**2nd Slide**

**WHY SHOULD WE CREATE CUSTOM PLUGINS**

**Develop plugins to extend and enhance Jenkins features (building custom functionality for Jenkins):**

It Increases Efficiency and Productivity. Creating custom plugins will automate tasks and streamline workflows.

**Customize plugin according to your needs:** Custom plugins allow you to tailor Jenkins to meet the specific needs of your organization or project.

You can implement features that are not available in the default Jenkins distribution, ensuring a perfect fit for your requirements.

**Extend Jenkins Capabilities (Having control over your plugin):**

Jenkins provides a robust set of core features, but custom plugins enable you to extend its capabilities.

You can integrate with additional tools, technologies, or services that your organization uses.

**Promote Reusability:**

Share and reuse custom functionalities across projects by packaging them into plugins.

Encourage best practices and consistency by leveraging the same set of plugins across different Jenkins instances.

**Enhanced Reporting and Monitoring:**

Create plugins to generate custom reports or integrate with reporting tools to provide detailed insights into your builds.

Implement monitoring solutions that fit your specific needs, improving visibility into the health of your builds.

**Community Contribution:**

Contribute to the Jenkins community by sharing your custom plugins with others.

Foster collaboration and benefit from feedback and improvements from the wider Jenkins user community.

**3rd Slide**

Jenkins plugins follow a modular and extensible architecture that allows developers to enhance and customize Jenkins' functionality. Here's an overview of the key components and concepts in the architecture of Jenkins plugins:

### **1. Extension Points:**

* Extension points define areas where plugins can contribute additional functionality or modify existing behaviour. Extension points act as interfaces or hooks that plugins can implement or extend. Developers use extension points to customize and extend Jenkins' capabilities. Here's a brief explanation of Jenkins extension points:
* Jenkins plugins extend the core functionality through extension points.
* Extension points define interfaces or abstract classes that plugins can implement or extend.
* Examples of extension points include SCM (Source Code Management), Builders, Publishers, and more.

### **2. Extension Classes:**

* Plugins provide implementations for extension points by creating extension classes.
* These classes are annotated with **@Extension** to indicate that they contribute to a specific extension point.
* Extension classes define the behaviour or contribution of the plugin.
* **hudson.tasks.Builder**: Extension point for defining build steps in a Jenkins job.

Extension Point:

Extension points serve as entry points or hooks for plugins to contribute additional functionality or modify existing behaviour in Jenkins.

Extension Class:

An extension class is a concrete implementation of an extension point.

It is a Java class that either implements an extension point interface or extends an extension point abstract class.

### **3. Plugin Manager:**

* Jenkins Plugin Manager handles the installation, updating, and management of plugins.
* It allows users to install new plugins, update existing ones, or uninstall plugins.
* Plugins are distributed as Java Archive (JAR) files, and the Plugin Manager manages their lifecycle.

**4. Descriptor:**

* A descriptor provides metadata about a plugin or an extension point.
* Descriptors define the configuration options and user interface elements for configuring a plugin.
* Plugins typically include a descriptor class that extends **hudson.model.Descriptor**.
* Many extension points in Jenkins have associated descriptors.
* For example, a **Builder** extension point has a corresponding **BuilderDescriptor**

**Difference between Descriptor and Extension**

A Descriptor is a class associated with an extension point that provides metadata and configuration options for instances of that extension point whereas An extension class is a class that implements or extends an extension point and provides the actual functionality contributed by a plugin.

Descriptors are often annotated with **@Extension** to signal to Jenkins that they contribute to a specific extension point.

The **@Extension** annotation helps Jenkins discover and register the descriptor during the initialization process.

Extension classes are often annotated with **@Extension** as well, but this is different from the **@Extension** used with descriptors. It signals that the class contributes to an extension point,

### **5. Jenkins Core:**

* The core of Jenkins provides the basic functionality and services.
* Jenkins core includes the web server, build scheduler, user authentication, and other essential features.
* Plugins leverage and extend Jenkins core functionality through extension points.

### **6. Persistence:**

* Jenkins plugins can use persistent storage to store configuration and build data.
* Data can be stored using various mechanisms like XML files, configuration files, or by integrating with external databases.
* Jenkins provides APIs for plugins to interact with the persistence layer.

### **7. User Interface (UI):**

* Plugins can contribute to the Jenkins user interface by adding new pages, enhancing existing ones, or providing custom views.
* The UI components include Jelly scripts, which define the layout and appearance of pages.
* Jenkins supports various UI technologies, including HTML, JavaScript, and CSS.

### **8. APIs and Libraries:**

* Jenkins provides a rich set of APIs and libraries for plugin development.
* These APIs cover areas such as SCM integration, build steps, publishers, administrative tasks, and more.
* Developers use these APIs to interact with Jenkins core services and implement custom functionality.

### **9. Event System:**

* Jenkins uses an event-driven system to notify plugins about specific occurrences.
* Plugins can register listeners for events, allowing them to react to changes or trigger actions.
* Examples of events include build completion, SCM changes, and system initialization.

4th Slide -

* Setting up a development environment
* Installing required software (JDK, Maven)
* Creating a new Maven project for the plugin
* Install Jenkins on the local system

Plugin Structure

* Explanation of the basic structure of a Jenkins plugin.

So the basic structure of a Jenkins Plugin starts by creating a maven project. We Use your IDE or Maven command line to create a new Maven project:

### **1. src Directory:**

* **main Subdirectory:**
  + Contains the main source code for the plugin.
  + Typically, this directory includes packages and Java classes implementing the plugin's functionality.

### **2. pom.xml File:**

* A Maven Project Object Model (POM) file that defines the project configuration, dependencies, and build settings.
* Specifies metadata such as the plugin name, version, and dependencies on Jenkins libraries.

### **3. resources Directory:**

* Contains non-Java resources used by the plugin, such as images, configuration files, and Jelly scripts for UI customization.

### **4. META-INF Directory:**

* Contains metadata information for the plugin.
* Often includes a **MANIFEST.MF** file with details about the plugin, such as its dependencies and extension points.

### **5. test Directory:**

* Contains test-related resources and source code.
* Typically includes unit tests to ensure the correctness of the plugin's functionality.

### **6. target Directory:**

* Automatically generated by the build process (Maven).
* Contains compiled classes, JAR files, and other artifacts generated during the build.

### **7. jenkins-plugin Directory (Optional):**

* Sometimes used as a convention for placing Jenkins-specific resources.
* May contain additional configuration files or scripts used by Jenkins during the plugin's execution.

### **8. lib Directory (Optional):**

* May contain third-party libraries or dependencies needed by the plugin.
* If external libraries are used, they are often bundled here.

### **9. CHANGELOG.md File (Optional):**

* Includes a changelog that documents the version history and changes made to the plugin over time.
* Provides users and contributors with insights into updates and improvements.

### **10. LICENSE File (Optional):**

* Contains licensing information for the plugin.
* Specifies the terms under which the plugin is distributed and used.

### **11. README.md File (Optional):**

* Typically includes documentation, usage instructions, and additional information about the plugin.
* Helps users and developers understand how to install, configure, and use the plugin.

### **12. Jenkinsfile (Optional):**

* Used for defining a Jenkins Pipeline to automate the plugin's build, test, and release process.
* Facilitates Continuous Integration for the plugin.

### **13. Plugin-Specific Files:**

* Depending on the type of plugin and its functionality, additional plugin-specific files might be present. For example:
  + **config.jelly**: Defines the configuration form for the plugin.
  + **index.jelly**: Defines the UI elements for the plugin.

This structure provides a starting point for Jenkins plugin development. Plugin developers use this structure as a foundation and build upon it to implement the specific functionality and features of their plugins. The Maven build system is commonly used to manage dependencies, compile source code, and package the plugin for distribution.

* Importance of POM file (pom.xml) - if asked search for it.
* Create the plugin descriptor

Creating a plugin descriptor in a Jenkins plugin involves creating a class that extends the **hudson.model.Descriptor** class. The descriptor provides metadata and configuration options for instances of your plugin. Below are the steps to create a basic plugin descriptor:

### **1. Create a Descriptor Class:**

Create a Java class that extends **hudson.model.Descriptor**. For example:

**2. Implement Descriptor Methods:**

In your descriptor class, you need to implement certain methods to provide metadata and configuration options. Commonly used methods include:

* **getDisplayName:** Specifies the display name of your plugin in the Jenkins UI.
* **getHelpFile:** Optionally provides a path to an HTML file containing help information.
* **configure:** Handles the form submission when configuring the plugin in Jenkins.
* **Additional Methods:** Depending on your plugin's requirements, you might need to implement other methods. For example, if your plugin has configurable global settings, you may need to override the **newInstance** method.

}**3. Annotate with @Extension:**

Annotate your descriptor class with **@Extension** to let Jenkins discover and register it:

### **4. Build and Deploy:**

Build your plugin using Maven and deploy the generated **.hpi** file to your Jenkins instance for testing.

This basic structure allows your Jenkins plugin to have a descriptor that provides essential metadata and handles configuration. Depending on your plugin's complexity, you may need to extend or customize the descriptor further.

Writing Code

* Creating a new Java class for the custom plugin
* Implementing ExtensionPoint or specific extension point interfaces
* Understanding different extension points in Jenkins
* Choosing the appropriate extension point for the custom plugin
* Implementing the extension point in the plugin code
* Defining plugin functionalities

Testing the Plugin

* Writing unit tests for the custom plugin
* Using Jenkins Plugin POM to simplify testing
* Running tests to ensure the plugin functions as expected

Packaging and Deployment

* Configuring the plugin for packaging
* Building the plugin using Maven
* Deploying the plugin to a Jenkins instance for testing

Publishing the Plugin

* Discussing the process of publishing the custom plugin
* Overview of plugin documentation and metadata
* Uploading the plugin to the Jenkins plugin repository

Slide 5 -

### **Advanced Topics and Best Practices**

* Advanced plugin features (e.g., pipeline support, UI enhancements)
* Jenkins API usage
* Best practices for plugin development
* Versioning and compatibility considerations

### **1. Advanced Plugin Features:**

* **Pipeline Support:**
  + Integrate your plugin with Jenkins Pipeline, enabling users to define their build processes as code.
  + Implement a custom DSL (Domain-Specific Language) for your plugin to be used in Jenkinsfiles.
* **UI Enhancements:**
  + Enhance the Jenkins user interface by creating custom views, widgets, or pages.
  + Utilize JavaScript and CSS to improve the user experience.
  + Consider integrating with the Blue Ocean UI for a modern and visually appealing interface.
* **Integration with Other Plugins:**
  + Explore integration possibilities with other popular plugins.
  + Leverage the extension points provided by other plugins to create a seamless user experience.

### **2. Jenkins API Usage:**

* **Job DSL API:**
  + Use the Job DSL API to programmatically create and configure Jenkins jobs.
  + Allows users to define jobs using a Groovy-based DSL.
* **BuildStep API:**
  + Implement custom build steps using the BuildStep API.
  + Define actions that should be taken during the build process.
* **Remote Access API:**
  + Interact with Jenkins remotely using the Remote Access API.
  + Implement features such as triggering builds, fetching build results, etc.

### **3. Best Practices for Plugin Development:**

* **Follow Code Standards:**
  + Adhere to Java coding standards and conventions.
  + Write clean, modular, and well-documented code.
* **Test-Driven Development (TDD):**
  + Write unit tests for your plugin using frameworks like JUnit.
  + Use Jenkins' integration testing framework for end-to-end testing.
* **Continuous Integration:**
  + Set up a continuous integration pipeline for your plugin using Jenkins.
  + Automate the build, test, and deployment processes.
* **Logging and Debugging:**
  + Use Jenkins logging facilities for effective debugging.
  + Implement logging statements strategically to aid troubleshooting.
* **Security Considerations:**
  + Follow best practices for securing your plugin.
  + Validate input, sanitize output, and adhere to Jenkins' security guidelines.

### **4. Versioning and Compatibility Considerations:**

* **Semantic Versioning:**
  + Adopt semantic versioning for your plugin (major.minor.patch).
  + Clearly communicate backward compatibility and breaking changes.
* **Jenkins API Changes:**
  + Stay informed about changes to Jenkins core API.
  + Test your plugin against different versions of Jenkins to ensure compatibility.
* **Plugin Dependencies:**
  + Clearly specify dependencies on other plugins and libraries.
  + Handle version conflicts and ensure compatibility with a range of plugin versions.
* **Documentation:**
  + Maintain comprehensive documentation for your plugin.
  + Include information about supported Jenkins versions and any version-specific considerations.
* **Release Management:**
  + Plan and communicate releases effectively.
  + Use version control tags and release notes for transparency.

These advanced topics and best practices contribute to the development of robust, maintainable, and user-friendly Jenkins plugins. They also ensure that your plugin aligns well with Jenkins' evolving ecosystem and provides a positive experience for users.

Show and explain the code.

Slide 7 -

Gitlab and Circle CI

Slide 8 -

Before we wrap up, I'd like to take a moment to extend my heartfelt thanks to someone who has been instrumental in making this journey possible. To my mentor, to my GURU, Mr. GuruMurthy Arumugam, your guidance and support have been invaluable. Your wisdom and encouragement have fueled not only the success of this presentation but also my personal and professional growth.

I'm truly grateful for the opportunities you've provided, the lessons you've shared, and the inspiration you've instilled in me. Your mentorship is the main reason for what I have accomplished in this new field today.

# Thank you!