Unit 4 & 5: Web Development Framework

Ansible: Creating Role

Ansible roles take your automations to the next level of abstraction. They are ideal for sharing functionality between different teams, environments, or projects, making your code DRY and enabling streamlined management of your IaC. With roles, you get a standardized structure for bundling related tasks, variables, templates, handlers, and files, and this enhances reusability.

What are Ansible Roles?

Ansible Roles provide a well-defined framework and structure for setting your tasks, variables, handlers, metadata, templates, and other files. They enable us to reuse and share our Ansible code efficiently. This way, we can reference and call them in our playbooks with just a few lines of code while we can reuse the same roles over many projects without the need to duplicate our code.

Why Roles Are Useful in Ansible?

When starting with Ansible, it's pretty common to focus on writing playbooks to automate repeating tasks quickly. As new users automate more and more tasks with playbooks and their Ansible skills mature, they reach a point where using just Ansible playbooks is limiting. Ansible Roles to the rescue!

Since we have our code grouped and structured according to the Ansible standards, it is quite straightforward to share it with others. We will see an example of how we can accomplish that later with Ansible Galaxy. Organizing our Ansible content into roles provides us with a structure that is more manageable than just using playbooks. This might not be evident in minimal projects but as the number of playbooks grows, so does the complexity of our projects.

Lastly, placing our Ansible code into roles lets us organize our automation projects into logical groupings and follow the separation of concerns design principle. Collaboration and velocity are also improved since different users can work on separate roles in parallel without modifying the same playbooks simultaneously.

What is the difference between roles and playbook in Ansible?

A role is a set of tasks and additional files that are used to configure a host to serve a specific role. A playbook is a mapping of hosts and roles.

What is role and task in Ansible?

Roles provide a framework for completely self-contained or interdependent collections of variables, tasks, files, templates, and modules. The role is the primary mechanism in Ansible for dividing a playbook into multiple files. This simplifies the creation of complex playbooks and makes them more reusable.

How to write an Ansible role?

To create an Ansible role, use the ansible-galaxy command, which includes templates. This will create it in the default directory /etc/ansible/roles and make the necessary changes; otherwise, we must manually create each directory and file. where ansible-galaxy is the command used to create roles from templates.

Ansible Role Structure

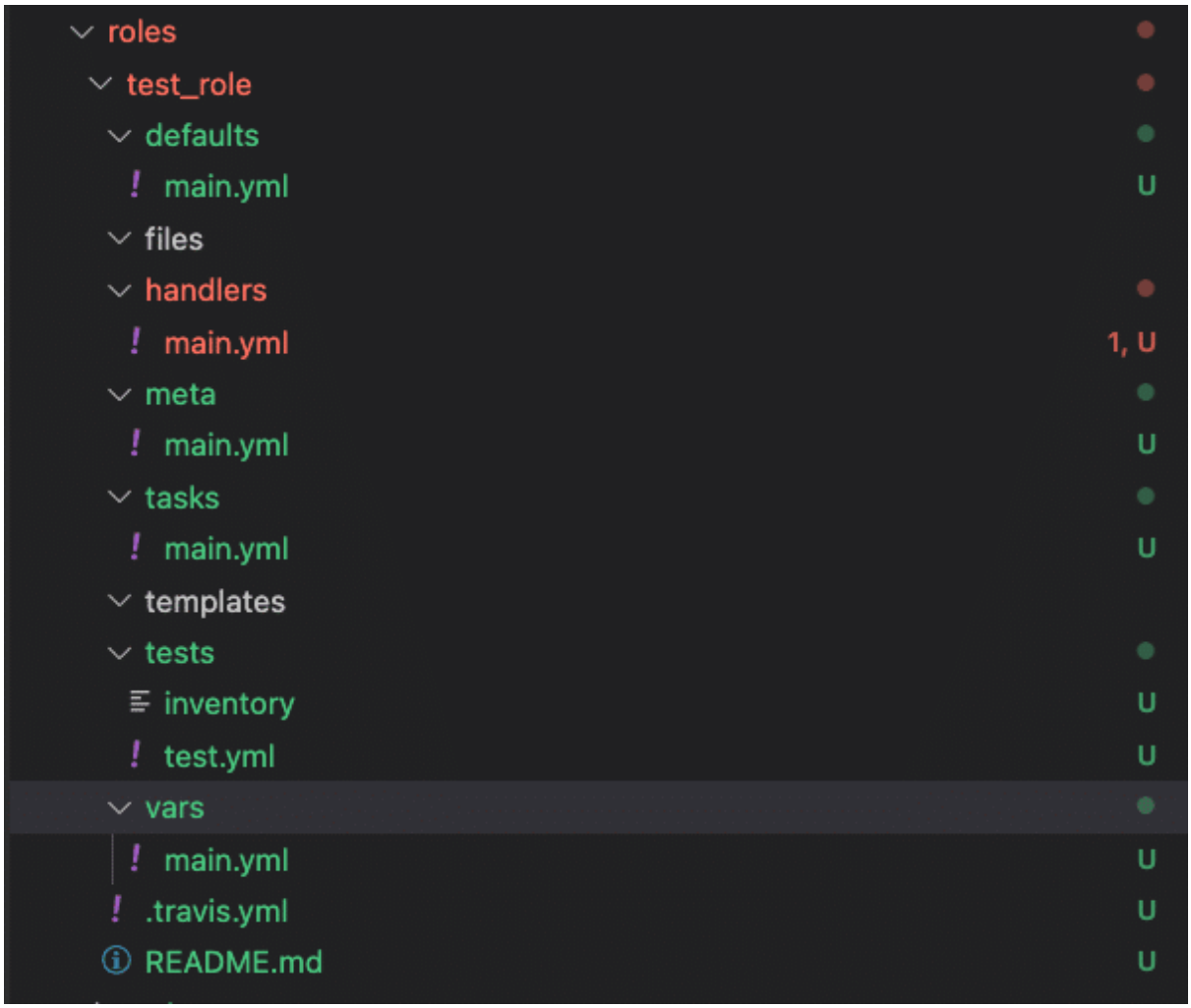
Let's have a look at the standard role directory structure. For each role, we define a directory with the same name. Inside, files are grouped into subdirectories according to their function. A role must include at least one of these standard directories and can omit any that isn't actively used.

To assist us with quickly creating a well-defined role directory structure skeleton, we can leverage the command ansible-galaxy init . The ansible-galaxy command comes bundled with Ansible, so there is no need to install extra packages.

Create a skeleton structure for a role named test_role:

```
→  ansible-roles git:(master) ✗ ansible-galaxy init test_role
- Role test_role was created successfully
```

This command generates this directory structure:



Ansible checks for main.yml files, possible variations, and relevant content in each subdirectory. It's possible to include additional YAML files in some directories. For instance, you can group your tasks in separate YAML files according to some characteristic.

- **defaults –** Includes default values for variables of the role. Here we define some sane default variables, but they have the lowest priority and are usually overridden by other methods to customize the role.
- **files –** Contains static and custom files that the role uses to perform various tasks.
- **handlers –** A set of handlers that are triggered by tasks of the role.
- **meta –** Includes metadata information for the role, its dependencies, the author, license, available platform, etc.
- **tasks –** A list of tasks to be executed by the role. This part could be considered similar to the task section of a playbook.
- **templates –** Contains Jinja2 template files used by tasks of the role. (Read more about how to create an Ansible template.)
- **tests –** Includes configuration files related to role testing.
- **vars –** Contains variables defined for the role. These have quite a high precedence in Ansible.

Another directory that wasn't automatically generated by the ansible-galaxy init command but is mentioned in the official Ansible docs, and you might find helpful in some cases, is the library directory. Inside it, we define any custom modules and plugins that we have written and used by the role.

Creating Ansible Roles

A common tactic is to refactor an Ansible playbook into a role. To achieve that, we have to decompose the different parts of a playbook and stitch them together into an Ansible role using the directories we've just seen in the previous section. Ansible searches for referenced roles in common paths like the orchestrating playbook's directory, the roles/ directory, or the configured roles_path configuration value. It's also possible to set a custom path when referencing a role:

```
- hosts: all
  roles:
    - role: "/custom_path/to/the/role"
```

Using the ansible-galaxy init command, we generate the initial directory structure for a role named webserver inside a parent directory named roles. Let's go ahead and delete the tests directory since we won't be using it. We will see how to utilize all the other directories during our demo.

For more detail:
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_reuse_roles.html

Ansible Variables

Variable in playbooks are very similar to using variables in any programming language. It helps you to use and assign a value to a variable and use that anywhere in the playbook. One can put conditions around the value of the variables and accordingly use them in the playbook.

Example:

```
- hosts : <your hosts>
vars:
tomcat_port : 8080
```

In the above example, we have defined a variable name tomcat_port and assigned the value 8080 to that variable and can use that in your playbook wherever needed.

Now taking a reference from the example shared. The following code is from one of the roles (install-tomcat) –

```
block:
   - name: Install Tomcat artifacts
      action: >
      yum name = "demo-tomcat-1" state = present
      register: Output
   always:
      - debug:
         msg:
            - "Install Tomcat artifacts task ended with message: {{Output}}"
            - "Installed Tomcat artifacts - {{Output·changed}}"
```

Why Variables Are Useful in Ansible?

The use of variables simplifies the management of dynamic values throughout an Ansible project and can potentially reduce the number of human errors. We have a convenient way to handle variations and differences between different environments and systems with variables.

Another advantage of variables in Ansible is that we have the flexibility to define them in multiple places with different precedence according to our use case. We can also register new variables in our playbooks by using the returned value of a task.

Ansible facts are a special type of variables that Ansible retrieves from any remote host for us to leverage them in Ansible projects. For example, we can get information regarding the operating system distribution with ansible_distribution, information about devices on the host, the python version that Ansible is using with

ansible_python_version, and the system architecture, among others. To access this data, we have to reference the ansible_facts variable.

Variable Name Rules

Ansible has a strict set of rules to create valid variable names. Variable names can contain only letters, numbers, and underscores and must start with a letter or underscore. Some strings are reserved for other purposes and aren't valid variable names, such as Python Keywords or Playbook Keywords.

Defining and Referencing Simple Variables

The simplest use case of variables is to define a variable name with a single value using standard YAML syntax. Although this pattern can be used in many places, we will show an example in a playbook for simplicity.

```
- name: Example Simple Variable
  hosts: all
  become: yes
  vars:
    username: bob

  tasks:
  - name: Add the user {{ username }}
    ansible.builtin.user:
      name: "{{ username }}"
      state: present
```

In the above example, after the vars block, we define the variable username, and assign the value bob. Later, to reference the value in the task, we use Jinja2 syntax like this "{{ username }}"

If a variable's value starts with curly braces, we must quote the whole expression to allow YAML to interpret the syntax correctly.

For more details: https://spacelift.io/blog/ansible-variables & https://docs.ansible.com/archive/ansible/2.4/playbooks_variables.html

Exception Handling in Playbooks

Exception handling in Ansible is similar to exception handling in any programming language. An example of the exception handling in playbook is shown below.

```
tasks:
   - name: Name of the task to be executed
      block:
         - debug: msg = 'Just a debug message , relevant for logging'
         - command: <the command to execute>
      rescue:
         - debug: msg = 'There was an exception·· '
         - command: <Rescue mechanism for the above exception occurred)
      always:
         - debug: msg = "this will execute in all scenarios· Always will get logged"
```

Following is the syntax for exception handling:

- **rescue** and **always** are the keywords specific to exception handling.
- Block is where the code is written (anything to be executed on the Unix machine).
- If the command written inside the block feature fails, then the execution reaches rescue block and it gets executed. In case there is no error in the command under block feature, then rescue will not be executed.
- **Always** gets executed in all cases.
- So if we compare the same with java, then it is similar to try, catch and finally block.
- Here, **Block** is similar to **try block** where you write the code to be executed and **rescue** is similar to **catch block** and **always** is similar to **finally**.

For more details: -
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_error_handling.html

Loops

Below is the example to demonstrate the usage of Loops in Ansible. The tasks is to copy the set of all the war files from one directory to tomcat webapps folder. Most of the commands used in the example below are already covered before. Here, we will concentrate on the usage of loops. Initially in the 'shell' command we have done ls *.war. So, it will list all the war files in the directory.

Output of that command is taken in a variable named output. To loop, the **'with_items'** syntax is being used.

with_items: "{{output.stdout_lines}}" --> output.stdout_lines gives us the line by line output and then we loop on the output with the with_items command of Ansible.

Attaching the example output just to make one understand how we used the stdout_lines in the with_items command.

```
---
#Tsting
- hosts: tomcat-node
   tasks:
      - name: Install Apache
      shell: "ls *.war"
      register: output
      args:
         chdir: /opt/ansible/tomcat/demo/webapps
      - file:
         src: '/opt/ansible/tomcat/demo/webapps/{{ item }}'
         dest: '/users/demo/abc/{{ item }}'
         state: link
      with_items: "{{output.stdout_lines}}"
```

For more details:
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_loops.html

Blocks

Blocks create logical groups of tasks. Blocks also offer ways to handle task errors, similar to exception handling in many programming languages. The playbook in totality is broken into blocks. The smallest piece of steps to execute is written in block. Writing the specific instruction in blocks helps to segregate functionality and handle it with exception handling if needed.

Example of blocks is covered in variable usage,exception handling and loops above.

For more details:
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_blocks.html

Conditionals

Conditionals are used where one needs to run a specific step based on a condition. In a playbook, you may want to execute different tasks or have different goals, depending on the value of a fact (data about the remote system), a variable, or the result of a previous task. You may want the value of some variables to depend on the value of other variables. Or you may want to create additional groups of hosts based on whether the hosts match other criteria. You can do all of these things with conditionals.

```
---
#Tsting
- hosts: all
   vars:
      test1: "Hello, Hey!"
   tasks:
      - name: Testing Ansible variable
      debug:
         msg: "Equals"
         when: test1 == "Hello, Hey!"
```

In this case, Equals will be printed as the test1 variable is equal as mentioned in the when condition. when can be used with a logical OR and logical AND condition as in all the programming languages.

For more details:
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_conditionals.html

Jenkins

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.

Jenkins is a Java-based open-source automation platform with plugins designed for continuous integration. It is used to continually create and test software projects, making it easier for developers and DevOps engineers to integrate changes to the project and for consumers to get a new build.

Organizations may use Jenkins to automate and speed up the software development process. Jenkins incorporates a variety of development life-cycle operations, such as build, document, test, package, stage, deploy, static analysis, and more. Jenkins is a well-known continuous integration tool developed initially by Hudson before it became available on the open-source market. Hudson was created by Kohsuke Kawaguchi in 2004 while working at Sun Microsystems (acquired by Oracle). There was a disagreement between Oracle and the Hudson community about the infrastructure employed after Oracle purchased Sun Microsystems in 2010.

In 2011, the Hudson community unanimously accepted a referendum to alter the project name from Hudson to Jenkins, resulting in the creation of the first “Jenkins” project. Hudson was later donated to the Eclipse Foundation and is no longer being worked on. Jenkins development is currently administered as an open-source project under the direction of the CD Foundation, a Linux Foundation initiative. Jenkins is a widely used program with over 300,000 installations worldwide and rising daily. Software firms may speed up their software development process by adopting Jenkins, which can automate test and build at a high pace. It’s a server-based application that necessitates using a web server such as Apache Tomcat.

Why are we using Jenkins?

Jenkins is used to build and test your product continuously, so developers can continuously integrate changes into the build. Jenkins is the most popular open source CI/CD tool on the market today and is used in support of DevOps, alongside other cloud native tools. Jenkins is used to building software projects and test them continuously, allowing developers to integrate any changes in their projects easily. It also allows you to consistently deliver your software since it's integrated with so many testing and deployment technologies.

Jenkins Core Concepts

- **Jenkins Controller (Formerly Master):** The Jenkins architecture supports distributed builds. One Jenkins node functions as the organizer, called a Jenkins Controller. This node manages other nodes running the Jenkins Agent. It can also execute builds, although it isn’t as scalable as Jenkins agents. The controller holds the central Jenkins configuration. It manages agents and their connections, loads plugins, and coordinates project flow.

- **Jenkins Agent (Formerly Slave):** The Jenkins Agent connects to the Jenkins Controller to run build jobs. To run it, you'll need to install Java on a physical machine, virtual machine, cloud compute instance, Docker image, or Kubernetes cluster. You can use multiple Jenkins Agents to balance build load, improve performance, and create a secure environment independent of the Controller.
- **Jenkins Node:** A Jenkins node is an umbrella term for Agents and Controllers, regardless of their actual roles. A node is a machine on which you can build projects and pipelines. Jenkins automatically monitors the health of all connected nodes, and if metrics go below a threshold, it takes the node offline.
- **Jenkins Project (Formerly Job):** A Jenkins project or task is an automated process created by a Jenkins user. The plain Jenkins distribution offers a variety of build tasks that can support continuous integration workflows, and more are available through a large ecosystem of plugins.
- **Jenkins Plugins:** Plugins are community-developed modules you can install on a Jenkins server. This adds features that Jenkins doesn't have by default. You can install/upgrade all available plugins from the Jenkins dashboard.
- **Jenkins Pipeline:** A Jenkins Pipeline is a user-created pipeline model. The pipeline includes a variety of plugins that help you define step-by-step actions in your software pipeline. This includes:
    - Automated builds.
    - Multi-step testing.
    - Deployment procedures.
    - Security scanning

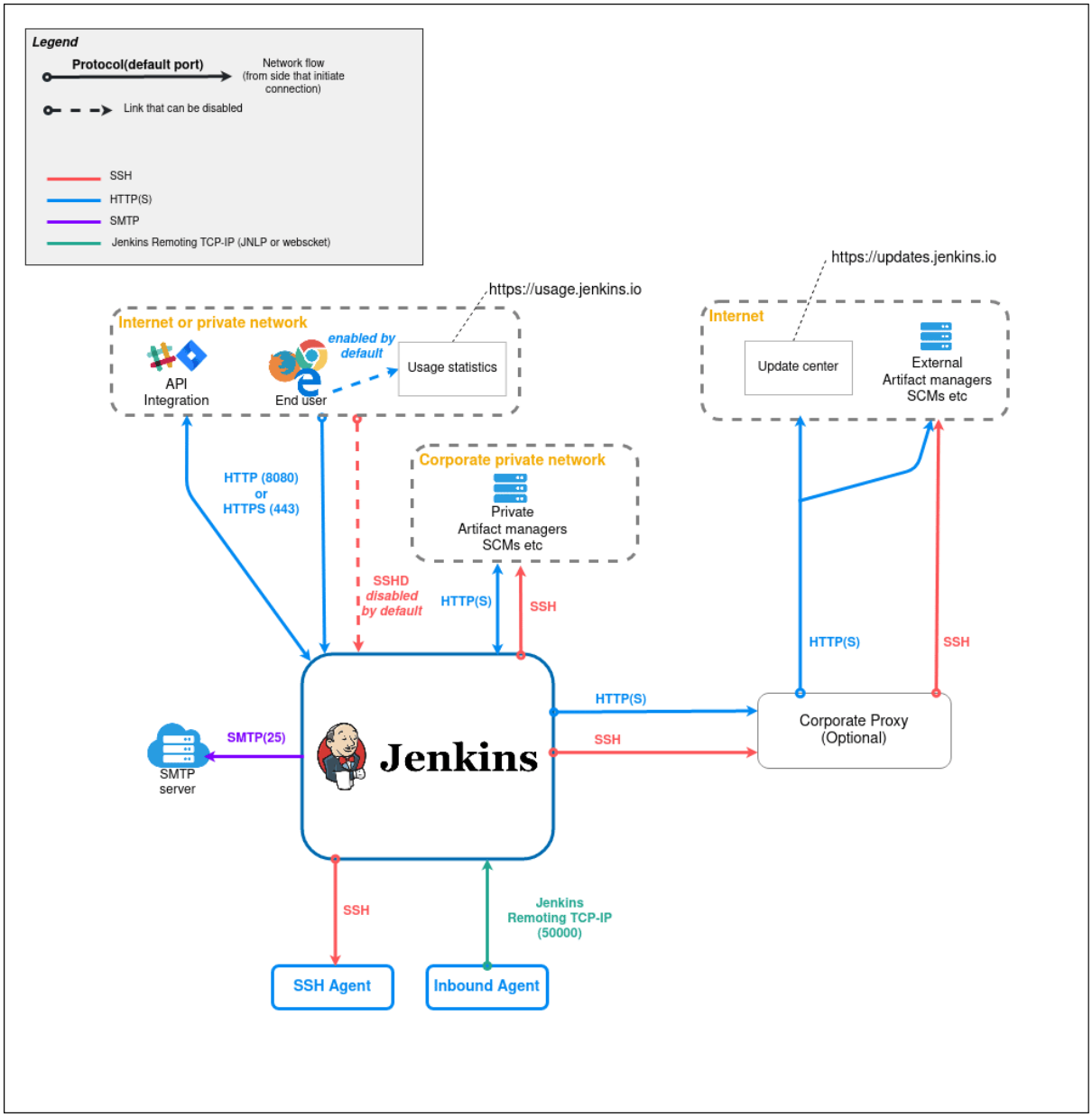
You can create pipelines directly in the user interface, or create a "Jenkinsfile" which represents a pipeline as code. Jenkinsfiles use a Groovy-compatible text-based format to define pipeline processes, and can be either declarative or scripted.

Jenkins Architecture

Jenkins elements interact and work together in the following way:

- Developers modify the source code, committing changes to the repository, and Jenkins creates a new build in order to handle the new Git commit.
- Jenkins can work in "push" or "pull" mode. The Jenkins CI server is either triggered by an event such as a code commit, or it can regularly check the repository for changes.
- The build server builds the code and generates an artifact. If the build fails, the developer receives an alert.
- Jenkins deploys the built application/executable to the test server, which can execute continuous, automated tests. Developers receive alerts if their changes impact functionality.
- Jenkins optionally deploys the changes to the production server if the code has no issues.

The following diagram illustrates the Jenkins architecture:



You may need more than one Jenkins server to test code in different environments. A single server cannot always handle the load for large projects. If this is the case, you can use the distributed Jenkins architecture to implement continuous integration and testing. The Jenkins server can access the Controller environment, which distributes the workload across different Jenkins Agents. This distributed model lets you run different builds and tests simultaneously, with each Jenkins Agent environment running a different version of the code and the Controller controlling the operations.

Jenkins Advantages and Disadvantages

Here are some of the key advantages of Jenkins:

- Highly extensible with a huge variety of existing plugins. Plugins contribute to Jenkins' flexibility and rich scripting and declarative language which supports advanced, custom pipelines.
- Robust and reliable at almost any scale.

- Mature and battle-tested.
- Supports hybrid and multi-cloud environments.
- Offers an extensive knowledge base, documentation, and community resources.
- Based on Java, an enterprise development language with a broad ecosystem, making it suitable for legacy enterprise environments.

Here are some disadvantages of Jenkins:

- **Single server architecture—**uses a single server architecture, which limits resources to resources on a single computer, virtual machine, or container. Jenkins doesn't allow server-to-server federation, which can cause performance issues in large-scale environments.
- **Jenkins sprawl—**this is a common problem which also stems from lack of federation. Multiple teams using Jenkins can create a large number of standalone Jenkins servers that are difficult to manage.
- **Relies on dated Java architectures and technologies—**specifically Servlet and Maven. In general, Jenkins uses a monolithic architecture and is not designed for newer Java technologies such as Spring Boot or GraalVM.
- **Not container native—**Jenkins was designed in an era before containers and Kubernetes gained popularity, and while it supports container technology, it does not have nuanced support for container and orchestration mechanisms.
- **Difficult to implement in production environments—**developing continuous delivery pipelines with Jenkinsfiles requires coding in a declarative or scripting language, and complex pipelines can be difficult to code, debug, and maintain.
- **Offers no functionality for real production deployments—**"deploying with Jenkins" means running a fully customized set of scripts to handle the deployment.
- **Jenkins itself requires deployment—**this can be difficult to automate. Organizations that need to combine Jenkins with a continuous delivery solution have traditionally used configuration management to do this, but this adds another layer of complexity and is error-prone.
- **Complicated plugin management—**Jenkins has nearly 2,000 plugins, which can be overwhelming to sort through until you find a useful plugin. Many plugins also have dependencies that increase the management burden, while some plugins may conflict with each other. There is no guarantee a plugin you use will continue to be maintained.
- **Groovy expertise requirements—**Jenkins has programmatic pipelines implemented in Groovy, a language that is currently not in wide use and can make scripts difficult to work with. Jenkins supports scripted and declarative Groovy modes.

Installation of Jenkins

Jenkins is typically run as a standalone application in its own process. The Jenkins WAR file bundles Winstone, a Jetty servlet container wrapper, and can be started on any operating system or platform with a version of Java supported by Jenkins.

Theoretically, Jenkins can also be run as a servlet in a traditional servlet container like Apache Tomcat or WildFly, but in practice this is largely untested and there are many caveats. In particular, support for WebSocket agents is only implemented for the Jetty servlet container.

For more details: https://www.jenkins.io/doc/book/installing/

CI/CD(Continuous Integration/Continuous Delivery)

Automate your software development workflows and deploy better quality code, more often. Using a continuous and iterative process to build, test, and deploy helps avoid bugs and code failures.

CI/CD falls under DevOps (the joining of development and operations teams) and combines the practices of continuous integration and continuous delivery. CI/CD automates much or all of the manual human intervention traditionally needed to get new code from a commit into production, encompassing the build, test (including integration tests, unit tests, and regression tests), and deploy phases, as well as infrastructure provisioning. With a CI/CD pipeline, development teams can make changes to code that are then automatically tested and pushed out for delivery and deployment. Get CI/CD right and downtime is minimized and code releases happen faster.

CI/CD is an essential part of DevOps and any modern software development practice. A purpose-built CI/CD platform can maximize development time by improving an organization's productivity, increasing efficiency, and streamlining workflows through built-in automation, testing, and collaboration. As applications grow larger, the features of CI/CD can help decrease development complexity. Adopting other DevOps practices — like shifting left on security and creating tighter feedback loops — helps organizations break down development silos, scale safely, and get the most out of CI/CD.

CI/CD is important because it helps development, security, and operations teams work as efficiently and effectively as possible. It decreases tedious and time-consuming manual development work and legacy approval processes, freeing DevOps teams to be more innovative in their software development. Automation makes processes predictable and repeatable so that there is less opportunity for error from human intervention. DevOps teams gain faster feedback and can integrate smaller changes frequently to reduce the risk of build-breaking changes. Making DevOps processes continuous and iterative speeds software development lifecycles so organizations can ship more features that customers love.

What is continuous integration (CI)?

Continuous integration is the practice of integrating all your code changes into the main branch of a shared source code repository early and often, automatically testing each change when you commit or merge them, and automatically kicking off a build. With continuous integration, errors and security issues can be identified and fixed more easily, and much earlier in the development process. By merging changes frequently and triggering automatic testing and validation processes, you minimize the

possibility of code conflict, even with multiple developers working on the same application. A secondary advantage is that you don't have to wait long for answers and can, if necessary, fix bugs and security issues while the topic is still fresh in your mind.

Common code validation processes start with a static code analysis that verifies the quality of the code. Once the code passes the static tests, automated CI routines package and compile the code for further automated testing. CI processes should have a version control system that tracks changes so you know the version of the code used.

What is continuous delivery (CD)?

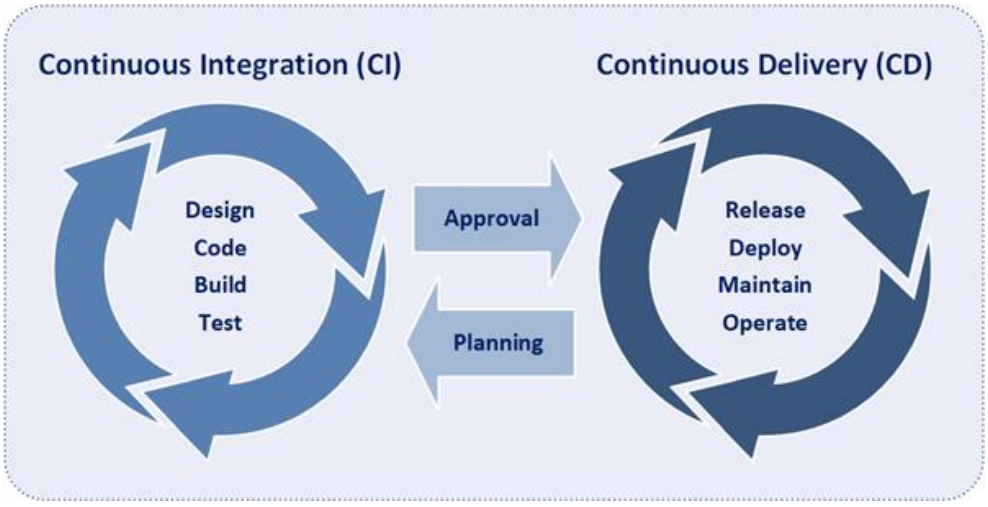
Continuous delivery is a software development practice that works in conjunction with CI to automate the infrastructure provisioning and application release process. Once code has been tested and built as part of the CI process, CD takes over during the final stages to ensure it's packaged with everything it needs to deploy to any environment at any time. CD can cover everything from provisioning the infrastructure to deploying the application to the testing or production environment.

With CD, the software is built so that it can be deployed to production at any time. Then you can trigger the deployments manually or move to continuous deployment, where deployments are automated as well.

What is continuous deployment?

Continuous deployment enables organizations to deploy their applications automatically, eliminating the need for human intervention. With continuous deployment, DevOps teams set the criteria for code releases ahead of time and when those criteria are met and validated, the code is deployed into the production environment. This allows organizations to be more nimble and get new features into the hands of users faster.

While you can do continuous integration without continuous delivery or deployment, you can't really do CD without already having CI in place. That's because it would be extremely difficult to be able to deploy to production at any time if you aren't practicing CI fundamentals like integrating code to a shared repo, automating testing and builds, and doing it all in small batches on a daily basis.



- ***By Tushar Sir***

CI/CD fundamentals

There are eight fundamental elements of CI/CD that help ensure maximum efficiency for your development lifecycle. They span development and deployment. Include these fundamentals in your pipeline to improve your DevOps workflow and software delivery:

1. **A single source repository:** Source code management (SCM) that houses all necessary files and scripts to create builds is critical. The repository should contain everything needed for the build. This includes source code, database structure, libraries, properties files, and version control. It should also contain test scripts and scripts to build applications.
2. **Frequent check-ins to main branch:** Integrate code in your trunk, mainline or master branch — i.e., trunk-based development — early and often. Avoid sub-branches and work with the main branch only. Use small segments of code and merge them into the branch as frequently as possible. Don't merge more than one change at a time.
3. **Automated builds:** Scripts should include everything you need to build from a single command. This includes web server files, database scripts, and application software. The CI processes should automatically package and compile the code into a usable application.
4. **Self-testing builds:** CI/CD requires continuous testing. Testing scripts should ensure that the failure of a test results in a failed build. Use static pre-build testing scripts to check code for integrity, quality, and security compliance. Only allow code that passes static tests into the build.
5. **Frequent iterations:** Multiple commits to the repository results in fewer places for conflicts to hide. Make small, frequent iterations rather than major changes. By doing this, it's possible to roll changes back easily if there's a problem or conflict.
6. **Stable testing environments:** Code should be tested in a cloned version of the production environment. You can't test new code in the live production version. Create a cloned environment that's as close as possible to the real environment. Use rigorous testing scripts to detect and identify bugs that slipped through the initial pre-build testing process.
7. **Maximum visibility:** Every developer should be able to access the latest executables and see any changes made to the repository. Information in the repository should be visible to all. Use version control to manage handoffs so developers know which is the latest version. Maximum visibility means everyone can monitor progress and identify potential concerns.
8. **Predictable deployments anytime:** Deployments should be so routine and low-risk that the team is comfortable doing them anytime. CI/CD testing and verification processes should be rigorous and reliable, giving the team confidence to deploy updates at any time. Frequent deployments incorporating limited changes also pose lower risks and can be easily rolled back.

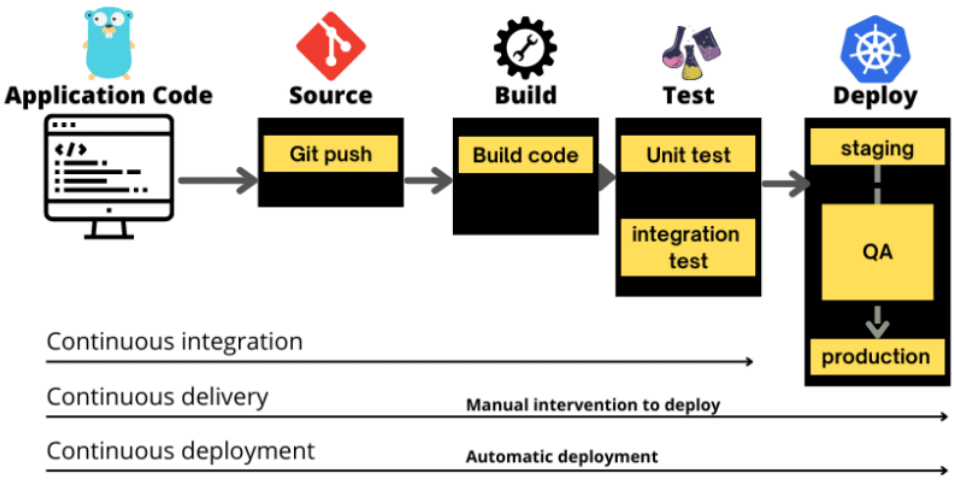
CI/CD Pipeline

A CI/CD pipeline is a series of steps that automates the process of delivering a new version of software. The pipeline combines continuous integration (CI) and continuous deployment (CD):

- **CI:** Builds code and runs tests
- **CD:** Deploys a new version of the application

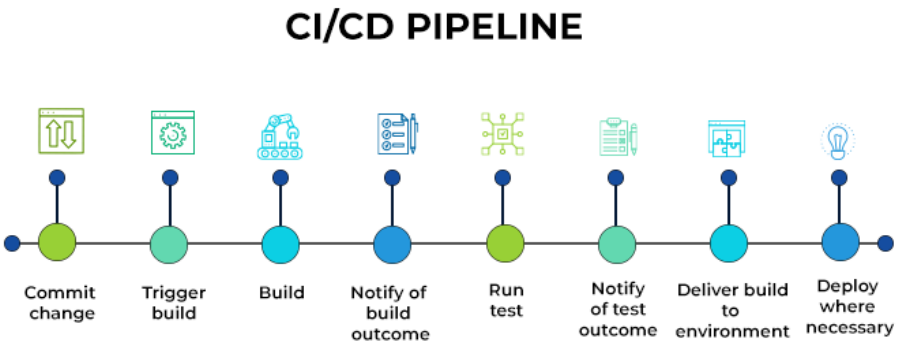
CI/CD pipelines can:

- Remove manual errors
- Provide standardized feedback loops to developers
- Enable fast product iterations
- Minimize manual errors
- Enhance feedback loops
- Allow teams to deliver smaller chunks of releases within a shorter time



CI/CD pipelines are often written in a markup language, with YAML being the most common language. Some CI/CD tools include:

- **Bitbucket Pipelines:** Integrates directly into Bitbucket, a cloud-based source control system
- **Travis CI:** A cloud-based CI/CD tool that supports a wide range of programming languages
- **CircleCI:** A cloud-based CI/CD tool



- ***By Tushar Sir***

The benefits of CI/CD implementation

Companies and organizations that adopt CI/CD tend to notice a lot of positive changes. Here are some of the benefits you can look forward to as you implement CI/CD:

- **Happier users and customers:** Fewer bugs and errors make it into production, so your users and customers have a better experience. This leads to improved levels of customer satisfaction, improved customer confidence, and a better reputation for your organization.
- **Accelerated time-to-value:** When you can deploy any time, you can bring products and new features to market faster. Your development costs are lower, and a faster turnaround frees your team for other work. Customers get results faster, giving your company a competitive edge.
- **Less fire fighting:** Testing code more often, in smaller batches, and earlier in the development cycle can seriously cut down on fire drills. This results in a smoother development cycle and less team stress. Results are more predictable, and it's easier to find and fix bugs.
- **Hit dates more reliably:** Removing deployment bottlenecks and making deployments predictable can remove a lot of the uncertainty around hitting key dates. Breaking work into smaller, manageable bites means it's easier to complete each stage on time and track progress. This approach gives plenty of time to monitor overall progress and determine completion dates more accurately.
- **Free up developers' time:** With more of the deployment process automated, the team has time for more rewarding projects. It's estimated that developers spend between 35% and 50% of their time testing, validating, and debugging code. By automating these processes, developers significantly improve their productivity.
- **Less context switching:** Getting real-time feedback on their code makes it easier for developers to work on one thing at a time and minimizes their cognitive load. By working with small sections of code that are automatically tested, developers can debug code quickly while their minds are still fresh from programming. Finding bugs is easier because there's less code to review.
- **Reduce burnout:** Research shows that CD measurably reduces deployment pain and team burnout. Developers experience less frustration and strain when working with CI/CD processes. This leads to happier and healthier employees and less burnout.
- **Recover faster:** CI/CD makes it easier to fix issues and recover from incidents, reducing mean time to resolution (MTTR). Continuous deployment practices mean frequent small software updates so when bugs appear, it's easier to pin them down. Developers have the option of fixing bugs quickly or rolling back the change so that the customer can get back to work quickly.

Concepts of Pipeline Security

While automation and team-level collaboration are two of the most crucial benefits of adopting a CI/CD practice, administering security in such frameworks is often more

complex than doing so on legacy models. CI/CD security is the ideology of bringing security into DevOps, securing the pipeline, and streamlining a workflow that meets compliance standards. This is in line with the “shift-left” testing philosophy. In other words, it’s better to detect and prevent security issues as early in the development process as possible.

But why is this important? Most of the time, when an application reaches the production stage, it has already been through multiple rounds of development, testing, and refinement. By the time it gets to prod, the code is pretty stable. So if a security issue does arise at this stage, it’s usually because of a misconfiguration or an external threat.

With CI/CD security in place, on the other hand, any potential security vulnerabilities can be detected and corrected much earlier in the development process – before they have a chance to cause any damage. DevOps pipelines tend to introduce code repositories and container image registries that can be private or public. This can bring in misconfigurations or additional open-source vulnerabilities. CI/CD security is the best way to tackle these issues at the root.

Common Security CI/CD Pipeline Threats

Poisoned Pipeline Execution (PPE)

Poisoned pipeline execution (PPE) is a type of software vulnerability that can be exploited by attackers to inject malicious code into a running process. The vulnerability can be exploited to take control of the process or to cause the process to crash. PPE vulnerabilities are often found in both client-side and server-side applications that accept untrusted input without appropriate data sanitation. An effective approach to prevent PPE vulnerabilities is to avoid using untrusted input in server-side applications and to avoid using untrusted output (generated from the server) in client-side applications. As a recommended practice, all untrusted input should be carefully validated to ensure that it does not contain any malicious code. As for untrusted outputs, the data should be displayed as untrusted and should be used with caution. Although poisoned pipeline execution vulnerabilities can be difficult to exploit, they pose severe consequences if exploited successfully.

Insufficient Pipeline-Based Access Controls

When distributed teams develop or make changes to one part of the module, unified access unintentionally changes the state of other dependent sections. In such instances, setting unified access to pipelines is not recommended and can become daunting very quickly as CI/CD is all about collaboration from diverse teams. Insufficient access controls can be detrimental to security as they can lead to privilege escalation and data leakage. An attacker with low privileges may be able to gain high-privileged access or modify code that is not supposed to be accessible. Also, if there are no access controls in place, an attacker may be able to view or tamper with data that is not intended for them.

Ungoverned Use of Third-Party Services

A common CI/CD security threat arises through the ungoverned use of third-party services. Although this offers developers the freedom to choose the services they intend to use, this can introduce a number of security issues, as the security risks associated with using these services are either unknown or ignored.

For example, a developer may choose to use a public code repository and container image registry that is not owned or controlled by their company. This code repository may not have the same security controls in place as the company's internal code repositories. As a result, the code that is stored in this public repository may be less secure and more vulnerable to attack.

To help mitigate these risks, companies should adopt policies and procedures governing the use of third-party services. Enforcing such policies should ensure that only approved services that satisfy the organization's security controls are used and are updated regularly to tackle the changing threat landscape.

Dependency Chain Abuse: Another common CI/CD security issue arises from the abuse of dependency chains, which represent a series of dependencies between components of an application stack.

For example, a web application may depend on a number of other components, such as a database, an application server, and a web server. Attackers may exploit a vulnerability in the database that allows them to execute arbitrary code on the database server. They can then use this code execution to gain access to the web application and the application server. This type of attack is known as a "dependency chain attack".

To help mitigate dependency-based risks, organizations should carefully consider and manifest dependencies between components in their systems. They should also put in place policies and procedures to ensure that these dependencies are maintained and monitored to prevent vulnerabilities.

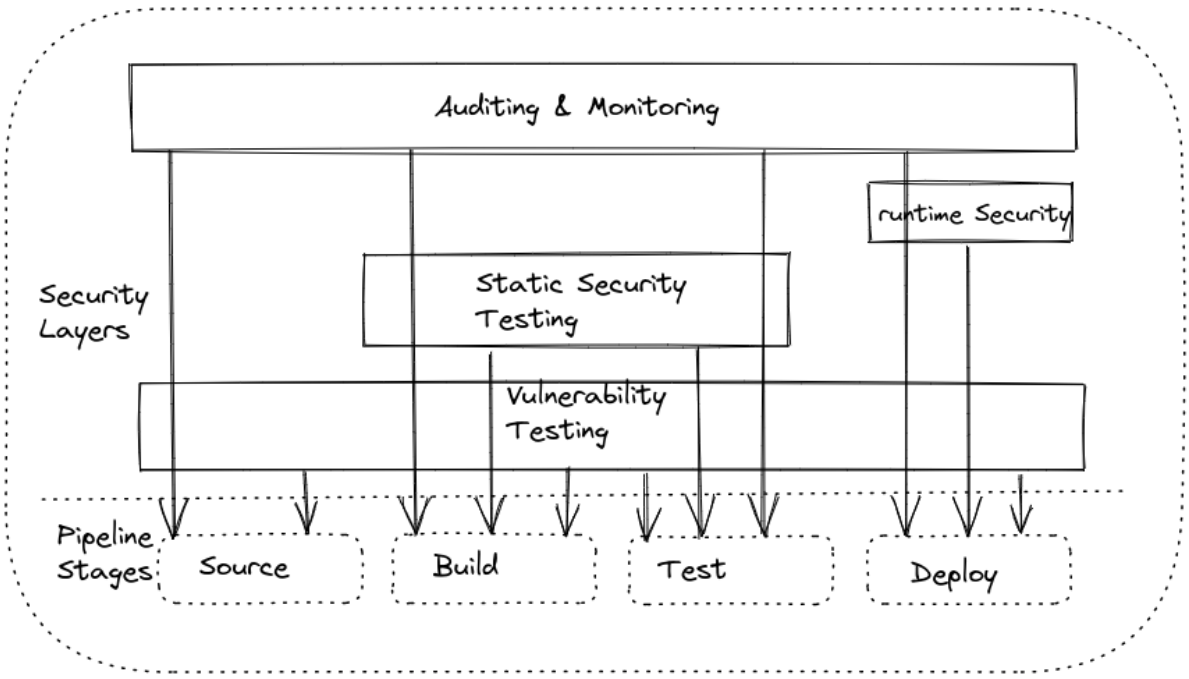
Best Practices for CI/CD Security

- **Embedding Security Gates:** Security gates can be embedded deeply in both CI and CD pipelines. Their purpose is to detect security issues early and prevent them before they reach production, such as documented vulnerabilities, common misconfigurations, permissions, and others. CI/CD security gates include not only the technical controls that need to be applied but also the way we think about security posture, risk appetite, and how to continuously enforce cloud-native security in an ongoing manner.
- **Avoid hardcoding secrets:** Ad-hoc development and testing with hardcoded values is attractive and gets the job done on the go, but placing keys and tokens in the code can trigger a security compliance issue. Avoiding this at all costs is ideal. The recommended practice is to use environment variables, which are variables that are set outside of the code and can be used by the code at runtime. This way, secrets are not hardcoded in the code and can be easily changed without having to modify the code itself.

- **Isolate configurations from dev, production, and non-production:** Configurations sometimes get executed irrespective of the source and permissions, producing unclear error stack traces and making debugging a nightmare. The recommended approach is to place config variables within different files using use-case-specific naming conventions to avoid conflicts and crashes in both non-prod and prod environments.
- **Implement explicit and private/protected variables:** As global and implicit values in the code complicate refactoring and debugging, a recommended practice is to use explicit, private, and immutable variables. The practice increases the stateful and stateless guarantees with an added advantage in developing performant and secure code.
- **Restrict pipeline access with dedicated policies:** As most CI/CD pipelines are deployed and reside in the cloud, interacting with external services and platforms is typically achieved via REST API. Allowing infrastructure resources to open public connections and access to all user groups can however lead to data exposure and GDPR compliance. To prevent this, it is recommended to restrict security policies and resource access to limited groups for enhanced security and trust.
- **Implement logging and tracking:** It is crucial to keep a tab on what went wrong and which part of the workflow is susceptible to attack vectors. Setting up logging at different levels and declaring paths for log collection during the initial stages of development helps keep a track of the execution of pipeline stages.

The Different Layers of CI/CD Security

CI/CD security comprises multiple comprehensive and multi-layered approaches to toughen pipeline security at every stage. The layers are listed as follows:



- **Runtime Security:** Bugs and misconfigurations are easy to bypass into production. The runtime security layer plays a vital role in scanning and detecting security leaks and threats from live applications running in production through a monitoring standpoint.
- **Static Security Testing:** Before the deployment stage, the code developed by internal teams has to undergo pre-defined tests for identifying potential vulnerabilities. The static security testing layer is about creating test cases in version control systems for detecting vulnerabilities before merging the features into the main branch.
- **Vulnerability Scanning:** Extend test cases of automated pipelines by adding application and environment scans to detect and classify weak code, infrastructural gaps, and third-party service resource leaking. A comprehensive vulnerability scanning also helps maintain a central repository with known vulnerabilities. Scanning the pipeline helps detect and compare classified issues with the scanned changes in the merged repo for remediating vulnerabilities.
- **Misconfiguration Scanning:** One of the most important things you can do is keep your systems up-to-date and free from any vulnerabilities. This process is known as misconfiguration scanning, and it involves regularly scanning your system for any potential security issues. One of the most common ways that hackers gain access to systems is through vulnerabilities that exist due to poor configuration. By regularly scanning your system, you can ensure that any potential issues are found and fixed before they can be exploited by criminals. In addition to keeping your system secure, misconfiguration scanning can also help improve its performance. By ensuring that all of your system's components are properly configured, you can help eliminate any bottlenecks or other issues that can slow down your system.
- **Auditing and Monitoring:** Logging phases of pipeline execution and tracking the success and failure patterns can be a resource in analyzing defect patterns and vulnerability detection for the overall CI/CD pipeline's security enhancement. Periodic auditing can help tackle evolving threats because it gives an overview of the system's health and resource utilization. The security team can use this data to further fine-tune detection mechanisms and response processes.

Building CI/CD Pipeline with Jenkins

To build a CI/CD pipeline with Jenkins, you can follow these tutorial:

https://towardsdatascience.com/create-your-first-ci-cd-pipeline-with-jenkins-and-github-6aefe21c9240