as Bazaar and Mercurial.

Time will tell whether Git and its underlying data model will fend off the contenders to the throne of source code management, who will undoubtedly manifest themselves in the coming years.

Roles and code

From a DevOps point of view, it is important to leverage the natural meeting point that a source code management tool is. Many different roles have a use for source code management in its wider meaning. It is easier to do so for technically-minded roles but harder for other roles, such as project management.

Developers live and breathe source code management. It's their bread and butter.

Operations personnel also favor managing the descriptions of infrastructure in the form of code, scripts, and other artifacts, as we will see in the coming chapters.
Such infrastructural descriptors include network topology, versions of software that should be installed on particular servers, and so on.

Quality assurance personnel can store their automated tests in codified form in the source code repository. This is true for software testing frameworks such as Selenium and Junit, among others.

There is a problem with the documentation of the manual steps needed to perform various tasks, though. This is more of a psychological or cultural problem than
a technical one.

While many organizations employ a wiki solution such as the wiki engine powering Wikipedia, there is still a lot of documentation floating around in the Word format on shared drives and in e-mails.

This makes documentation hard to find and use for some roles and easy for others. From a DevOps viewpoint, this is regrettable, and an effort should be made so that all roles can have good and useful access to the documentation in the organization.

It is possible to store all documentation in the wiki format in the central source code repository, depending on the wiki engine used.

Which source code managementsystem?

There are many source code management (SCM) systems out there, and since SCM is such an important part of development, the development of these systems will continue to happen.

Currently, there is a dominant system, however, and that system is Git.

Git has an interesting story: it was initiated by Linus Torvalds to move Linux kernel development from BitKeeper, which was a proprietary system used at the time. The license of BitKeeper changed, so it wasn't practical to use it for the kernel anymore.

Git therefore supports the fairly complicated workflow of Linux kernel development and is, at the base technical level, good enough for most organizations.

The primary benefit of Git versus older systems is that it is a distributed version control system (DVCS). There are many other distributed version control systems, but Git is the most pervasive one.

Distributed version control systems have several advantages, including, but not limited to,
the following:

- It is possible to use a DVCS efficiently even when not connected to a network. You can take your work with you when traveling on a train or an intercontinental flight.
- Since you don't need to connect to a server for every operation, a DVCS can be faster than the alternatives in most scenarios.
- You can work privately until you feel your work is ready enough to be shared.
- It is possible to work with several remote logins simultaneously, which avoids a single point of failure.

Other distributed version control systems apart from Git include the following:

- Bazaar: This is abbreviated as bzr. Bazaar is endorsed and supported by Canonical, which is the company behind Ubuntu. Launchpad, which is Canonical's code hosting service, supports Bazaar.
- Mercurial: Notable open source projects such as Firefox and OpenJDK use Mercurial. It originated around the same time as Git.

Git can be complex, but it makes up for this by being fast and efficient. It can be hard to understand, but that can be made easier by using frontends that support different tasks.

A word about source code management system migrations

I have worked with many source code management systems and experienced many transitions from one type of system to another.

Sometimes, much time is spent on keeping all the history intact while performing a migration. For some systems, this effort is well spent, such as for venerable free or open source projects.

For many organizations, keeping the history is not worth the significant expenditure in time and effort. If an older version is needed at some point, the old source code management system can be kept online and referenced. This includes migrations from Visual SourceSafe and ClearCase.

Some migrations are trivial though, such as moving from Subversion to Git. In these cases, historic accuracy need not be sacrificed.

Choosing a branching strategy

When working with code that deploys to servers, it is important to agree on a branching strategy across the organization.

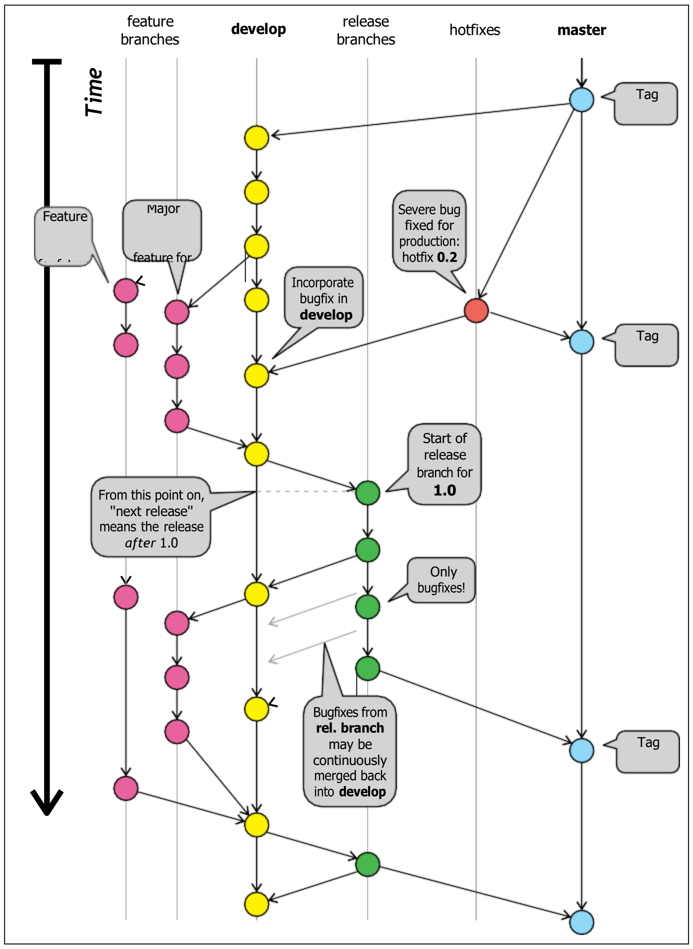
A branching strategy is a convention, or a set of rules, that describes when branches are created, how they are to be named, what use branches should have, and so on.

Branching strategies are important when working together with other people and are, to a degree, less important when you are working on your own, but they still have a purpose.

Most source code management systems do not prescribe a particular branching strategy and neither does Git. The SCM simply gives you the base mechanics to perform branching.

With Git and other distributed version control systems, it is usually pretty cheap to work locally with feature branches. A feature, or topic, branch is a branch that is used to keep track of ongoing development regarding a particular feature, bug, and so on. This way, all changes in the code regarding the feature can be handled together.

There are many well-known branching strategies. Vincent Driessen formalized a branching strategy called Git flow, which has many good features. For some, Git flow is too complex, and in those cases, it can be scaled down. There are many such scaled-down models available. This is what Git flow looks like:


feature branches
develop
release branches
hotfixes
master
Time
Tag
Feature
Major
feature for
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag

Git flow looks complex, so let's have a brief look at what the branches are for:

- The master branch only contains finished work. All commits are tagged, since they represent releases. All releases happen from master.
- The develop branch is where work on the next release happens. When work is finished

here, develop is merged to master.

- We use separate feature branches for all new features. Feature branches are merged to develop.
- When a devastating bug is revealed in production, a hotfix branch is made where a bug fix is created. The hotfix branch is then merged to master, and a new release for production is made.

Git flow is a centralized pattern, and as such, it's reminiscent of the workflows used with Subversion, CVS, and so on. The main difference is that using Git has some technical and efficiency-related advantages.

Another strategy, called the forking pattern, where every developer has a central repository, is rarely used in practice within organizations, except when, for instance, external parties such as subcontractors are being employed.

Branching problem areas

There is a source of contention between Continuous Delivery practices and branching strategies. Some Continuous Delivery methods advocate a single master branch and all releases being made from the master branch. Git flow is one such model.

This simplifies some aspects of deployment, primarily because the branching graph becomes simpler. This in turn leads to simplified testing, because there is only one branch that leads to the production servers.

On the other hand, what if we need to perform a bug fix on released code, and the master has new features we don't want to release yet? This happens when the installation cadence in production is slower than the release cadence of the development teams. It is an undesirable state of affairs, but not uncommon.

There are two basic methods of handling this issue:

- Make a bug fix branch and deploy to production from it: This is simpler in the sense that we don't disrupt the development flow. On the other hand,
  this method might require duplication of testing resources. They might need to mirror the branching strategy.
- Feature toggling: Another method that places harder requirements on the developers is feature toggling. In this workflow, you turn off features that are not yet ready for production. This way you can release the latest development version, including the bug fix, together with new features, which will be disabled and inactive.

There are no hard rules, and dogmatism is not helpful in this case.

It is better to prepare for both scenarios and use the method that serves you best in the given circumstances.

Here are some observations to guide you in your choice:

Feature toggling is good when you have changes that are mostly backward compatible or are entirely new. Otherwise, the feature toggling code might become overly complex, catering to many different cases.

This, in turn, complicates testing.

Bug fix branches complicate deployment and testing. While it is straightforward to branch and make the bug fix, what should the new version be called, and where do we test it?

Shared authentication

In most organizations, there is some form of a central server for handling authentication. An LDAP (Lightweight Directory Access Protocol) server is a fairly common choice. While it is assumed that your organization has already dealt with this core issue one way or the other and is already running an LDAP server of some sort, it is comparatively easy to set up an LDAP server for testing purposes.

One possibility is using 389 Server, named after the port commonly used for the LDAP server, together with the php LDAP admin web application for administration of the LDAP server.

Having this test LDAP server setup is useful, since we can then use the same LDAP server for all the different servers we are going to investigate.

Hosted Git servers

Many organizations can't use services hosted within another organization's walls at all.

These might be government organizations or organizations dealing with money, such as bank, insurance, and gaming organizations.

The causes might be legal or, simply nervousness about letting critical code leave the organization's doors, so to speak.

If you have no such qualms, it is quite reasonable to use a hosted service, such as GitHub or GitLab, that offers private accounts.

Using GitHub or GitLab is, at any rate, a convenient way to get to learn to use Git and explore its possibilities.

Both vendors are easy to evaluate, given that they offer free accounts where you can get to know the services and what they offer. See if you really need all the services or if you can make do with something simpler.

Some of the features offered by both GitLab and GitHub over plain Git are as follows:

- Web interfaces
- A documentation facility with an inbuilt wiki
- Issue trackers
- Commit visualization

- Branch visualization
- The pull request workflow

While these are all useful features, it's not always the case that you can use the facilities provided. For instance, you might already have a wiki, a documentation system, an issue tracker, and so on that you need to integrate with.

The most important features we are looking for, then, are those most closely related to managing and visualizing code.

Large binary files

GitHub and GitLab are pretty similar but do have some differences. One of them springs from the fact that source code systems like Git traditionally didn't cater much for the storage of large binary files. There have always been other ways, such as storing file paths to a file server in plain text files.

But what if you actually have files that are, in a sense, equivalent to source files except that they are binary, and you still want to version them? Such file types might include image files, video files, audio files, and so on. Modern websites make increasing use of media files, and this area has typically been the domain of content management systems (CMSes). CMSes, however nice they might be, have disadvantages compared to DevOps flows, so the allure of storing media files in your ordinary source handling system is strong. Disadvantages of CMSes include the fact that they often have quirky or nonexistent scripting capabilities. Automation, another word that should have really been included in the "DevOps" portmanteau, is therefore often difficult with a CMS.

You can, of course, just check in your binary to Git, and it will be handled like any other file. What happens then is that Git operations involving server operations suddenly become sluggish. And then, some of the main advantages of Git— efficiency and speed—vanish out the window.

Solutions to this problem have evolved over time, but there are no clear winners yet. The contenders are as follows:

- Git LFS, supported by GitHub
- Git Annex, supported by GitLab but only in the enterprise edition

Git Annex is the more mature of these. Both solutions are open source and are implemented as extensions to Git via its plugin mechanism.

There are several other systems, which indicates that this is an unresolved pain point in the current state of Git. The Git Annex has a comparison between the different breeds at http://git-annex.branchable.com/not/.

Trying out different Git server implementations

The distributed nature of Git makes it possible to try out different Git implementations for various purposes. The client-side setup will be similar regardless of how the server is set up.

You can also have several solutions running in parallel. The client side is not unduly complicated by this, since Git is designed to handle several remotes if need be.

Docker intermission

Deploying the Code, we will have a look at a new and exciting way to package our applications, called Docker.

we have a similar challenge to solve. We need to be able to try out a couple of different Git server implementations to see which one suits our organization best.

We can use Docker for this, so we will take this opportunity to peek at the possibilities of simplified deployments that Docker offers us.

Since we are going to delve deeper into Docker further along, that Docker is used to download and run software. While this description isn't entirely untrue, Docker is much more than that. To get started with Docker, follow these steps:

To begin with, install Docker according to the particular instructions for your operating system. For Red Hat derivatives, it's a simple dnf install docker-io command.

Then, the docker service needs to be running:

systemctl enable docker systemctl start docker

We need another tool, Docker Compose, which, at the time of writing, isn't packaged for Fedora. If you don't have it available in your package repositories, follow the instructions on this page

https://docs.docker. com/compose/install/

Gerrit

A basic Git server is good enough for many purposes. Sometimes, you need precise control over the workflow, though.

One concrete example is merging changes into configuration code for critical parts of the infrastructure. While my opinion is that it's core to DevOps to not place unnecessary red tape around infrastructure code, there is no denying that it's sometimes necessary. If nothing else, developers might feel nervous committing changes to the infrastructure and would like for someone more experienced to review the code changes.

Gerrit is a Git-based code review tool that can offer a solution in these situations. In brief, Gerrit allows you to set up rules to allow developers to review and approve changes made to a codebase by other developers. These might be senior developers reviewing changes made by inexperienced developers or the more common case, which is simply that more eyeballs on the code is good for quality in general.

Gerrit is Java-based and uses a Java-based Git implementation under the hood.

Gerrit can be downloaded as a Java WAR file and has an integrated setup method. It needs a relational database as a dependency, but you can opt to use an integrated Java-based H2 database that is good enough for evaluating Gerrit.

An even simpler method is using Docker to try out Gerrit. There are several Gerrit images on the Docker registry hub to choose from. The following one was selected for this evaluation exercise: https://hub.docker.com/r/openfrontier/gerrit/

To run a Gerrit instance with Docker, follow these steps:

1. Initialize and start Gerrit:

   ```
   docker run -d -p 8080:8080 -p 29418:29418 openfrontier/gerrit
   ```

2. Open your browser to http://<docker host url>:8080

   Now, we can try out the code review feature we would like to have.

Installing the git-review package

Install git-review on your local installation:

```
sudo dnf install git-review
```

This will install a helper application for Git to communicate with Gerrit. It adds a new command, git-review, that is used instead of git push to push changes to the Gerrit Git server.

The value of history revisionism

When we work with code together with other people in a team, the code's history becomes more important than when we work on our own. The history of changes to files becomes a way to communicate. This is especially important when working with code review and code review tools such as Gerrit .

The code changes also need to be easy to understand. Therefore, it is useful, although perhaps counterintuitive, to edit the history of the changes in order to make the resulting history clearer.

As an example, consider a case where you made a number of changes and later changed your mind and removed them. It is not useful information for someone else that you made a set of edits and then removed them. Another case is when you have a set of commits that are easier to understand if they are a single commit. Adding commits together in this way is called squashing in the Git documentation.

Another case that complicates history is when you merge from the upstream central repository several times, and merge commits are added to the history. In this case, we want to simplify the changes by first removing our local changes, then fetching and applying changes from the upstream repository, and then finally reapplying our local changes. This process is called rebasing.

Both squashing and rebasing apply to Gerrit.

The changes should be clean, preferably one commit. This isn't something that is particular to Gerrit; it's easier for a reviewer to understand your changes if they are packaged nicely. The review will be based on this commit.

1. To begin with, we need to have the latest changes from the Git/Gerrit server side. We rebase

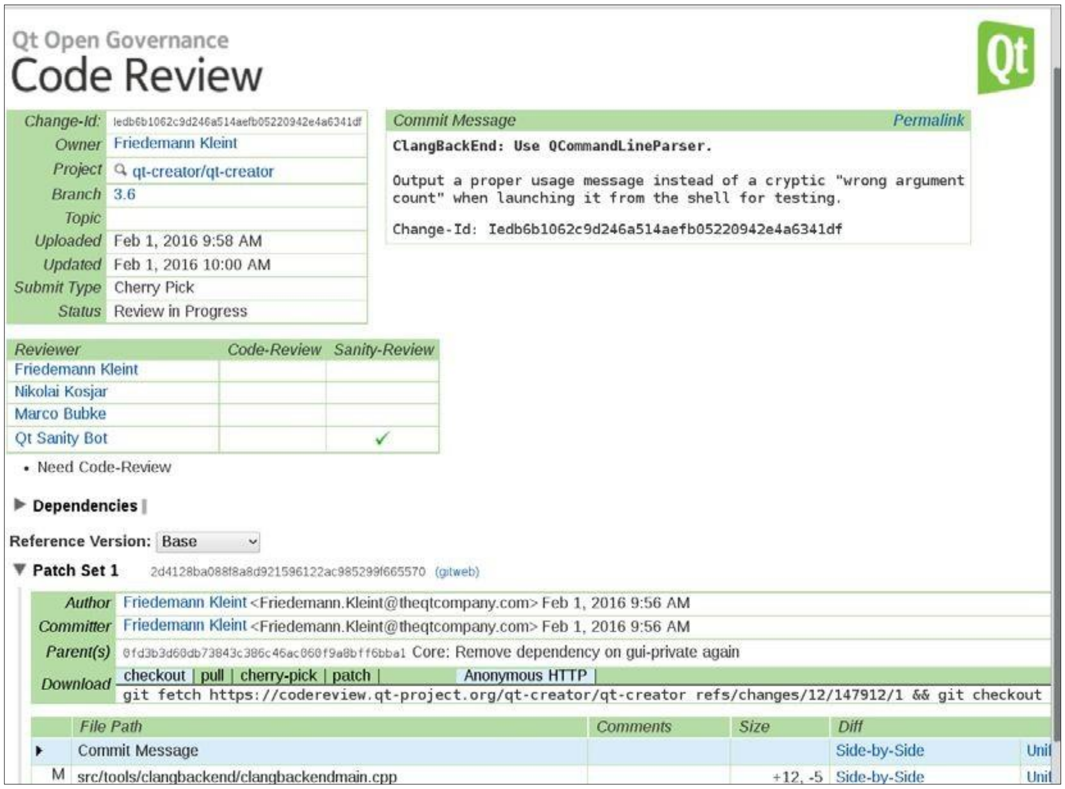
our changes on top of the server-side changes:

```
git pull --rebase origin master
```

2. Then, we polish our local commits by squashing them:

   ```
   git rebase -i origin/master
   ```

Now, let's have a look at the Gerrit web interface:



Qt Open Governance
Code Review

Change-Id:	Iedb6b1062c9d246a514aefb05220942e4a6341df
Owner	Friedemann Kleint
Project	qt-creator/qt-creator
Branch	3.6
Topic	
Uploaded	Feb 1, 2016 9:58 AM
Updated	Feb 1, 2016 10:00 AM
Submit Type	Cherry Pick
Status	Review in Progress

Commit Message — Permalink

```
ClangBackEnd: Use QCommandLineParser.

Output a proper usage message instead of a cryptic "wrong argument
count" when launching it from the shell for testing.

Change-Id: Iedb6b1062c9d246a514aefb05220942e4a6341df
```

Reviewer	Code-Review	Sanity-Review
Friedemann Kleint		
Nikolai Kosjar		
Marco Bubke		
Qt Sanity Bot		✓

- Need Code-Review

▶ Dependencies

Reference Version: Base

▼ Patch Set 1 2d4128ba088f8a8d921596122ac985299f665570 (gitweb)

Author	Friedemann Kleint <Friedemann.Kleint@theqtcompany.com> Feb 1, 2016 9:56 AM
Committer	Friedemann Kleint <Friedemann.Kleint@theqtcompany.com> Feb 1, 2016 9:56 AM
Parent(s)	0fd3b3d60db73843c386c46ac060f9a8bff6bba1 Core: Remove dependency on gui-private again
Download	checkout | pull | cherry-pick | patch | Anonymous HTTP |

```
git fetch https://codereview.qt-project.org/qt-creator/qt-creator refs/changes/12/147912/1 && git checkout
```

	File Path	Comments	Size	Diff	
▸	Commit Message			Side-by-Side	Unit
M	src/tools/clangbackend/clangbackendmain.cpp		+12, -5	Side-by-Side	Unit

We can now approve the change, and it is merged to the master branch.

There is much to explore in Gerrit, but these are the basic principles and should be enough to base an evaluation on.

Are the results worth the hassle, though? These are my observations:

Gerrit allows fine-grained access to sensitive

- codebases. Changes can go in after being reviewed by authorized personnel.

  This is the primary benefit of Gerrit. If you just want to have mandatory code reviews for unclear reasons, don't. The benefit has to be clear for everyone involved. It's better to agree on other more informal methods of code review than an authoritative system.

- If the alternative to Gerrit is to not allow access to code bases at all, even read-only access, then implement Gerrit.

- Some parts of an organization might be too nervous to allow access to things such as infrastructure configuration. This is usually for the wrong reasons.

- The problem you usually face isn't people taking an interest in your code; it's the opposite.
- Sometimes, sensitive passwords are checked in to code, and this is taken as a reason to disallow access to the source. Well, if it hurts, don't do it. Solve the problem that leads to there being passwords in the repositories instead.

The pull request model

There is another solution to the problem of creating workflows around code reviews: the pull request model, which has been made popular by GitHub.

In this model, pushing to repositories can be disallowed except for the repository owners. Other developers are allowed to fork the repository, though, and make changes in their fork. When they are done making changes, they can submit a pull request. The repository owners can then review the request and opt to pull the changes into the master repository.

This model has the advantage of being easy to understand, and many developers have experience in it from the many open source projects on GitHub.

Setting up a system capable of handling a pull request model locally will require something like GitHub or GitLab.

GitLab

GitLab supports many convenient features on top of Git. It's a large and complex software system based on Ruby. As such, it can be difficult to install, what with getting all the dependencies right and so on.

There is a nice Docker Compose file for GitLab available at https://registry.hub.docker.com/u/sameersbn/gitlab/. If you followed the instructions for Docker shown previously, including the installation of docker-compose, it's now pretty simple to start a local GitLab instance:

```
mkdir gitlab cd gitlab

wget https://raw.githubusercontent.com/sameersbn/docker-gitlab/master/ docker-compose.yml

docker-compose up
```

The docker-compose command will read the .yml file and start all the required services in a default demonstration configuration.

If you read the startup log in the console window, you will notice that three separate application containers have been started: gitlab postgresql1, gitlab redis1, and gitlab gitlab1.

The GitLab container includes the Ruby base web application and Git backend functionality. Redis is distributed key-value store, and PostgreSQL is a relational database.

If you are used to setting up complicated server functionality, you will appreciate that we have saved a great deal of time with docker-compose.

The docker-compose.yml file sets up data volumes at /srv/docker/gitlab.

To log in to the web user interface, use the administrator password given with the installation instructions for the GitLab Docker image. They have been replicated here, but beware that they might change as the Docker image author sees fit:

- Username: root
- Password: DevopsLife

Here is a screenshot of the GitLab web user interface login screen:

Try importing a project to your GitLab server from, for instance, GitHub or a local private project.

Have a look at how GitLab visualizes the commit history and branches.

While investigating GitLab, you will perhaps come to agree that it offers a great deal of interesting functionality.

When evaluating features, it's important to keep in mind whether it's likely that they will be used after all. What core problem would GitLab, or similar software, solve for you?

It turns out that the primary value added by GitLab, as exemplified by the following two features, is the elimination of bottlenecks in DevOps workflows:

- The management of user ssh keys
- The creation of new repositories

These features are usually deemed to be the most useful.

Visualization features are also useful, but the client-side visualization available with Git clients is more useful to developers.

UNIT-IV

Integrating the system: Build systems, Jenkins build server, Managing build dependencies, Jenkins plugins, and file system layout, The host server, Build slaves, Software on the host, Triggers, Job chaining and build pipelines, Build servers and infrastructure as code, Building by dependency order, Build phases, Alternative build servers, Collating quality measures.

Build systems

You need a system to build your code, and you need somewhere to build it. Jenkins is a flexible open source build server that grows with your needs. Some alternatives to Jenkins will be explored as well. We will also explore the different build systems and how they affect our DevOps work.

Why do we build code?

Most developers are familiar with the process of building code. When we work in the field of DevOps, however, we might face issues that developers who specialize in programming a particular component type won't necessarily experience. we define software building as the process of molding code from one form to another. During this process, several things might happen:

- The compilation of source code to native code or virtual machine bytecode, depending on our production platform.
- Linting of the code: checking the code for errors and generating code quality measures by means of static code analysis. The term "Linting" originated with a program called Lint, which started shipping with early versions of the Unix operating system. The purpose of the program was to find bugs in programs that were syntactically correct, but contained suspicious code patterns that could be identified with a different process than compiling.
- Unit testing, by running the code in a controlled manner.
- The generation of artifacts suitable for deployment. It's a tall order!

Not all code goes through each and every one of these phases. Interpreted languages,for example, might not need compilation, but they might benefit from quality checks.

The many faces of build systems

There are many build systems that have evolved over the history of software development. Sometimes, it might feel as if there are more build systems than there are programming languages.

Here is a brief list, just to get a feeling for how many there are:

For Java, there is Maven, Gradle, and Ant

For C and C++, there is Make in many different flavors

For Clojure, a language on the JVM, there is Leiningen and Boot apart from Maven

For JavaScript, there is Grunt

For Scala, there is sbt

For Ruby, we have Rake

Finally, of course, we have shell scripts of all kinds

Depending on the size of your organization and the type of product you are building, you might encounter any number of these tools. To make life even more interesting, it's not uncommon for organizations to invent their own build tools.

As a reaction to the complexity of the many build tools, there is also often the idea of standardizing a particular tool. If you are building complex heterogeneous systems, this is rarely efficient. For example, building JavaScript software is just easier with Grunt than it is with Maven or Make, building C code is not very efficient with Maven, and so on. Often, the tool exists for a reason.

Normally, organizations standardize on a single ecosystem, such as Java and Maven or Ruby and Rake. Other build systems besides those that are used for the primary code base are encountered mainly for native components and third-party components.

At any rate, we cannot assume that we will encounter only one build system within our organization's code base, nor can we assume only one programming language.

I have found this rule useful in practice: it should be possible for a developer to check out the code and build it with minimal surprises on his or her local developer machine.

This implies that we should standardize the revision control system and have a single interface to start builds locally.

If you have more than one build system to support, this basically means that you need to wrap one build system in another. The complexities of the build are thus hidden and more than one build system at the same time are allowed. Developers not familiar with a particular build can still expect to check it out and build it with reasonable ease.

Maven, for example, is good for declarative Java builds. Maven is also capable of starting other builds from within Maven builds.

This way, the developer in a Java-centric organization can expect the following command line to always build one of the organization's components:

```
mvn clean install
```

One concrete example is creating a Java desktop application installer with the Nullsoft NSIS Windows installation system. The Java components are built with Maven. When the Java artifacts are ready, Maven calls the NSIS installer script to produce a self-contained executable that will install the application on Windows.

While Java desktop applications are not fashionable these days, they continue to be popular in some domains.

What is Jenkins?

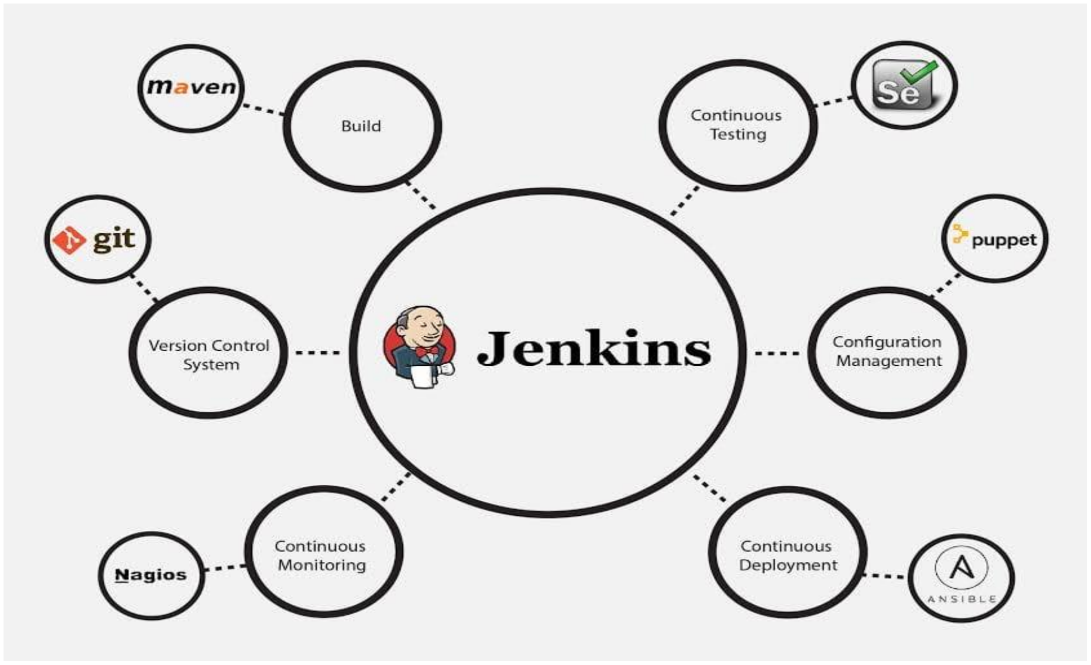
Continuous Integration is the most important part of DevOps that is used to integrate various **DevOps stages**. Jenkins is the most famous Continuous Integration tool, I know you are curious to know the reason behind the popularity of Jenkins, and if Jenkins is easy to learn. I am pretty sure after reading this ***What is Jenkins*** blog, all your questions will get answered.

What is Jenkins and why we use it?
Jenkins is an open-source automation tool written in Java with plugins built for continuous integration. Jenkins is used to build and test your software projects continuously making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build. It also allows you to continuously deliver your software by integrating with a large number of testing and deployment technologies.

With Jenkins, organizations can accelerate the software development process through automation. Jenkins integrates development life-cycle processes of all kinds, including build, document, test, package, stage, deploy, static analysis, and much more.

Jenkins achieves Continuous Integration with the help of plugins. Plugins allow the integration of Various DevOps stages. If you want to integrate a particular tool, you need to install the plugins for that tool. For example Git, Maven 2 project, Amazon EC2, HTML publisher etc.

The image below depicts that Jenkins is integrating various DevOps stages:



Advantages of Jenkins include:

- It is an open-source tool with great community support.
- It is easy to install.

- It has 1000+ plugins to ease your work. If a plugin does not exist, you can code it and share it with the community.
- It is free of cost.
- It is built with Java and hence, it is portable to all the major platforms.

There are certain things about Jenkins that separates it from other the Continuous Integration tool. Let us take a look on those points.

Jenkins Features
The following are some facts about Jenkins that makes it better than other Continuous Integration tools:

- **Adoption**: Jenkins is widespread, with more than 147,000 active installations and over 1 million users around the world.
- **Plugins**: Jenkins is interconnected with well over 1,000 plugins that allow it to integrate with most of the development, testing and deployment tools.

It is evident from the above points that Jenkins has a very high demand globally. Before we dive into Jenkins it is important to know what is Continuous Integration and why it was introduced.

What is Continuous Integration?
Continuous Integration is a development practice in which the developers are required to commit changes to the source code in a shared repository several times a day or more frequently. Every commit made in the repository is then built. This allows the teams to detect the problems early. Apart from this, depending on the Continuous Integration tool, there are several other functions like deploying the build application on the test server, providing the concerned teams with the build and test results, etc.

Let us understand its importance with a use-case.

Continuous Integration Example: Nokia
I am pretty sure you all have used **Nokia** phones at some point in your life. In a software product development project at Nokia, there was a process called **Nightly builds**. Nightly builds can be thought of as a predecessor to Continuous Integration. It means that every night an automated system pulls the code added to the shared repository throughout the day and builds that code. The idea is quite similar to Continuous Integration, but since the code that was built at night was quite large, locating and fixing of bugs was a real pain. Due to this, Nokia adopted Continuous Integration (CI). As a result, every commit made to the source code in the repository was built. If the build result shows that there is a bug in the code, then the developers only need to check that particular commit. This significantly reduced the time required to release new software.

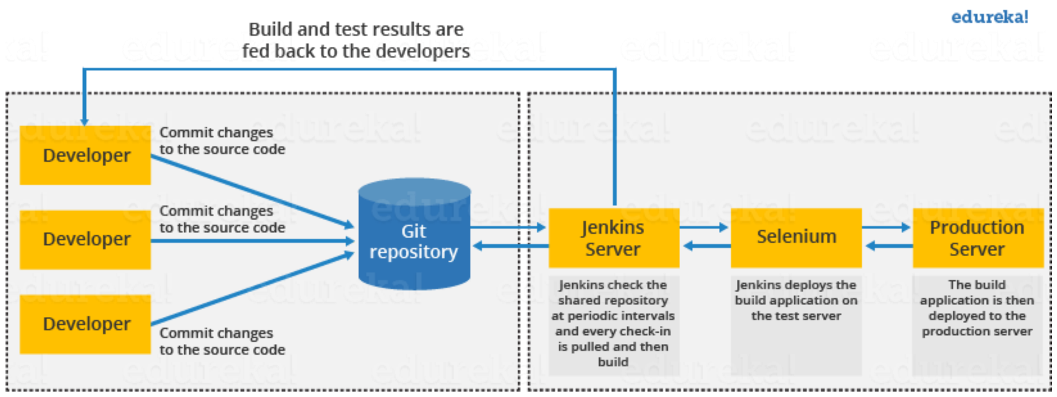
Now is the correct time to understand how Jenkins achieves Continuous Integration.

Continuous Integration With Jenkins
Let us imagine a scenario where the complete source code of the application was built and then deployed on test server for testing. It sounds like a perfect way to develop software, but, this process has many flaws. I will try to explain them one by one:

- Developers have to wait until the complete software is developed for the test results.
- There is a high possibility that the test results might show multiple bugs. It was tough for developers to locate those bugs because they have to check the entire source code of the application.
- It slows the software delivery process.
- Continuous feedback pertaining to things like coding or architectural issues, build failures, test status and file release uploads was missing due to which the quality of software can go down.
- The whole process was manual which increases the risk of frequent failure.

It is evident from the above-stated problems that not only the software delivery process became slow but the quality of software also went down. This leads to customer dissatisfaction. So to overcome such chaos there was a dire need for a system to exist where developers can continuously trigger a build and test for every change made in the source code. This is what CI is all about. Jenkins is the most mature CI tool available so let us see how Continuous Integration with Jenkins overcame the above shortcomings.

I will first explain to you a generic flow diagram of Continuous Integration with Jenkins so that it becomes self-explanatory, how Jenkins overcomes the above shortcomings. This will help you in understanding how does Jenkins work.


edureka!
Build and test results are fed back to the developers
Developer
Commit changes to the source code
Developer
Commit changes to the source code
Developer
Commit changes to the source code
Git repository
Jenkins Server
Jenkins check the shared repository at periodic intervals and every check-in is pulled and then build
Selenium
Jenkins deploys the build application on the test server
Production Server
The build application is then deployed to the production server

The above diagram is depicting the following functions:

- First, a developer commits the code to the source code repository. Meanwhile, the Jenkins server checks the repository at regular intervals for changes.
- Soon after a commit occurs, the Jenkins server detects the changes that have occurred in the source code repository. Jenkins will pull those changes and will start preparing a new build.
- If the build fails, then the concerned team will be notified.
- If built is successful, then Jenkins deploys the built in the test server.
- After testing, Jenkins generates a feedback and then notifies the developers about the build and test results.
- It will continue to check the source code repository for changes made in the source code and the whole process keeps on repeating.

You now know how Jenkins overcomes the traditional SDLC shortcomings. The table below shows the comparison between "Before and After Jenkins".

Before and After Jenkins

Before Jenkins	After Jenkins
The entire source code was built and then tested. Locating and fixing bugs in the event of build and test failure was difficult and time-consuming, which in turn slows the software delivery process.	Every commit made in the source code is built and tested. So, instead of checking the entire source code developers only need to focus on a particular commit. This leads to frequent new software releases.
Developers have to wait for test results	Developers know the test result of every commit made in the source code on the run.
The whole process is manual	You only need to commit changes to the source code and Jenkins will automate the rest of the process for you.

The Jenkins build server

What is Jenkins build server?

Jenkins is an open source automation server. It helps automate the parts of software development related to building, testing, and deploying, facilitating continuous integration and continuous delivery. It is a server-based system that runs in servlet containers such as Apache Tomcat.

What is Jenkins server used for?

Jenkins is an open source continuous integration/continuous delivery and deployment (CI/CD) automation software DevOps tool written in the Java programming language. It is used to implement CI/CD workflows, called pipelines.

A build server is, in essence, a system that builds software based on various triggers. There are several to choose from. In this book, we will have a look at Jenkins, which is a popular build server written in Java.

Jenkins is a fork of the Hudson build server. Kohsuke Kawaguchi was Hudson's principal contributor, and in 2010, after Oracle acquired Hudson, he continued work on the Jenkins fork. Jenkins is clearly the more successful of the two strains today.

Jenkins has special support for building Java code but is in no way limited to just building Java.

Setting up a basic Jenkins server is not particularly hard at the outset. In Fedora, you can just install it via dnf:

```
dnf install jenkins
```

Jenkins is handled as a service via systemd:

```
systemctl start jenkins
```

You can now have a look at the web interface at http://localhost:8080:

The Jenkins instance in the screenshot has a couple of jobs already defined. The fundamental entity in Jenkins is the job definition, and there are several types to choose from. Let's create a simple job in the web interface. To keep it simple, this job will just print a classic Unix fortune quote:

1. Create a job of the type Freestyle project:
2. Add a shell build step.
3. In the shell entry (Command), type fortune:

Whenever you run the job, a fortune quote will be printed in the job log.

Jobs can be started manually, and you will find a history of job executions and can examine each job log. This keeps a history of previous executions, which is very handy when you are trying to figure out which change broke a build and how
to fix it.

If you don't have the fortune command, install it with dnf install fortune-mod, or you might opt to simply run the date command instead. This will just output the date in the build log instead of classic quotes and witticisms.

Managing build dependencies

In the previous, simple example, we printed a fortune cookie to the build log.

While this exercise can't be compared in complexity with managing real builds, we at least learned to install and start Jenkins, and if you had issues installing the fortune utility, you got a glimpse of the dark underbelly of managing a Continuous Integration server: managing the build dependencies.

Some build systems, such as the Maven tool, are nice in the way that the Maven POM file contains descriptions of which build dependencies are needed, and they are fetched automatically by Maven if they aren't already present on the build server.
Grunt works in a similar way for JavaScript builds. There is a build description file that contains the dependencies required for the build. Golang builds can even contain links to GitHub repositories required for completing the build.

C and C++ builds present challenges in a different way. Many projects use GNU Autotools; among them is Autoconf, which adapts itself to the dependencies that are available on the host rather than describing which dependencies they need. So, to build Emacs, a text editor, you first run a configuration script that determines which of the many potential dependencies are available on the build system. While this is a useful feature if you want your software to work in many different configurations depending on which system it should run on, it's not often the way we would like our builds to behave in an enterprise setting. In this case, we need to be perfectly sure which features will be available in the end. We certainly don't want bad surprises in the form of missing functionality on our production servers.

The RPM (short for Red Hat Package Manager) system, which is used on systems derived from Red Hat, offers a solution to this problem. At the core of the RPM system is a build descriptor file called a spec file, short for specification file. It lists, among other things, the build dependencies required for a successful build and the build commands and configuration options used. Since a spec file is essentially a macro-based shell script, you can use it to build many types of software. The RPM system also has the idea that build sources should be pristine. The spec file can adapt the source code by patching the source before building it.

The final artifact

After finishing the build using the RPM system, you get an RPM file, which is a very convenient type of deployment artifact for operating systems based on Red Hat. For Debian-based distributions, you get a .deb file.

The final output from a Maven build is usually an enterprise archive, or EAR file for short. This contains Java Enterprise applications.
It is final deployment artifacts such as these that we will later deploy to our production servers.

We concern ourselves with building the artifacts required for deployment, and in Deploying the Code, we talk about the final deployment of our artifacts.

However, even when building our artifacts, we need to understand how to deploy them. At the moment, we will use the following rule of thumb: OS-level packaging is preferable to specialized packaging. This is my personal preference, and others might disagree.

Let's briefly discuss the background for this rule of thumb as well as the alternatives.

As a concrete example, let's consider the deployment of a Java EAR. Normally, we can do this in several ways. Here are some examples:

- Deploy the EAR file as an RPM package through the mechanisms and channels available in the base operating system
- Deploy the EAR through the mechanisms available with the Java application server, such as JBoss, WildFly, and Glassfish
- It might superficially look like it would be better to use the mechanism specific to the Java application server to deploy the EAR file, since it is specific to the application server anyway. If Java development is all you ever do, this might be a reasonable supposition. However, since you need to manage your base operating system anyway, you already have methods of deployment available to you that are possible to reuse.
- Also, since it is quite likely that you are not just doing Java development but also need to deploy and manage HTML and JavaScript at the very least, it starts to make sense to use a more versatile method of deployment.
- Nearly all the organizations had complicated architectures comprising many different technologies, and this rule of thumb has served well in most scenarios.
- The only real exception is in mixed environments where Unix servers coexist with Windows

servers. In these cases, the Unix servers usually get to use their preferred package distribution method, and the Windows servers have to limp along with some kind of home-brewed solution. This is just an observation and not a condoning of the situation.

Cheating with FPM

Building operating system deliverables such as RPMs with a spec file is very useful knowledge. However, sometimes you don't need the rigor of a real spec file. The spec file is, after all, optimized for the scenario where you are not yourself the originator of the code base.

There is a Ruby-based tool called FPM, which can generate source RPMs suitable for building, directly from the command line.

The tool is available on GitHub at https://github.com/jordansissel/fpm. On Fedora, you can

install FPM like this:

```
yum install rubygems yum install ruby

yum install ruby-devel gcc gem install fpm
```

This will install a shell script that wraps the FPM Ruby program.

One of the interesting aspects of FPM is that it can generate different types of package; among the supported types are RPM and Debian.

Here is a simple example to make a "hello world" shell script:

```
#!/bin/sh

echo 'Hello World!'
```

We would like the shell script to be installed in /usr/local/bin, so create a directory in your home directory with the following structure:

```
$HOME/hello/usr/local/bin/hello.sh
```

Make the script executable, and then package it:

```
chmod a+x usr/local/bin/hello.sh

fpm -s dir -t rpm -n hello-world -v 1 -C installdir usr
```

This will result in an RPM with the name hello-world and version 1.

To test the package, we can first list the contents and then install it:

```
rpm  -qivp  hello-world.rpm  rpm  -ivh  hello-
world.rpm
```

The shell script should now be nicely installed in /usr/local/bin.

FPM is a very convenient method for creating RPM, Debian, and other package types. It's a little bit like cheating!

Continuous Integration

The principal benefit of using a build server is achieving Continuous Integration. Each time a change in the code base is detected, a build that tests the quality of the newly submitted code is started.

Since there might be many developers working on the code base, each with slightly different versions, it's important to see whether all the different changes work together properly. This is called integration testing. If integration tests are too far apart, there is a growing risk of the different code branches diverging too much, and merging is no longer easy. The result is often referred to as "merge hell". It's no longer clear how a developer's local changes should be merged to the master branch, because of divergence between the branches. This situation is very undesirable. The root cause of merge hell is often, perhaps surprisingly, psychological. There is a mental barrier to overcome in order to merge your changes to the mainline. Part of working with DevOps is making things easier and thus reducing the perceived costs associated with doing important work like submitting changes.

Continuous Integration builds are usually performed in a more stringent manner than what developers do locally. These builds take a longer time to perform, but since perform Ant hardware is not so expensive these days, our build server is beefy enough to cope with these builds.

If the builds are fast enough to not be seen as tedious, developers will be enthused to check in often, and integration problems will be found early.

Continuous Delivery

After the Continuous Integration steps have completed successfully, you have shiny new artifacts that are ready to be deployed to servers. Usually, these are test environments set up to behave like production servers.

Often, the last thing a build server does is to deploy the final artifacts from the successful build to an artifact repository. From there, the deployment servers take over the responsibility of deploying them to the application servers. In the Java world, the Nexus repository manager is fairly common. It has support for other formats besides the Java formats, such as JavaScript artifacts and

Yum channels for RPMs. Nexus also supports the Docker Registry API now.

Using Nexus for RPM distributions is just one option. You can build Yum channels with a shell script fairly easily.

Jenkins plugins

Jenkins has a plugin system to add functionality to the build server. There are many different plugins available, and they can be installed from within the Jenkins web interface. Many of them can be installed without even restarting Jenkins. This screenshot shows a list of some of the available plugins:

Among others, we need the Git plugin to poll our source code repositories.

The host server

The build server is usually a pretty important machine for the organization. Building software is processor as well as memory and disk intensive. Builds shouldn't take too long, so you will need a server with good specifications for the build server—with lots of disk space, processor cores, and RAM.

The build server also has a kind of social aspect: it is here that the code of many different people and roles integrates properly for the first time. This aspect grows in importance if the servers are fast enough. Machines are cheaper than people, so don't let this particular machine be the area you save money on.

Build slaves

To reduce build queues, you can add build slaves. The master server will send builds to the slaves based on a round-robin scheme or tie specific builds to specific build slaves.

The reason for this is usually that some builds have certain requirements on the host operating system.

Build slaves can be used to increase the efficiency of parallel builds. They can also be used to build software on different operating systems. For instance, you can have a Linux Jenkins master server and Windows slaves for components that use Windows build tools. To build software for the Apple Mac, it's useful to have a Mac build slave, especially since Apple has quirky rules regarding the deployment of their operating system on virtual servers.

There are several methods to add build slaves to a Jenkins master; see https://wiki.jenkins-ci.org/display/JENKINS/Distributed+builds.

In essence, there must be a way for the Jenkins master to issue commands to the build slave. This command channel can be the classic SSH method, and Jenkins has a built-in SSH facility. You can also start a Jenkins slave by downloading a Java JNLP client from the master to the slave. This is useful if the build slave doesn't have an SSH server.

Software on the host

Depending on the complexity of your builds, you might need to install many different types of build tool on your build server. Remember that Jenkins is mostly used to trigger builds, not perform the builds themselves. That job is delegated to the build system used, such as Maven or Make.

It's most convenient to have a Linux-based host operating system. Most of the build systems are available in the distribution repositories, so it's very convenient to install them from there.

To keep your build server up to date, you can use the same deployment servers that you use to keep your application servers up to date

Triggers

You can either use a timer to trigger builds, or you can poll the code repository for changes and build if there were changes.

It can be useful to use both methods at the same time:

- Git repository polling can be used most of the time so that every check in triggers a build.
- Nightly builds can be triggered, which are more stringent than continuous builds, and thus take a longer time. Since these builds happen at night when nobody is supposed to work, it doesn't matter if they are slow.
- An upstream build can trigger a downstream build.

You can also let the successful build of one job trigger another job.

Job chaining and build pipelines
It's often useful to be able to chain jobs together. In its simplest form, this works by triggering a second job in the event that the first job finishes successfully. Several jobs can be cascaded this way in a chain. Such a build chain is quite often good enough for many purposes. Sometimes, a nicer visualization of the build steps as well as greater control over the details of the chain is desired.

In Jenkins terminology, the first build in a chain is called the upstream build, and the second one is called the downstream build.

While this way of chaining builds is often sufficient, there will most likely be a need for greater control of the build chain eventually. Such a build chain is often called a pipeline or workflow.

There are many plugins that create improved pipelines for Jenkins, and the fact that there are several shows that there is indeed a great desire for improvements in this area.

Two examples are the multijob plugin and the workflow plugin.

The workflow plugin is the more advanced and also has the advantage that it can be described in a Groovy DSL rather than fiddling about in a web UI.

The workflow plugin is promoted by CloudBees, who are the principal contributors to Jenkins today.

A look at the Jenkins filesystem layout
It is often useful to know where builds wind up in the filesystem.

In the case of the Fedora package, the Jenkins jobs are stored here:

/var/lib/jenkins/jobs

Each job gets its own directory, and the job description XML is stored in this directory as well as a directory for the build called workspace. The job's XML files can be backed up to another server in order to be able to rebuild the Jenkins server in the event of a catastrophic failure. There are dedicated backup plugins for this purpose as well.

Builds can consume a lot of space, so it may sometimes happen that you need to clean out this space manually.

This shouldn't be the normal case, of course. You should configure Jenkins to only leave the number of builds you have space for. You can also configure your configuration management tool to clear out space if needed.

Another reason you might need to delve into the file system is when a build mysteriously fails, and you need to debug the cause of the failure. A common cause of this is when the build server state does not meet expectations. For a Maven build, broken dependencies could be polluting the local repository on the build server, for example.

Build servers and infrastructure as code

While we are discussing the Jenkins file structure, it is useful to note an impedance mismatch that often occurs between GUI-based tools such as Jenkins and the DevOps axiom that infrastructure should be described as code.

One way to understand this problem is that while Jenkins job descriptors are text file-based, these text files are not the primary interface for changing the job descriptors.

The web interface is the primary interface. This is both a strength and weakness.

It is easy to create ad-hoc solutions on top of existing builds with Jenkins. You don't need to be intimately familiar with Jenkins to do useful work.

On the other hand, the out-of-the-box experience of Jenkins lacks many features that we are used to from the world of programming. Basic features like inheritance and even function definitions take some effort to provide in Jenkins.

The build server feature in GitLab, for example, takes a different approach. Build descriptors are just code right from the start. It is worth checking out this feature in GitLab if you don't need all the possibilities that Jenkins offers.

Building by dependency order

Many build tools have the concept of a build tree where dependencies are built in the order required for the build to complete, since parts of the build might depend on other parts.

In Make-like tools, this is described explicitly; for instance, like this:

```
a.out : b.o c.o

b.o : b.c

c.o : c.c
```

So, in order to build a.out, b.o and c.o must be built first In tools such as Maven, the build graph is derived from the dependencies we set for an artifact. Gradle, another Java build tool, also creates a build graph before building.

Jenkins has support for visualizing the build order for Maven builds, which is called the reactor in Maven parlance, in the web user interface.

This view is not available for Make-style builds, however Build phases
One of the principal benefits of the Maven build tool is that it standardizes builds.

This is very useful for a large organization, since it won't need to invent its own build standards. Other build tools are usually much more lax regarding how to implement various build phases. The rigidity of Maven has its pros and cons. Sometimes, people who get started with Maven reminisce about the freedom that could be had with tools such as Ant.

You can implement these build phases with any tool, but it's harder to keep the habit going when the tool itself doesn't enforce the standard order: building, testing, and deploying.

We will examine testing in more detail in a later chapter, but we should note here that the testing phase is very important. The Continuous Integration server needs to be very good at catching errors, and automated testing is very important for achieving that goal.

Alternative build servers

While Jenkins appears to be pretty dominant in the build server scene in my experience, it is by no means alone. Travis CI is a hosted solution that is popular among open source projects. Buildbot is a buildserver that is written in, and configurable with, Python. The Go server is another one, from ThoughtWorks. Bamboo is an offering from Atlassian. GitLab also supports build server functionality now.

Do shop around before deciding on which build server works best for you.

When evaluating different solutions, be aware of attempts at vendor lock-in. Also keep in mind that the build server does not in any way replace the need for builds that are well behaved locally on a developer's machine.

Also, as a common rule of thumb, see if the tool is configurable via configuration files. While management tends to be impressed by graphical configuration, developers and operations personnel rarely like being forced to use a tool that can only be configured via a graphical user interface.

Collating quality measures

A useful thing that a build server can do is the collation of software quality metrics. Jenkins has some support for this out of the box. Java unit tests are executed and can be visualized directly on the job page.

Another more advanced option is using the Sonar code quality visualizer, which is shown in the following screenshot. Sonar tests are run during the build phase and propagated to the Sonar server, where they are stored and visualized.

A Sonar server can be a great way for a development team to see the fruits of their efforts at improving the code base.

The drawback of implementing a Sonar server is that it sometimes slows down the builds. The recommendation is to perform the Sonar builds in your nightly builds, once a day.

About build status visualization

The build server produces a lot of data that is amenable to visualization on a shared display. It is useful to be immediately aware that a build has failed, for instance.

The easiest thing is to just hook up a monitor in a kiosk-like configuration with a web browser pointing to your build server web interface. Jenkins has many plugins that provide a simplified job overview suitable for kiosk displays. These are sometimes called information radiators.

It is also common to hook up other types of hardware to the build status, such as lava lamps or colorful LED lamps.

In my experience, this kind of display can make people enthusiastic about the build server. Succeeding with having a useful display in the long run is more tricky than it would first appear, though. The screen can be distracting. If you put the screen where it's not easily seen in order to circumvent the distraction, the purpose of the display is defeated.

A lava lamp in combination with a screen placed discreetly could be a useful combination. The lava lamp is not normally lit, and thus not distracting. When a build error occurs, it lights up, and then you know that you should have a look at the build information radiator. The lava lamp even conveys a form of historical record
of the build quality. As the lava lamp lights up, it grows warm, and after a while, the lava moves around inside the lamp. When the error is corrected, the lamp cools down, but the heat remains for a while, so the lava will move around for a time proportional to how long it took to fix the build error.

Taking build errors seriously

The build server can signal errors and code quality problems as much as it wants; if developer teams don't care about the problems, then the investment in the notifications and visualization is all for nought.

This isn't something that can be solved by technical means alone. There has to be a process that everybody agrees on, and the easiest way for a consensus to be achieved is for the process to be of obvious benefit to everyone involved.

Part of the problem is organizations where everything is on fire all the time. Is a build error more important than a production error? If code quality measures estimate that it will take years to improve a code base's quality, is it worthwhile to even get started with fixing the issues? How do we solve these kinds of problems? Here are some ideas:

- Don't overdo your code quality metrics. Reduce testing until reports show levels that are fixable. You can add tests again after the initial set of problems is taken care of.
- Define a priority for problems. Fix production issues first. Then fix build errors. Do not submit new code on top of broken code until the issues are resolved.

Robustness

While it is desirable that the build server becomes one of the focal points in your Continuous Delivery pipeline, also consider that the process of building and deployment should not come to a standstill in the event of a breakdown of the build server. For this reason, the builds themselves should be as robust as possible and repeatable on any host.

This is fairly easy for some builds, such as Maven builds. Even so, a Maven build can exhibit any number of flaws that makes it non-portable.

A C-based build can be pretty hard to make portable if one is not so fortunate as to have all build dependencies available in the operating system repositories. Still, robustness is usually worth the effort.

UNIT - V
Testing Tools and automation: Various types of testing, Automation of testing Pros and cons, Selenium - Introduction, Selenium features, JavaScript testing, Testing backend integration points, Test-driven development, REPL-driven development Deployment of the system: Deployment systems, Virtualization stacks, code execution at the client, Puppet master and agents, Ansible, Deployment tools: Chef, Salt Stack and Docker

Various types of testing

If we are going to release our code early and often, we ought to be confident of its quality. Therefore, we need automated regression testing.

Here some frameworks for software testing are explored, such as **JUnit for unit testing and Selenium for web frontend testing**. We will also find out how these tests are run in our Continuous Integration server, Jenkins, thus forming the first part of our Continuous Delivery pipeline.

Testing is very important for software quality, and it's a very large subject in itself. We will concern

ourselves with these topics in this chapter:

How to make manual testing easier and less error prone

Various types of testing, such as unit testing, and how to perform them in practice

Automated system integration testing

We already had a look at how to accumulate test data with Sonar and Jenkins in the previous chapter, and we will continue to delve deeper into this subject.

Manual testing

Even if test automation has larger potential benefits for DevOps than manual testing, manual testing will always be an important part of software development. If nothing else, we will need to perform our tests manually at least once in order to automate them.

Acceptance testing in particular is hard to replace, even though there have been attempts to do so. Software requirement specifications can be terse and hard to understand even for the people developing the features that implement those requirements. In these cases, quality assurance people with their eyes on the ball are invaluable and irreplaceable.

The things that make manual testing easier are the same things that make automated integration testing easier as well, so there is a synergy to achieve between the different testing strategies.

In order to have happy quality assurance personnel, you need to:

- Manage test data, primarily the contents of backend databases, so that tests give the same results when you run them repeatedly
- Be able to make rapid deployments of new code in order to verify bug fixes

Obvious as this may seem, it can be hard in practice. Maybe you have large production databases that can't just be copied to test environments. Maybe they contain end-user data that needs to be protected under law. In these cases, you need to de-identify the data and wash it of any personal details before deploying it to test environments.

Each organization is different, so it is not possible to give generally useful advice in this area other than the KISS rule: "Keep it simple, stupid."

Pros and cons with test automation

When you talk with people, most are enthusiastic about the prospect of test automation. Imagine all the benefits that await us with it:

- Higher software quality
- Higher confidence that the software releases we make will work as intended
- Less of the monotonous tedium of laborious manual testing. All very good

and desirable things!

In practice, though, if you spend time with different organizations with complex multi-tiered products, you will notice people talking about test automation, but you will also notice a suspicious absence of test automation in practice. Why is that?

If you just compile programs and deploy them once they pass compilation, you will likely be in for a bad experience. Software testing is completely necessary for a program to work reliably in the real world. Manual testing is too slow to achieve Continuous Delivery. So, we need test automation to succeed with Continuous Delivery. Therefore, let's further investigate the problem areas surrounding test automation and see if we can figure out what to do to improve the situation:

- Cheap tests have lower value.

  One problem is that the type of test automation that is fairly cheap to produce, unit testing, typically has lower perceived value than other types of testing. Unit testing is still a good type of testing, but manual testing might be perceived as exposing more bugs in practice. It might then feel unnecessary to write unit tests.

- It is difficult to create test cradles that are relevant to automated integration testing.

  While it is not very difficult to write test cradles or test fixtures for unit tests, it tends to get harder as the test cradle becomes more production-like. This can be because of a lack of hardware resources, licensing, manpower, and so on.

- The functionality of programs vary over time and tests must be adjusted accordingly, which takes time and effort.

  This makes it seem as though test automation just makes it harder to write software without providing a perceived benefit.

  This is especially true in organizations where developers don't have a close relationship with the people working with operations, that is, a non DevOps oriented organization. If someone else will have to deal with your crappy code that doesn't really work as intended, there is no real cost involved for the developers. This isn't a healthy relationship. This is the central problem DevOps aims to solve. The DevOps approach bears this repeating rule: help people with different roles work closer together. In organizations like Netflix, an Agile team is entirely responsible for the success, maintenance, and outages of their service.

- It is difficult to write robust tests that work reliably in many different build scenarios.

  A consequence of this is that developers tend to disable tests in their local builds so that they can work undisturbed with the feature they have been assigned. Since people don't work with the tests, changes that The build server will pick up the build error, but nobody remembers how the test works now, and it might take several days to fix the test error. While the test is broken, the build displays will show red, and eventually, people will stop caring about build issues. Someone else will fix the problem eventually.

- It is just hard to write good automated tests, period.

  It can indeed be hard to create good automated integration tests. It can also be rewarding, because you get to learn all the aspects of the system you are testing.

  These are all difficult problems, especially since they mostly stem from people's perceptions and relationships.

There is no panacea, but I suggest adopting the following strategy:

- Leverage people's enthusiasm regarding test automation
- Don't set unrealistic goals
- Work incrementally

Unit testing

Unit testing is the sort of testing that is normally close at heart for developers.

The primary reason is that, by definition, unit testing tests well-defined parts of the system in isolation from other parts. Thus, they are comparatively easy to write and use.

Many build systems have built-in support for unit tests, which can be leveraged without undue difficulty.

With Maven, for example, there is a convention that describes how to write tests such that the build system can find them, execute them, and finally prepare a report of the outcome. Writing tests basically boils down to writing test methods, which are tagged with source code annotations to mark the methods as being tests. Since they are ordinary methods, they can do anything, but by convention, the tests should be written so that they don't require considerable effort to run. If the test code starts to require complicated setup and runtime dependencies, we are no longer dealing with unit tests.

Here, the difference between unit testing and functional testing can be a source of confusion. Often, the same underlying technologies and libraries are reused between unit and functional testing.

This is a good thing as reuse is good in general and lets you benefit from your expertise in one area as you work on another. Still, it can be confusing at times, and it pays to raise your eyes now and then to see that you are doing the right thing.

JUnit in general and JUnit in particular

You need something that runs your tests. JUnit is a framework that lets you define unit tests in your Java code and run them.

JUnit belongs to a family of testing frameworks collectively called xUnit. The SUnit is the grandfather of this family and was designed by Kent Beck in 1998 for the Smalltalk language.

While JUnit is specific to Java, the ideas are sufficiently generic for ports to have been made in, for instance, C#. The corresponding test framework for C# is called, somewhat unimaginatively, NUnit. The N is derived from .NET, the name of the Microsoft software platform.

We need some of the following nomenclature before carrying on. The nomenclature is not specific to JUnit, but we will use JUnit as an example to make it easier to relate to the definitions.

- Test runner: A test runner runs tests that are defined by an xUnit framework.

  JUnit has a way to run unit tests from the command line, and Maven employs a test runner called Surefire. A test runner also collects and reports test results. In the case of Surefire, the reports are in XML format, and these reports can be further processed by other tools, particularly for visualization.

- Test case: A test case is the most fundamental type of test definition.

  How you create test cases differs a little bit among JUnit versions. In earlier versions, you inherited from a JUnit base class; in recent versions, you just need to annotate the test methods. This is better since Java doesn't support multiple inheritance and you might want to use your own inheritance hierarchies rather than the JUnit ones. By convention, Surefire also locates test classes that have the Test suffix in the class name.

- Test fixtures: A test fixture is a known state that the test cases can rely on so that the tests can have well-defined behavior. It is the responsibility of the developer to create these. A test fixture is also sometimes known as a test context.

  With JUnit, you usually use the @Before and @After annotations to define test fixtures. @Before is, unsurprisingly, run before a test case and is used to bring up the environment. @After likewise restores the state if there is a need to.

  Sometimes, @Before and @After are more descriptively named Setup and Teardown. Since annotations are used, the method can have the names that are the most intuitive in that context.

- Test suites: You can group test cases together in test suites. A test suite is usually a set of test cases that share the same test fixture.
- Test execution: A test execution runs the tests suites and test cases.

  Here, all the previous aspects are combined. The test suites and test cases are located, the appropriate test fixtures are created, and the test cases run. Lastly, the test results are collected and collated.

- Test result formatter: A test result formatter formats test result output for human consumption. The format employed by JUnit is versatile enough to be used by other testing frameworks and formatters not directly associated with JUnit. So, if you have some tests that don't really use any of the xUnit frameworks, you can still benefit by presenting the test results in Jenkins by providing a test result XML file. Since the file format is XML, you can produce it from your own tool, if need be.
- Assertions: An assertion is a construct in the xUnit framework that makes sure that a condition is met. If it is not met, it is considered an error, and a test error is reported. The test case is also usually terminated when the assertion fails.

JUnit has a number of assertion methods available. Here is a sample of the available assertion methods:

- To check whether two objects are equal:

  assertEquals(str1, str2);

- To check whether a condition is true:

  assertTrue (val1 < val2);

- To check whether a condition is false:

assertFalse(val1 > val2);

A JUnit example

JUnit is well supported by Java build tools. It will serve well as an example of JUnit testing frameworks in general.

If we use Maven, by convention, it will expect to find test cases in the following directory:

/src/test/java

Mocking
Mocking refers to the practice of writing simulated resources to enable unit testing. Sometimes, the words "fake" or "stub" are used.
For example, a middleware system that responds with JSON structures from a database would "mock" the database backend for its unit tests. Otherwise, the unit tests would require the database backend to be online, probably also requiring exclusive access. This wouldn't be convenient.

Mockito is a mocking framework for Java that has also been ported to Python.

Test Coverage
When you hear people talk about unit testing, they often talk about test coverage. Test coverage is the percentage of the application code base that is actually executed by the test cases.

In order to measure unit test code coverage, you need to execute the tests and keep track of the code that has or hasn't been executed.

Cobertura is a test coverage measurement utility for Java that does this. Other such utilities include jcoverage and Clover.

Cobertura works by instrumenting the Java bytecode, inserting code fragments of its own into already compiled code. These code fragments are executed while measuring code coverage during execution of test cases

Its usually assumed that a hundred percent test coverage is the ideal. This might not always be the case, and one should be aware of the cost/benefit trade-offs.

A simple counterexample is a simple getter method in Java:

```
private int positiveValue; void setPositiveValue(int  x){

   this.positiveValue=x;

}


int getPositiveValue(){

return positiveValue;

}
```

If we write a test case for this method, we will achieve a higher test coverage. On the other hand, we haven't achieved much of anything, in practice. The only thing we are really testing is that our Java implementation doesn't have bugs.

If, on the other hand, the setter is changed to include validation to check that the value is not a negative number, the situation changes. As soon as a method includes logic of some kind, unit testing is useful.

Automated integration testing

Automated integration testing is similar in many ways to unit testing with respect to the basic techniques that are used. You can use the same test runners and build system support. The primary difference with unit testing is that less mocking is involved.

Where a unit test would simply mock the data returned from a backend database, an integration test would use a real database for its tests. A database is a decent example of the kind of testing resources you need and what types of problems they could present.

Automated integration testing can be quite tricky, and you need to be careful with your choices.

If you are testing, say, a read-only middleware adapter, such as a SOAP adapter for a database, it might be possible to use a production database copy for your testing. You need the database contents to be predictable and repeatable; otherwise, it will be hard to write and run your tests.

The added value here is that we are using a production data copy. It might contain data that is hard to predict if you were to create test data from scratch. The requirements are the same as for manual testing. With automated integration testing, you need, well, more automation than with manual testing. For databases, this doesn't have to be very complicated. Automated database backup and restore are well-known operations.

Docker in automated testing

Docker can be quite convenient when building automated test rigs. It adds some of the features of unit testing but at a functional level. If your application consists of several server components in a cluster, you can simulate the entire cluster with a set of containers. Docker provides a virtual network for the cluster that makes clear how the containers interact at the network level.

Docker also makes it easy to restore a container to a known state. If you have your test database in a Docker container, you can easily restore the database to the same state as before the tests taking place. This is similar to restoring the environment in the After method in unit tests.

The Jenkins Continuous Integration server has support for the starting and stopping of containers, which can be useful when working with Docker test automation.

Using Docker Compose to run the containers you need is also a useful option.

Docker is still young, and some aspects of using Docker for test automation can require glue code that is less than elegant.

A simple example could be firing up a database container and an application server container that communicate with each other. The basic process of starting the containers is simple and can be done with a shell script or Docker Compose.
But, since we want to run tests on the application server container that has been started, how do we know whether it was properly started? In the case of the WildFly container, there is no obvious way to determine the running state apart from watching the log output for occurrences of strings or maybe

polling a web socket. In any case, these types of hacks are not very elegant and are time consuming to write. The end result can be well worth the effort, though.

Arquillian

Arquillian is an example of a test tool that allows a level of testing closer to integration testing than unit testing together with mocking allows. Arquillian is specific to Java application servers such as WildFly. Arquillian is interesting because it illustrates the struggle of reaching a closer approximation of production systems during testing. You can reach such approximations in any number of ways, and the road to there is filled with trade-offs.

There is a "hello world" style demonstration of Arquillian in the book's source code archive.

Performance testing

Performance testing is an essential part of the development of,for instance, large public web sites.

Performance testing presents similar challenges as integration testing. We need a testing system that is similar to a production system in order for the performance test data to be useful to make a forecast about real production system performance.

The most commonly used performance test is load testing. With load testing, we measure, among other things, the response time of a server while the performance testing software generates synthetic requests for the server.

Apache JMeter is an example of a an open source application for measuring performance. While it's simpler than its proprietary counterparts, such as LoadRunner, JMeter is quite useful, and simplicity is not really a bad thing.

JMeter can generate simulated load and measure response times for a number of protocols, such as HT, LDAP, SOAP, and JDBC.

There is a JMeter Maven plugin, so you can run JMeter as part of your build.

JMeter can also be used in a Continuous Integration server. There is a plugin for Jenkins, called the performance plugin, that can execute JMeter test scenarios.

Ideally, the Continuous Integration server will deploy code that has been built to a test environment that is production-like. After deployment, the performance tests will be executed and test data collected, as shown in this screenshot:

Automated acceptance testing

Automated acceptance testing is a method of ensuring that your testing is valid from the end user's point of view.

Cucumber is a framework where test cases are written in plaintext and associated with test code. This is called behavior-driven development. The original implementation of Cucumber was written in Ruby, but ports now exist for many different languages.

The appeal of Cucumber from a DevOps point of view is that it is intended to bring different roles together. Cucumber feature definitions are written in a conversational style that can be achieved without programming skills. The hard data required for test runs is then extracted from the descriptions and used for the tests.

While the intentions are good, there are difficulties in implementing Cucumber that might not immediately be apparent. While the language of the behavior specifications is basically free text, they still need to be somewhat spartan and formalized; otherwise, it becomes difficult to write matching code that extracts the test data from the descriptions. This makes writing the specifications less attractive to the roles that were supposed to write them in the first place. What then happens is that programmers write the specifications, and they often dislike the verbosity and resort to writing ordinary unit tests.

As with many things, cooperation is of the essence here. Cucumber can work great when developers and product owners work together on writing the specifications in a way that works for everyone concerned.

Now, let's look at a small "hello world" style example for Cucumber.

Cucumber works with plaintext files called feature files, which look like this:

```
Feature: Addition

I would like to add numbers with my pocket calculator

Scenario: Integer numbers

I have entered 4 into the calculator

I press add

I have entered 2 into the calculator

I press equal

The result should be 6 on the screen
```

The feature description is implementation-language neutral. Describing the Cucumber test code is done in a vocabulary called Gherkin.

If you use the Java 8 lambda version of Cucumber, a test step could look somewhat like this:

```
Calculator calc;

public MyStepdefs() {

Given("I have entered (\\d+) into the calculator", (Integer i) -> { System.out.format("Number
```

entered: %n\n", i); calc.push(i);});

When("I press (\\w+)", (String op) -> { System.out.format("operator entered: %n\n", op); calc.op(op); });

Then("The result should be (\\d+)", (Integer i) -> { System.out.format("result : %n\n", i); assertThat(calc.result(),i); });

This is a simple example, but it should immediately be apparent both what the strengths and weaknesses with Cucumber are. The feature descriptions have a nice human-readable flair. However, you have to match the strings with regular expressions in your test code. If your feature description changes even slightly in wording, you will have to adjust the test code.

Automated GUI testing

Automating GUI testing has many desirable properties, but it is also difficult. One reason is that user interfaces tend to change a lot during the development phase, and buttons and controls move around in the GUI.

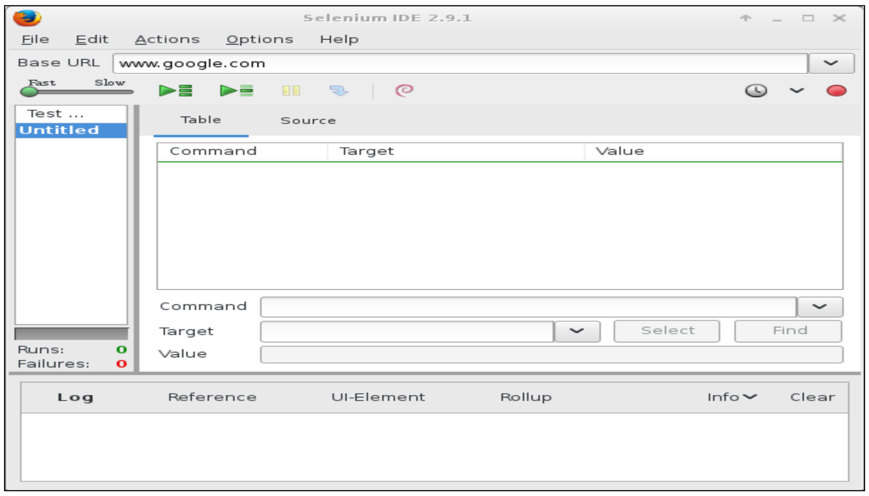
Older generations of GUI testing tools often worked by synthesizing mouse events and sending them to the GUI. When a button moved, the simulated mouse click went nowhere, and the test failed. It then became expensive to keep the tests updated with changes in the GUI.

Selenium is a web UI testing toolkit that uses a different, more effective, approach. The controllers are instrumented with identifiers so that Selenium can find the controllers by examining the document object model (DOM) rather than blindly generating mouse clicks.

Selenium works pretty well in practice and has evolved over the years.

Another method is employed by the Sikuli test framework. It uses a computer vision framework, OpenCV, to help identify controllers even if they move or change appearances. This is useful for testing native applications, such as games.

The screenshot included below is from the Selenium IDE.



Integrating Selenium tests in Jenkins

Selenium works by invoking a browser, pointing it to a web server running your application, and then remotely controlling the browser by integrating itself in the JavaScript and DOM layers.

When you develop the tests, you can use two basic methods:

Record user interactions in the browser and later save the resulting test code for reuse

Write the tests from scratch using Selenium's test API

Many developers prefer to write tests as code using the Selenium API at the outset, which can be combined with a test-driven development approach.

Regardless of how the tests are developed, they need to run in the integration build server.

This means that you need browsers installed somewhere in your test environment. This can be a bit problematic since build servers are usually headless, that is, they are servers that don't run user interfaces.

It's possible to wrap a browser in a simulated desktop environment on the build server.

A more advanced solution is using Selenium Grid. As the name implies, Selenium Grid provides a server that gives a number of browser instances that can be used by the tests. This makes it possible to run a number of tests in parallel as well as to provide a set of different browser configurations.

You can start out with the single browser solution and later migrate to the Selenium

Grid solution when you need it.

There is also a convenient Docker container that implements Selenium Grid.

JavaScript testing

Since there usually are web UI implementations of nearly every product these days, the JavaScript testing frameworks deserve special mention:

Karma is a test runner for unit tests in the JavaScript language

Jasmine is a Cucumber-like behavior testing framework

Protractor is used for AngularJS

Protractor is a different testing framework, similar to Selenium in scope but optimized for AngularJS, a popular JavaScript user interface framework. While it would appear that new web development frameworks come and go everyday, it's interesting to note why a test framework like Protractor exists when Selenium is available and is general enough to test AngularJS applications too.

First of all, Protractor actually uses the Selenium web driver implementation under the hood.

You can write Protractor tests in JavaScript, but you can use JavaScript for writing test cases for Selenium as well if you don't like writing them in a language like Java.

The main benefit turns out to be that Protractor has internalized knowledge about the Angular framework, which a general framework like Selenium can't really have.

AngularJS has a model/view setup that is particular to it. Other frameworks use other setups, since the model/view setup isn't something that is intrinsic to the JavaScript language—not yet, anyway.

Protractor knows about the peculiarities of Angular, so it's easier to locate controllers in the testing code with special constructs.

Testing backend integration points

Automated testing of backend functionality such as SOAP and REST endpoints is normally quite cost effective. Backend interfaces tend to be fairly stable, so the corresponding tests will also require less maintenance effort than GUI tests, for instance.

The tests can also be fairly easy to write with tools such as soapUI, which can be used to write and execute tests. These tests can also be run from the command line and with Maven, which is great for Continuous Integration on a build server.

The soapUI is a good example of a tool that appeals to several different roles. Testers who build test cases get a fairly well-structured environment for writing tests and running them interactively. Tests can be built incrementally.

Developers can integrate test cases in their builds without necessarily using the GUI. There are Maven plugins and command-line runners.

The command line and Maven integration are useful for people maintaining the build server too.

Furthermore, the licensing is open source with some added features in a separate, proprietary version. The open source nature makes the builds more reliable. It is very stress-inducing when a build fails because a license has unexpectedly reached its end or a floating license has run out.

The soapUI tool has its share of flaws, but in general, it is flexible and works well.

Here's what the user interface looks like:

The soapUI user interface is fairly straightforward. There is a tree view listing test cases on the left. It is possible to select single tests or entire test suites and run them. The results are presented in the area on the right.

It is also worth noting that the test cases are defined in XML. This makes it possible to manage them as code in the source code repository. This also makes it possible to edit them in a text editor on occasion, for instance, when we need to perform a global search and replace on an identifier that has changed names—just the way we like it in DevOps!

Test-driven development

Test-driven development (TDD) has an added focus on test automation. It was made popular by the Extreme programming movement of the nineties.

TDD is usually described as a sequence of events, as follows:

- Implement the test: As the name implies, you start out by writing the test and write the code afterwards. One way to see it is that you implement the interface specifications of the code to be developed and then progress by writing the code. To be able to write the test, the developer must find all relevant requirement specifications, use cases, and user stories.

  The shift in focus from coding to understanding the requirements can be beneficial for implementing them correctly.

- Verify that the new test fails: The newly added test should fail because there is nothing to implement the behavior properly yet, only the stubs and interfaces needed to write the test. Run the test and verify that it fails.
- Write code that implements the tested feature: The code we write doesn't yet have to be particularly elegant or efficient. Initially, we just want to make the new test pass.
- Verify that the new test passes together with the old tests: When the new test passes, we know that we have implemented the new feature correctly. Since the old tests also pass, we haven't broken existing functionality.
- Refactor the code: The word "refactor" has mathematical roots. In programming, it means cleaning up the code and, among other things, making it easier to understand and maintain. We need to refactor since we cheated a bit earlier in the development.

TDD is a style of development that fits well with DevOps, but it's not necessarily the only one. The primary benefit is that you get good test suites that can be used in Continuous Integration tests.

REPL-driven development

While REPL-driven development isn't a widely recognized term, it is my favored style of development and has a particular bearing on testing. This style of development is very common when working with interpreted languages, such as Lisp, Python, Ruby, and JavaScript.

When you work with a Read Eval Print Loop (REPL), you write small functions that are independent and also not dependent on a global state. The functions are tested even as you write them.
This style of development differs a bit from TDD. The focus is on writing small functions with no or

very few side effects. This makes the code easy to comprehend rather than when writing test cases before functioning code is written, as in TDD.

You can combine this style of development with unit testing. Since you can use REPL-driven development to develop your tests as well, this combination is a very effective strategy.

A complete test automation scenario

We have looked at a number of different ways of working with test automation. Assembling the pieces into a cohesive whole can be daunting.

In this section, we will have a look at a complete test automation example, continuing from the user database web application for our organization, Matangle.

You can find the source code in the accompanying source code.

The application consists of the following layers:

- A web frontend
- A JSON/REST service interface
- An application backend layer
- A database layer

The test code will work through the following phases during execution:

- Unit testing of the backend code
- Functional testing of the web frontend, performed with the Selenium web testing framework
- Functional testing of the JSON/REST interface, executed with soapUI

All the tests are run in sequence, and when all of them succeed, the result can be used as the basis for a decision to see whether the application stack is deemed healthy enough to deploy to a test environment, where manual testing can commence.

Manually testing our web application

Before we can automate something usefully, we need to understand the details of what we will be automating. We need some form of a test plan.

Below, we have a test plan for our web application. It details the steps that a human tester needs to perform by hand if no test automation is available. It is similar to what a real test plan would look like, except a real plan would normally have many more formalities surrounding it. In our case, we will go directly to the details of the test in question:

1. Start a fresh test. This resets the database backend to a known state and sets up the testing scenario so that manual testing can proceed from a known state.

   The tester points a browser to the application's starting URL:

Running the automated test

The test is available in a number of flavors in the source bundle. To run the first one, you need a Firefox installation.

Choose the one called autotest_v1, and run it from the command line:

```
autotest_v1/bin/autotest.sh
```

If all goes well, you will see a Firefox window open, and the test you previously performed by hand will be done automatically. The values you filled in and the links you clicked on by hand will all be automated.

This isn't foolproof yet, because maybe the Firefox version you installed isn't compatible, or something else is amiss with the dependencies. The natural reaction to problems like these is dependency management, and we will look at a variant of dependency management using Docker shortly.

Finding a bug

Now we will introduce a bug and let the test automation system find it.

As an exercise, find the string "Turing" in the test sources. Change one of the occurrences to "Tring" or some other typographical error. Just change one; otherwise, the verification code will believe there is no error and that everything is alright!

Run the tests again, and notice that the error is found by the automatic test system.

Test walkthrough

Now we have run the tests and verified that they work. We have also verified that they are able to discover the bug we created.

What does the implementation look like? There is a lot of code, and it would not be useful to reprint it in the book. It is useful, though, to give an overview of the code and have a look at some snippets of the code.

Open the autotest_v1/test/pom.xml file. It's a Maven project object model file, and it's here that all the plugins used by the tests are set up. Maven POM files are declarative XML files and the test steps are step-by-step imperative instructions, so in the latter case, Java is used.

There's a property block at the top, where dependency versions are kept. There is no real need to break out the versions; it has been used in this case to make the rest of the POM file less version-dependent:

```
<properties>
<junit.version>XXX</junit.version>
<selenium.version>XXX</selenium.version>
<cucumber.version>XXX</cucumber.version>
...
</properties>
```

Here are the dependencies for JUnit, Selenium and Cucumber:

```
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>${junit.version}</version>
</dependency>
```

```
<dependency>
<groupId>org.seleniumhq.selenium</groupId>
<artifactId>selenium-java</artifactId>
<version>${selenium.version}</version>
</dependency>
<dependency>
<groupId>info.cukes</groupId>
<artifactId>cucumber-core</artifactId>
<version>${cucumber.version}</version>
</dependency>

<dependency>
<groupId>info.cukes</groupId>
<artifactId>cucumber-java</artifactId>
<version>${cucumber.version}</version>
</dependency>

<dependency>
<groupId>info.cukes</groupId>
<artifactId>cucumber-junit</artifactId>
<version>${cucumber.version}</version>
</dependency>
```

To define the tests according to the Cucumber method, we need a feature file that describes the test steps in a human-readable language. This feature file corresponds to our previous test plan for manual tests:

```
Feature: Manage users As an Administrator I want to be able to
Create a user
Search for the user
Delete the user

Scenario: Create a named user
Given a user with the name 'Alan' And the surname 'Turing'
When the administrator clicks 'Add User' Then the user should be added

Scenario: Search for a named user Given a user with the name 'Alan' And the surname 'Turing'
When the administrator clicks 'Search for User' Then the user 'Alan Turing' should be shown

Scenario: Delete a named user
Given a user with the name 'Alan' And the surname 'Turing'
When the administrator clicks 'Delete User' Then the user should be deleted
```

The feature file is mostly plaintext with small elements of machine-readable markup. It's up to the corresponding test code to parse the plaintext of the scenarios with regexes.

It is also possible to localize the feature files to the language used within your own team. This can be useful since the feature files might be written by people who are not accustomed to English.

The feature needs actual concrete code to execute, so you need some way to bind the feature to the code.

You need a test class with some annotations to make Cucumber and JUnit
work together:

```
@RunWith(Cucumber.class) @Cucumber.Options(
glue = "matangle.glue.manageUser", features = "features/manageUser.feature",
format = {"pretty", "html:target/Cucumber"}
)
```

In this example, the names of the Cucumber test classes have, by convention, a Step suffix.

Now, you need to bind the test methods to the feature scenarios and need some way to pick out the arguments to the test methods from the feature description. With the Java Cucumber implementation, this is mostly done with Java annotations. These annotations correspond to the keywords used in the feature file:

```
@Given(".+a user with the name '(.+)'") public void addUser(String name) {
```

In this case, the different inputs are stored in member variables until the entire user interface transaction is ready to go. The sequence of operations is determined by the order in which they appear in the feature file.

To illustrate that Cucumber can have different implementations, there is also a Clojure example in the book's source code bundle.

So far, we have seen that we need a couple of libraries for Selenium and Cucumber to run the tests and how the Cucumber feature descriptor is bound to methods in our test code classes.

The next step is to examine how Cucumber tests execute Selenium test code. Cucumber test steps mostly call classes with Selenium implementation details in classes with a View suffix. This isn't a technical necessity, but it makes the test step classes more readable, since the particulars of the Selenium framework are kept in a separate class.

The Selenium framework takes care of the communication between the test code and the browser. View classes are an abstraction of the web page that we are automating. There are member variables in the view code that correspond to HTML controllers. You can describe the binding between test code member variables and HTML elements with annotations from the Selenium framework, as follows:

The Selenium framework, as follows:

```
@FindBy(id = "name") private WebElement nameInput; @FindBy(id = "surname")
private WebElement surnameInput;
```

The member variable is then used by the test code to automate the same steps that the human tester followed using the test plan. The partitioning of classes into view and step classes also makes the similarity of the step classes to a test plan more apparent. This separation of concerns is useful when people involved with testing and quality assurance work with the code.

To send a string, you use a method to simulate a user typing on a keyboard:

```
nameInput.clear(); nameInput.sendKeys(value);
```

There are a number of useful methods, such as click(), which will simulate a user clicking on the control.

Handling tricky dependencies with Docker

Because we used Maven in our test code example, it handled all code dependencies except the browser. While you could clearly deploy a browser such as Firefox to a Maven-compatible repository and handle the test dependency that way if you put your mind to it, this is normally not the way this issue is handled in the case of browsers. Browsers are finicky creatures and show wildly differing behavior in different versions. We need a mechanism to run many different browsers of many different versions.

Luckily, there is such a mechanism, called Selenium Grid. Since Selenium has a pluggable driver architecture, you can essentially layer the browser backend in a client server architecture.

To use Selenium Grid, you must first determine how you want the server part to run. While the easiest option would be to use an online provider, for didactic reasons, it is not the option we will explore here.

There is an autotest_seleniumgrid directory, which contains a wrapper to run the test using Docker in order to start a local Selenium Grid. You can try out the example by running the wrapper.

The latest information regarding how to run Selenium Grid is available on the project's GitHub page.

Selenium Grid has a layered architecture, and to set it up, you need three parts:

A RemoteWebDriver instance in your testing code. This will be the interface to Selenium Grid.

Selenium Hub, which can be seen as a proxy for browser instances.

Firefox or Chrome grid nodes. These are the browser instances that will be proxied by Hub.

The code to set up the RemoteWebDriver could look like this:

```
DesiredCapabilities capabilities = new DesiredCapabilities();
capabilities.setPlatform(Platform.LINUX);
capabilities.setBrowserName("Firefox");
capabilities.setVersion("35");
driver = new RemoteWebDriver(
new URL("http://localhost:4444"), capabilities);
```

The code asks for a browser instance with a particular set of capabilities. The system will do its best to oblige.

The code can only work if there is a Selenium Grid Hub running with a Firefox node attached.

Here is how you start Selenium Hub using its Docker packaging:

```
docker run -d -p 4444:4444 --name selenium-hub selenium/hub
```

And here is how you can start a Firefox node and attach it to Hub:

```
docker run -d --link selenium-hub:hub selenium/node-firefox
```

Virtualization stacks

Organizations that have their own internal server farms tend to use virtualization a lot in order to encapsulate the different components of their applications. There are many different solutions depending on your requirements.

Virtualization solutions provide virtual machines that have virtual hardware, such as network cards and CPUs. Virtualization and container techniques are sometimes confused because they share some similarities.

You can use virtualization techniques to simulate entirely different hardware than the one you have physically. This is commonly referred to as emulation.

If you want to emulate mobile phone hardware on your developer machine so that you can test your mobile application, you use virtualization in order to emulate a device. The closer the underlying hardware is to the target platform, the greater the efficiency the emulator can have during emulation. As an example, you can use the QEMU emulator to emulate an Android device. If you emulate an Android x86_64 device on an x86_64-based developer machine, the emulation will be much more efficient than if you emulate an ARM-based Android device on an x86_64-based developer machine.

With server virtualization, you are usually not really interested in the possibility of emulation. You are interested instead in encapsulating your application's server components. For instance, if a server application component starts to run amok and consume unreasonable amounts of CPU time or other resources, you don't want the entire physical machine to stop working altogether.

This can be achieved by creating a virtual machine with, perhaps, two cores on a machine with 64 cores. Only two cores would be affected by the runaway application. The same goes for memory allocation.

Container-based techniques provide similar degrees of encapsulation and control over resource allocation as virtualization techniques do. Containers do not normally provide the emulation features of virtualization, though. This is not an issue since we rarely need emulation for server applications.

The component that abstracts the underlying hardware and arbitrates hardware resources between different competing virtual machines is called a hypervisor.

The hypervisor can run directly on the hardware, in which case it is called a bare metal hypervisor. Otherwise, it runs inside an operating system with the help of the operating system kernel.

VMware is a proprietary virtualization solution, and exists in desktop and server hypervisor variants. It is well supported and used in many organizations. The server variant changes names sometimes; currently, it's called VMware ESX, which is a bare metal hypervisor.

KVM is a virtualization solution for Linux. It runs inside a Linux host operating system. Since it is an open source solution, it is usually much cheaper than proprietary solutions since there are no licensing costs per instance and is therefore popular with organizations that have massive amounts of virtualization.

Xen is another type of virtualization which, amongst other features, has paravirtualization. Paravirtualization is built upon the idea that that if the guest operating system can be made to use a modified kernel, it can execute with greater efficiency. In this way, it sits somewhere between full CPU emulation, where a fully independent kernel version is used, and container-based virtualization, where the host kernel is used.

VirtualBox is an open source virtualization solution from Oracle. It is pretty popular with developers and sometimes used with server installations as well but rarely on a larger scale. Developers who use Microsoft Windows on their developer machines but want to emulate Linux server environments locally often find VirtualBox handy.
Likewise, developers who use Linux on their workstations find it useful to emulate Windows machines.

What the different types of virtualization technologies have in common is that they provide APIs in order to allow the automation of virtual machine management. The libvirt API is one such API that can be used with several different underlying hypervisors, such as KVM, QEMU, Xen, and LXC

Executing code on the client(Code execution at the client)

Several of the configuration management systems described here allow you to reuse the node descriptors to execute code on matching nodes. This is sometimes convenient. For example, maybe you want to run a directory listing command on all HTTP servers facing the public Internet, perhaps for debugging purposes.

In the Puppet ecosystem, this command execution system is called Marionette Collective, or MCollective for short.

A note about the exercises

It is pretty easy to try out the various deployment systems using Docker to manage the base operating system, where we will do our experiments. It is a time-saving method that can be used when developing and debugging the deployment code specific to a particular deployment system. This code will then be used for deployments on physical or virtual machines.

We will first try each of the different deployment systems that are usually possible in the local deployment modes. Further down the line, we will see how we can simulate the complete deployment of a system with several containers that together form a virtual cluster.

We will try to use the official Docker images if possible, but sometimes there are none, and sometimes the official image vanishes, as happened with the official Ansible image. Such is life in the fast-moving world of DevOps, for better or for worse.

It should be noted, however, that Docker has some limitations when it comes to emulating a full operating system. Sometimes, a container must run in elevated privilege modes. We will deal with those issues when they arise.

It should also be noted that many people prefer Vagrant for these types of tests. I prefer to use Docker when possible, because it's lightweight, fast, and sufficient most of the time.

The Puppet master and Puppet agents

Puppet is a deployment solution that is very popular in larger organizations and is one of the first systems of its kind.

Puppet consists of a client/server solution, where the client nodes check in regularly with the

Puppet server to see if anything needs to be updated in the local configuration.

The Puppet server is called a Puppet master, and there is a lot of similar wordplay in the names chosen for the various Puppet components.

Puppet provides a lot of flexibility in handling the complexity of a server farm, and as such, the tool itself is pretty complex.

This is an example scenario of a dialogue between a Puppet client and a Puppet master:

1. The Puppet client decides that it's time to check in with the Puppet master to discover any new configuration changes. This can be due to a timer or manual intervention by an operator at the client. The dialogue between the Puppet client and master is normally encrypted using SSL.
2. The Puppet client presents its credentials so that the Puppet master can know exactly which client is calling. Managing the client credentials is a separate issue.
3. The Puppet master figures out which configuration the client should have by compiling the Puppet catalogue and sending it to the client. This involves a number of mechanisms, and a particular setup doesn't need to utilize all possibilities.

   It is pretty common to have both a role-based and concrete configuration for a Puppet client. Role-based configurations can be inherited.
4. The Puppet master runs the necessary code on the client side such that the configuration matches the one decided on by the Puppet master.

In this sense, a Puppet configuration is declarative. You declare what configuration a machine should have, and Puppet figures out how to get from the current to the desired client state.

There are both pros and cons of the Puppet ecosystem:

- Puppet has a large community, and there are a lot of resources on the Internet for Puppet. There are a lot of different modules, and if you don't have a really strange component to deploy, there already is, with all likelihood, an existing module written for your component that you can modify according to your needs.
- Puppet requires a number of dependencies on the Puppet client machines. Sometimes, this gives rise to problems. The Puppet agent will require a Ruby runtime that sometimes needs to be ahead of the Ruby version available in your distribution's repositories. Enterprise distributions often lag behind in versions.
- Puppet configurations can be complex to write and test.

Ansible

Ansible is a deployment solution that favors simplicity.

The Ansible architecture is agentless; it doesn't need a running daemon on the client side like Puppet does. Instead, the Ansible server logs in to the Ansible node and issues commands over SSH in order to install the required configuration.

While Ansible's agentless architecture does make things simpler, you need a Python interpreter installed on the Ansible nodes. Ansible is somewhat more lenient about the Python version required for its code to run than Puppet is for its Ruby code to run, so this dependence on Python being available is not a great hassle in practice.

Like Puppet and others, Ansible focuses on configuration descriptors that are idempotent. This basically means that the descriptors are declarative and the Ansible system figures out how to bring the server to the desired state. You can rerun the configuration run, and it will be safe, which is not necessarily the case for an imperative system.

Let's try out Ansible with the Docker method we discussed earlier.

We will use the williamyeh/ansible image, which has been developed for the purpose, but it should be possible to use any Ansible Docker image or different ones altogether, to which we just add Ansible later.

1. Create a Dockerfile with this statement:

   ```
   FROM williamyeh/ansible:centos7
   ```

2. Build the Docker container with the following command:

   ```
   docker build .
   ```

   This will download the image and create an empty Docker container that we can use.

   Normally, you would, of course, have a more complex Dockerfile that can add the things we need, but in this case, we are going to use the image interactively, so we will instead mount the directory with Ansible files from the host so that we can change them on the host and rerun them easily.

3. Run the container.

   The following command can be used to run the container. You will need the hash from the previous build command:

   ```
   docker run -v `pwd`/ansible:/ansible -it <hash> bash
   ```

   Now we have a prompt, and we have Ansible available. The -v trick is to make parts of the host filesystem visible to the Docker guest container.
   The files will be visible in the /ansible directory in the container.

The playbook.yml file is as follows:

```
---

- hosts: localhost vars:

    http_port: 80

    max_clients:  200  remote_user:  root
  tasks:
```

```
- name: ensure apache is at the latest version yum: name=httpd
  state=latest
```

This playbook doesn't do very much, but it demonstrates some concepts of Ansible playbooks.

Now, we can try to run our Ansible playbook:

```
cd /ansible

ansible-playbook -i inventory playbook.yml        --connection=local --sudo
```

The output will look like this:

```
PLAY [localhost] ******************************************************

******

GATHERING FACTS ******************************************************

******

ok: [localhost]

TASK: [ensure apache is at the latest version] *************************

******

ok: [localhost]

PLAY RECAP ************************************************************

******

localhost                    : ok=2      changed=0     unreachable=0 failed=0
```

Tasks are run to ensure the state we want. In this case, we want to install Apache's httpd using yum, and we want httpd to be the latest version.

To proceed further with our exploration, we might like to do more things, such as starting services automatically. However, here we run into a limitation with the approach of using Docker to emulate physical or virtual hosts. Docker is a container technology, after all, and not a full-blown virtualization system. In Docker's normal use case scenarios, this doesn't matter, but in our case, we need make some workarounds in order to proceed. The main problem is that the systemd init system requires special care to run in a container.
Developers at Red Hat have worked out methods of doing this. The following is a slightly modified version of a Docker image by Vaclav Pavlin, who works with Red Hat:

```
FROM fedora
```

```
RUN yum -y update; yum clean all RUN yum install
ansible sudo

RUN systemctl mask systemd-remount-fs.service dev-hugepages.mount sys- fs-fuse-
connections.mount systemd-logind.service getty.target console- getty.service

RUN cp /usr/lib/systemd/system/dbus.service /etc/systemd/system/; sed

-i 's/OOMScoreAdjust=-900//' /etc/systemd/system/dbus.service


VOLUME ["/sys/fs/cgroup", "/run", "/tmp"] ENV
container=docker

CMD ["/usr/sbin/init"]
```

The environment variable container is used to tell the systemd init system that it runs inside a container and to behave accordingly.

We need some more arguments for docker run in order to enable systemd to work in the container:

```
docker run -it --rm -v /sys/fs/cgroup:/sys/fs/cgroup:ro -v `pwd`/ ansible:/ansible <hash>
```

The container boots with systemd, and now we need to connect to the running container from a different shell:

```
docker exec -it <hash> bash
```

Phew! That was quite a lot of work just to get the container more lifelike! On the other hand, working with virtual machines, such as VirtualBox, is even more cumbersome in my opinion. The reader might, of course, decide differently.

Now, we can run a slightly more advanced Ansible playbook inside the container, as follows:

```
---

- hosts: localhost vars:
    http_port: 80

    max_clients: 200 remote_user: root
  tasks:

  - name: ensure apache is at the latest version yum: name=httpd
    state=latest
  - name: write the apache config file
    template: src=/srv/httpd.j2 dest=/etc/httpd.conf notify:

    - restart apache
    - name: ensure apache is running (and enable it at boot) service: name=httpd
      state=started enabled=yes
    handlers:

      - name: restart apache
        service: name=httpd state=restarted
```

This example builds on the previous one, and shows you how to:

- Install a package
- Write a template file
- Handle the running state of a service

The format is in a pretty simple YML syntax.

PalletOps

PalletOps is an advanced deployment system, which combines the declarative power of Lisp with a very lightweight server configuration.

PalletOps takes Ansible's agentless idea one step further. Rather than needing a Ruby or Python interpreter installed on the node that is to be configured, you only need ssh and a bash installation. These are pretty simple requirements.

PalletOps compiles its Lisp-defined DSL to Bash code that is executed on the slave node. These are such simple requirements that you can use it on very small and simple servers—even phones!

On the other hand, while there are a number of support modules for Pallet called crates, there are fewer of them than there are for Puppet or Ansible.

Deployment tools:

Deploying with Chef

Chef is a Ruby-based deployment system from Opscode.

It is pretty easy to try out Chef; for fun, we can do it in a Docker container so we don't pollute our host environment with our experiments:

```
docker run -it ubuntu
```

We need the curl command to proceed with downloading the chef installer:

```
apt-get -y install curl
curl -L https://www.opscode.com/chef/install.sh | bash
```

The Chef installer is built with a tool from the Chef team called omnibus. Our aim here is to try out a Chef tool called chef-solo. Verify that the tool is installed:

```
chef-solo -v
```

This will give output as:

```
Chef: 12.5.1
```

The point of chef-solo is to be able to run configuration scripts without the full infrastructure of the

configuration system, such as the client/server setup. This type of testing environment is often useful when working with configuration systems, since it can be hard to get all the bits and pieces in working order while developing the configuration that you are going to deploy.
Chef prefers a file structure for its files, and a pre-rolled structure can be retrieved from GitHub. You can download and extract it with the following commands:

```
curl -L http://github.com/opscode/chef-repo/tarball/master -o master.tgz tar -zxf master.tgz
mv chef-chef-repo* chef-repo rm master.tgz
```

You will now have a suitable file structure prepared for Chef cookbooks, which looks like the following:

```
./cookbooks
./cookbooks/README.md
./data_bags
./data_bags/README.md
./environments
./environments/README.md
./README.md
./LICENSE
./roles
./roles/README.md
./chefignore
```

You will need to perform a further step to make everything work properly, telling chef where to find its cookbooks as:

```
mkdir .chef
```

echo "cookbook_path ['/root/chef-repo/cookbooks']" > .chef/knife.rb Now we can use the knife tool to create a template for a configuration, as follows: knife cookbook
create phpapp

Deploying with SaltStack
SaltStack is a Python-based deployment solution.
There is a convenient dockerized test environment for Salt, by Jackson Cage. You can start it with the following:

```
docker run -i -t --name=saltdocker_master_1 -h master -p 4505 -p 4506 \
-p 8080 -p 8081 -e SALT_NAME=master -e SALT_USE=master \
-v `pwd`/srv/salt:/srv/salt:rw jacksoncage/salt
```

This will create a single container with both a Salt master and a Salt minion. We can create a shell inside the container for our further explorations: docker exec -i -t saltdocker_master_1 bash
We need a configuration to apply to our server. Salt calls configurations "states", or Salt states.
In our case, we want to install an Apache server with this simple Salt state:

```
top.sls:
base:
'*':
webserver

webserver.sls:
```

```
apache2:            # ID declaration pkg:  # state declaration
installed   # function declaration
```

Salt uses .yml files for its configuration files, similar to what Ansible does.

The file top.sls declares that all matching nodes should be of the type webserver. The webserver state declares that an apache2 package should be installed, and that's basically it. Please note that this will be distribution dependent. The Salt Docker test image we are using is based on Ubuntu, where the Apache web server package is called apache2. On Fedora for instance, the Apache web server package is instead simply called httpd.

Run the command once to see Salt in action, by making Salt read the Salt state and apply it locally:

```
salt-call --local state.highstate -l debug
```

The first run will be very verbose, especially since we enabled the debug flag!

Now, let's run the command again:

```
salt-call --local state.highstate -l debug
```

This will also be pretty verbose, and the output will end with this:

```
local:
          ----------

ID: apache2 Function: pkg.installed
Result: True
Comment: Package apache2 is already installed. Started: 22:55:36.937634
Duration: 2267.167 ms
Changes:

Summary
          ------------

Succeeded: 1
Failed:     0
          ------------

Total states run:   1
```

Now, you can quit the container and restart it. This will clean the container from the Apache instance installed during the previous run.

This time, we will apply the same state but use the message queue method rather than applying the state locally:

```
salt-call state.highstate
```

This is the same command as used previously, except we omitted the –local flag. You could also try running the command again and verify that the state remains the same.

Salt versus Ansible versus Puppet versus PalletOps execution models

While the configuration systems we explore in this chapter share a fair number of similarities, they differ a lot in the way code is executed on the client nodes:

With Puppet, a Puppet agent registers with the Puppet master and opens a communication channel to retrieve commands. This process is repeated periodically, normally every thirty minutes.

Ansible pushes changes over SSH when desired. This is a push model.

Salt uses a push model, but with a different implementation. It employs a ZeroMQ messaging server that the clients connect to and listen for notifications about changes. This works a bit like Puppet, but faster.

Which method is best is an area of contention between developer communities. Proponents of the message queue architecture believe that it is faster and that speed matters. Proponents of the plain SSH method claim that it is fast enough and that simplicity matters. I lean toward the latter stance. Things tend to break, and the likelihood of breakage increases with complexity.

Vagrant

Vagrant is a configuration system for virtual machines. It is geared towards creating virtual machines for developers, but it can be used for other purposes as well.

Vagrant supports several virtualization providers, and VirtualBox is a popular provider for developers.

First, some preparation. Install vagrant according to the instructions for your distribution. For Fedora, the command is this:

```
yum install 'vagrant*'
```

This will install a number of packages. However, as we are installing this on Fedora, we will experience some problems. The Fedora Vagrant packages use libvirt as a virtual machine provider rather than VirtualBox. That is useful in many cases, but in this case, we would like to use VirtualBox as a provider, which requires some extra steps on Fedora. If you use some other distribution, the case might be different.

First, add the VirtualBox repository to your Fedora installation. Then we can install VirtualBox with the dnf command, as follows:

```
dnf install VirtualBox
```

VirtualBox is not quite ready to be used yet, though. It needs special kernel modules to work, since it needs to arbitrate access to low-level resources. The VirtualBox kernel driver is not distributed with the Linux kernel. Managing Linux kernel drivers outside of the Linux source tree has always been somewhat inconvenient compared to the ease of using kernel drivers that are always installed by default.

The VirtualBox kernel driver can be installed as a source module that needs to be compiled. This process can be automated to a degree with the dkms command, which will recompile the driver as needed when there is a new kernel installed. The other method, which is easier and less error-prone, is to use a kernel module compiled for your kernel by your distribution. If your distribution provides a kernel module, it should be loaded automatically. Otherwise, you could try modprobe vboxdrv. For some distributions, you can compile the driver by calling an init.d script as follows:

```
sudo /etc/init.d/vboxdrv setup
```

Now that the Vagrant dependencies are installed, we can bring up a Vagrant virtual machine.

The following command will create a Vagrant configuration file from a template. We will be able to change this file later. The base image will be hashicorp/precise32, which in turn is based on Ubuntu.

```
vagrant init hashicorp/precise32
```

Now, we can start the machine:

```
vagrant up
```

If all went well, we should have a vagrant virtual machine instance running now, but since it is headless, we won't see anything.

Vagrant shares some similarities with Docker. Docker uses base images that can be extended. Vagrant also allows this. In the Vagrant vocabulary, a base image is called a box.

To connect to the headless vagrant instance we started previously, we can use this command:

```
vagrant ssh
```

Now we have an ssh session, where we can work with the virtual machine. For this to work, Vagrant has taken care of a couple of tasks, such as setting up keys for the SSH communication channel for us.

Vagrant also provides a configuration system so that Vagrant machine descriptors can be used to recreate a virtual machine that is completely configured from source code.

Here is the Vagrantfile we got from the earlier stage. Comments are removed for brevity.

```
Vagrant.configure(2) do |config| config.vm.box = "hashicorp/precise32"
end
```

Add a line to the Vagrantfile that will call the bash script that we will provide:

```
Vagrant.configure("2") do |config| config.vm.box = "hashicorp/precise32"
config.vm.provision :shell, path: "bootstrap.sh" end
```

The bootstrap.sh script will look like this:

```
#!/usr/bin/env bash apt-get update
apt-get install -y apache2
```

This will install an Apache web server in the Vagrant-managed virtual machine.

Now we know enough about Vagrant to be able to reason about it from a DevOps perspective:

Vagrant is a convenient way of managing configurations primarily for virtual machines based on VirtualBox. It's great for testing.

The configuration method doesn't really scale up to clusters, and it's not the intended use case, either.

On the other hand, several configuration systems such as Ansible support Vagrant, so Vagrant can be very useful while testing our configuration code.

Deploying with Docker

A recent alternative for deployment is Docker, which has several very interesting traits. You can make use of Docker's features for test automation purposes even if you use, for instance, Puppet or Ansible to deploy your products.

Docker's model of creating reusable containers that can be used on development machines, testing environments, and production environments is very appealing.

At the time of writing, Docker is beginning to have an impact on larger enterprises, but solutions such as Puppet are dominant.

While it is well known how to build large Puppet or Ansible server farms, it's not yet equally well known how to build large Docker-based server clusters.

There are several emerging solutions, such as these:

Docker Swarm: Docker Swarm is compatible with Docker Compose, which is appealing. Docker Swarm is maintained by the Docker community.

Kubernetes: Kubernetes is modeled after Google's Borg cluster software, which is appealing since it's a well-tested model used in-house in Google's vast data centers. Kubernetes is not the same as Borg though, which must be kept in mind. It's not clear whether Kubernetes offers scaling the same way Borg does.

Comparison tables

Everyone likes coming up with new words for old concepts. While the different concepts in various products don't always match, it's tempting to make a dictionary that maps the configuration systems' different terminology with each other.

Here is such a terminology comparison chart:

System	Puppet	Ansible	Pallet	Salt
Client	Agent	Node	Node	Minion
Server	Master	Server	Server	Master
Configuration	Catalog	Playbook	Crate	Salt State

Also, here is a technology comparison chart:

System	Puppet	Ansible	Pallet	Chef	Salt
Agentless	No	Yes	Yes	Yes	Both
Client dependencies	Ruby	Python, sshd, bash	sshd, bash	Ruby, sshd, bash	Python
Language	Ruby	Python	Clojure	Ruby	Python

Cloud solutions

First, we must take a step back and have a look at the landscape. We can either use a cloud provider, such as AWS or Azure, or we can use our own internal cloud solution, such as VMware or OpenStack. There are valid arguments for both external and internal cloud providers or even both, depending on your organization.

Some types of organizations, such as government agencies, must store all data regarding citizens within their own walls. Such organizations can't use external cloud providers and services and must instead build their own internal cloud equivalents.

Smaller private organizations might benefit from using an external cloud provider but can't perhaps afford having all their resources with such a provider. They might opt to have in-house servers for normal loads and scale out to an external cloud provider during peak loads.

Many of the configuration systems we have described here support the management of cloud nodes as well as local nodes. PalletOps supports AWS and Puppet supports Azure, for instance. Ansible supports a host of different cloud services.

AWS

Amazon Web Services allows us to deploy virtual machine images on Amazon's clusters. You can also deploy Docker images. Follow these steps to set up AWS:

Sign up for an account with AWS. Registration is free of charge, but a credit card number is required even for the free of charge tier. Some identity verification will need to happen, which can be done via an automated challenge-response phone call. When the user verification process is complete, you will be able to log in to AWS and use the web console.

Go to EC2 network and security. Here you can create management keys that will be required later.

As a first example, let's create the default container example provided by AWS, console-sample-app-static. To log in to the generated server, you need first to create an SSH key pair and upload your public key to AWS. Click through all the steps and you will get a small sample cluster. The final resource creation step can be slow, so it's the perfect opportunity to grab a cup of coffee!

Now, we can view the details of the cluster and choose the web server container. You can see the IP address. Try opening it in a web browser.

Now that we have a working account on AWS, we can manage it with the configuration management system of our choosing.

Azure

Azure is a cloud platform from Microsoft. It can host both Linux and Microsoft virtual machines. While AWS is the service people often default to, at least in the Linux space, it never hurts to explore the options. Azure is one such option that is gaining market share at the moment.

Creating a virtual machine on Azure for evaluation purposes is comparable to creating a virtual machine on AWS. The process is fairly smooth