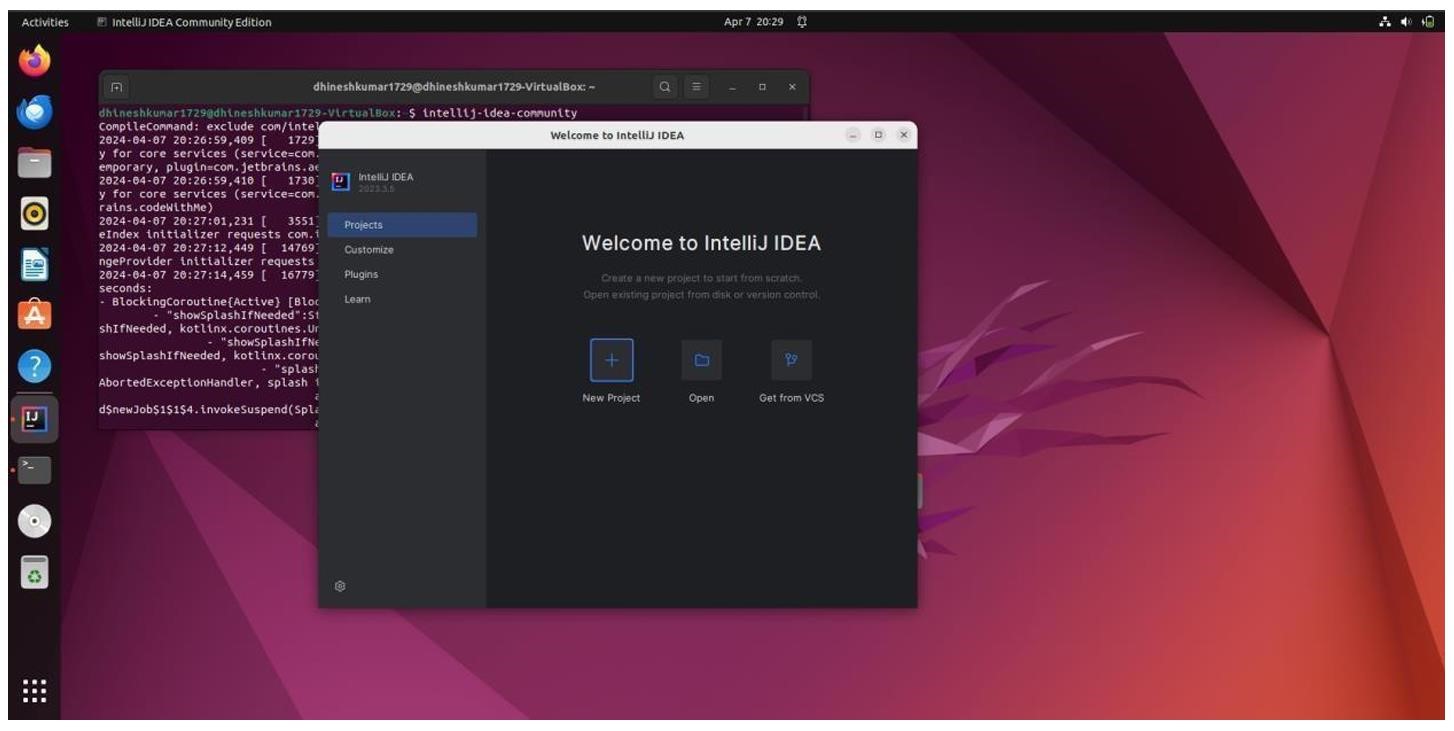
**Experiment 10: Creating Build Pipelines: Building a Maven/Gradle Project with Azure Pipelines, Integrating Code Repositories (e.g., GitHub, Azure Repos), Running Unit Tests and Generating Reports**

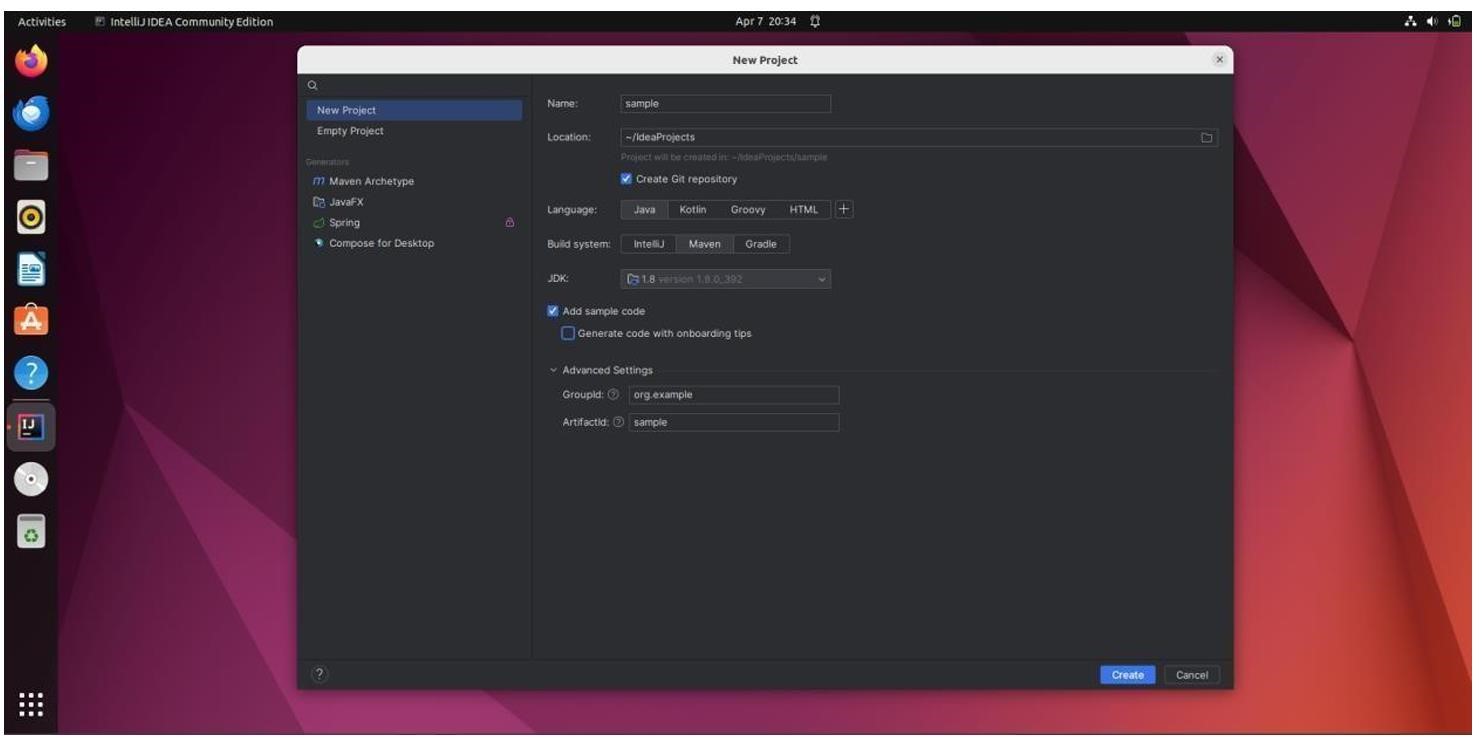
# Step 1: Installing IntelliJ IDEA

1. Download IntelliJ Idea from https://www.jetbrains.com/idea/download/?section=linux and activate a free trial for 30 days or through the following command. $ sudo snap install intellij-idea-community --community
2. Open the IDE using the below command in the terminal. $ intellij-idea-community
3. Java is a pre-requisite for using this IDE



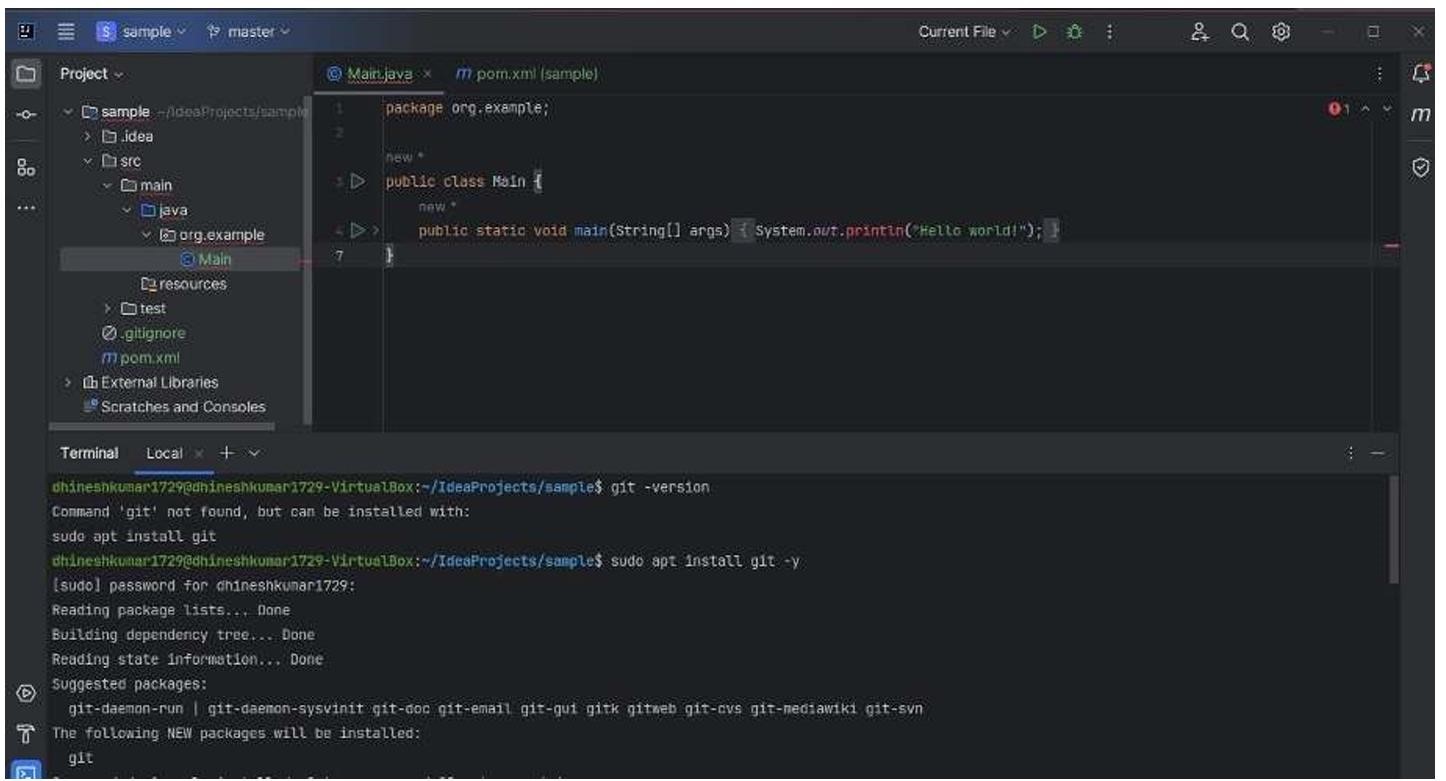
**Step 2: Create a new Maven project using the IDE.**

Create a project by clicking on the New Project option, choose an appropriate name and select the following features as mentioned in the image (Build system: Maven andLanguage: Java)



# Step 3: Install GIT and configure it

1. Check whether git is installed using the following command. $ git -version
2. Install git using $ sudo apt install git



1. Once git has been installed execute the following commands one by one.

$ git init $ git add .

$ git commit -m “First Commit”

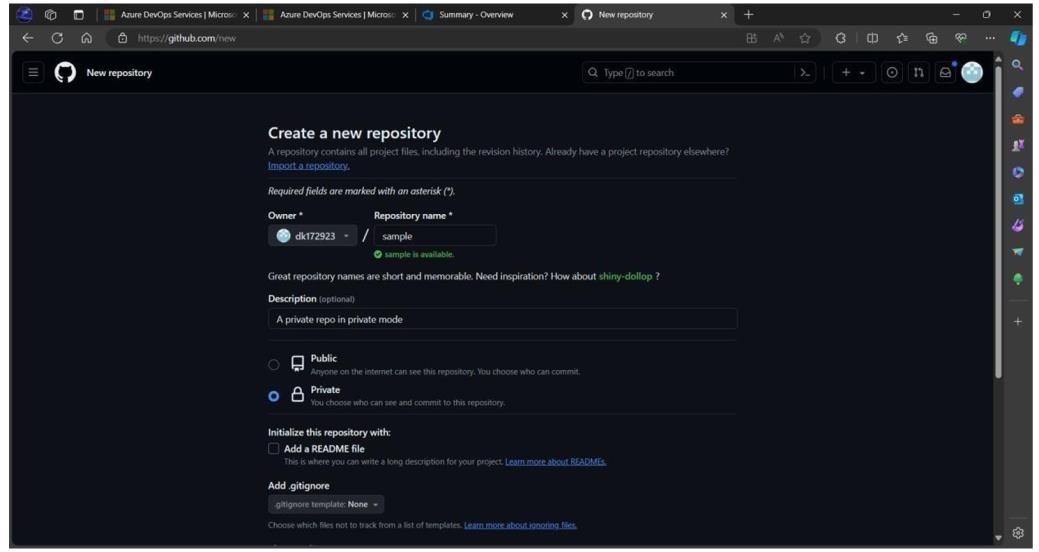
$ git branch -M main

Configure git using the below commands

$ git config --global user.name “Your\_Name”

$ git config --global user.email “Your\_Email\_ID”

Go to https://github.com sign in with your account and create a new private repository with the same name as your maven project.



1. Copy the SSH URL for the created repository.
2. Now go back to IDE’s terminal and execute the following commands

$ ssh-keygen -t rsa -b 4096 -C youremail@gmail.com

(press ENTER for all questions)

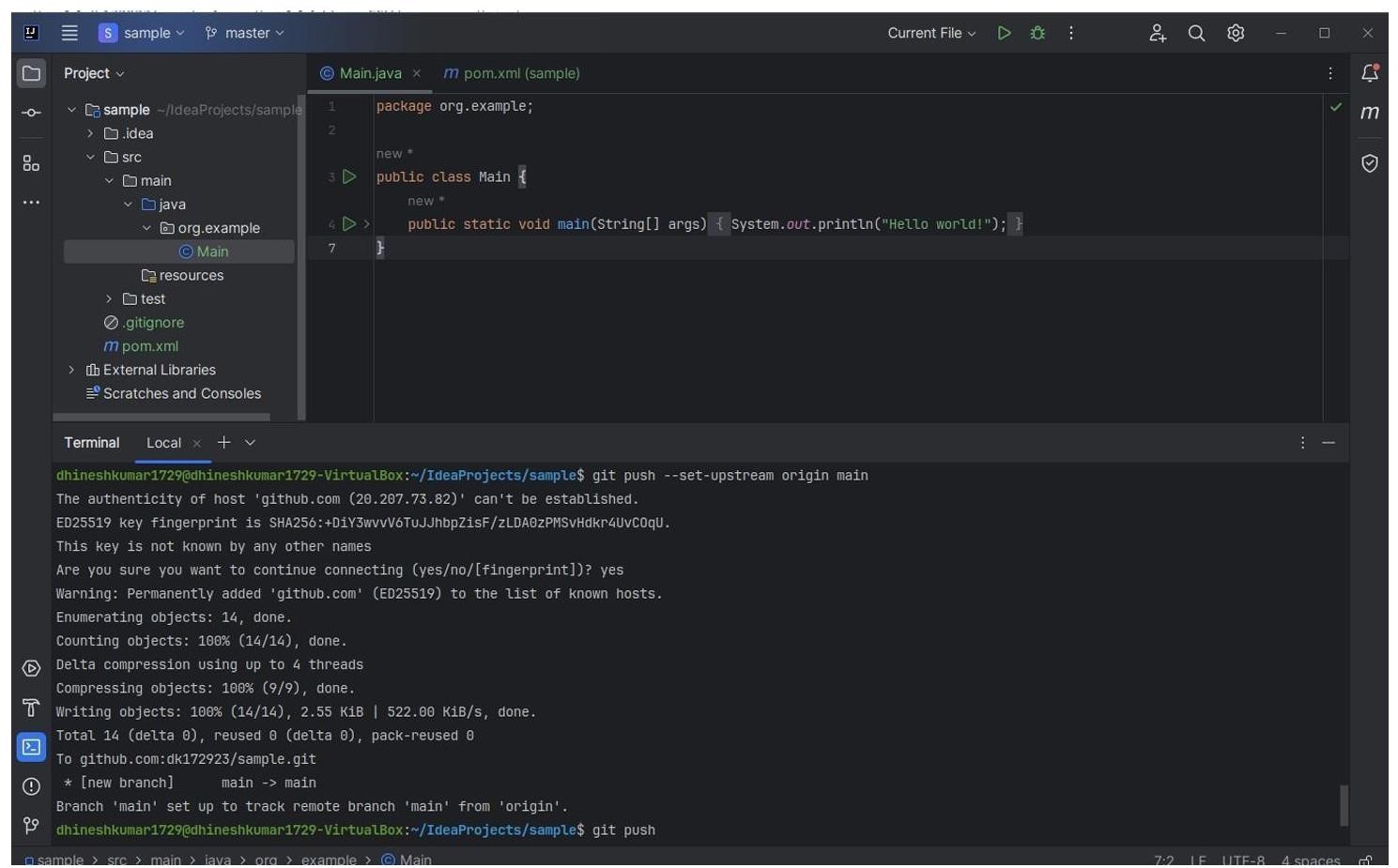
$ cat ~/.ssh/id\_rsa.pub (copy the printed SSH key)

1. Now go to the SSH and GPG section in the GitHub settings option.
2. Click on the New SSH key button.
3. Choose a suitable name and paste the SSH key into the provided space.
4. Now get back to the IDE’s terminal and type the following command $ git remote set-url origin git@github.com:repo\_owner/repo\_name.git (Here, git@github.com:dk172923/sample.git is the remote URL to use SSH dk172923 – GitHub ID sample.git – repository name)

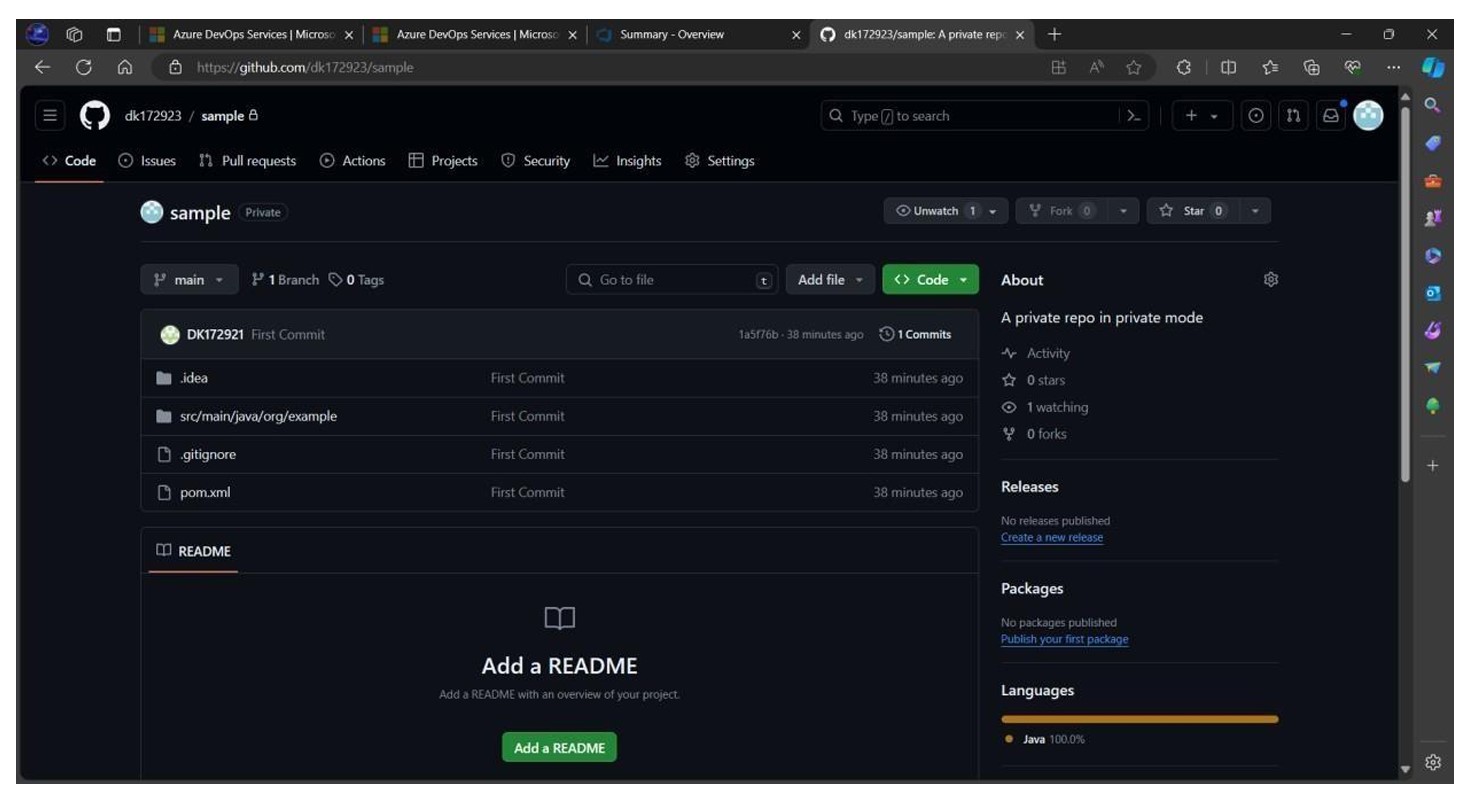
Now finally run the push commands to push the code to the remote repository from the local repository.

$ git push --set-upstream origin main (Type YES for prompted question)

$ git push

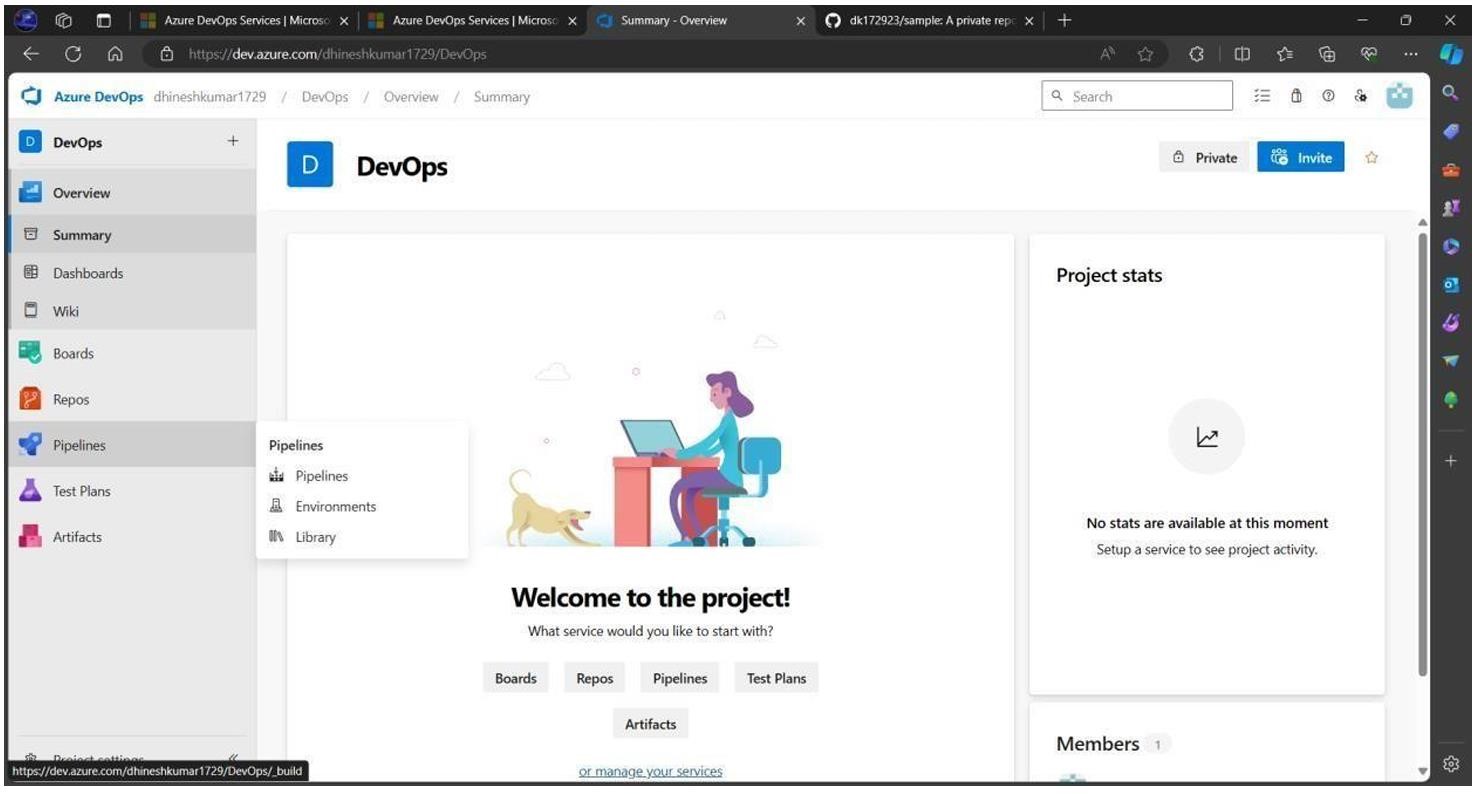


The code has been successfully pushed to the remote repository.

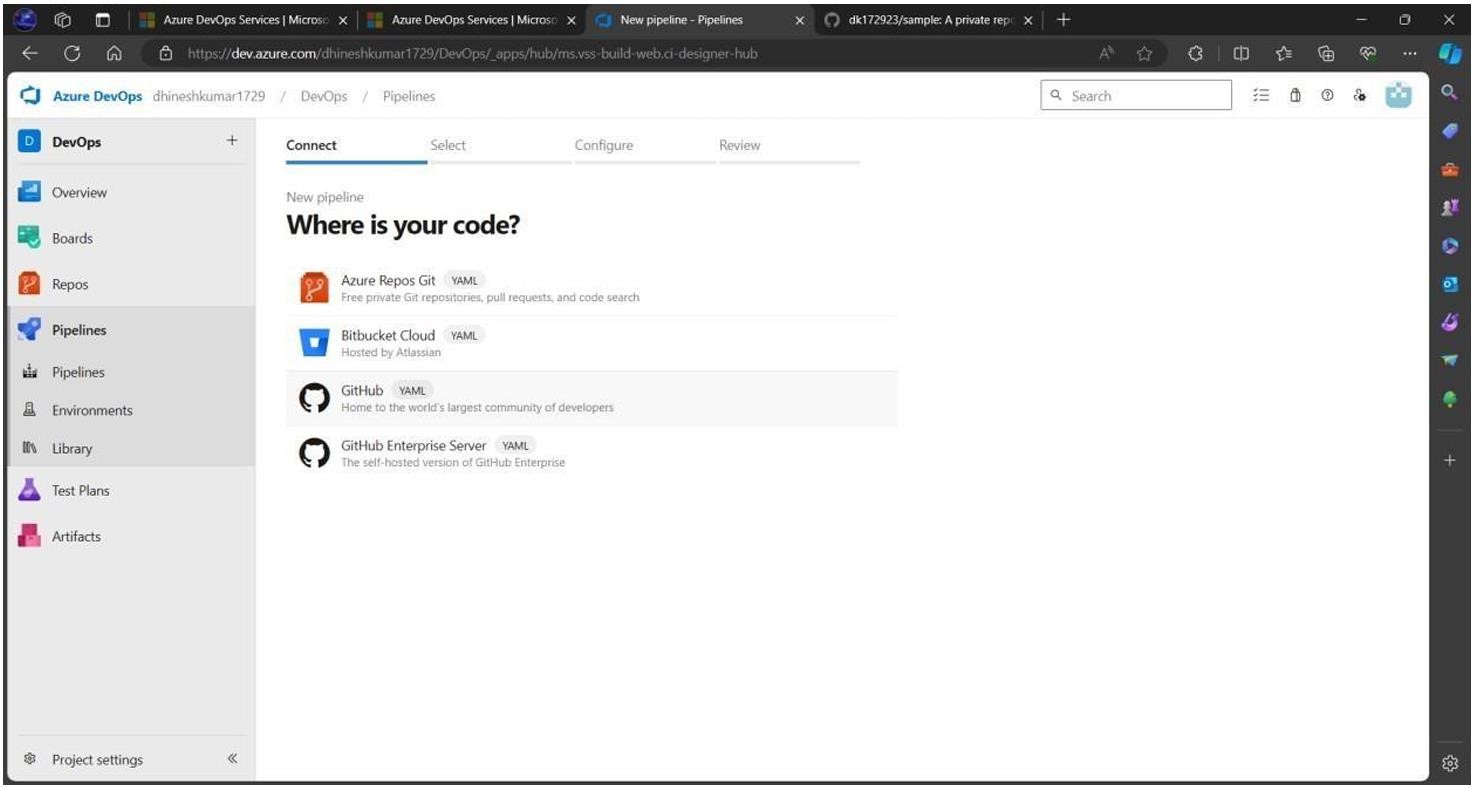


STEP 5: Create a pipeline for the created maven repository.

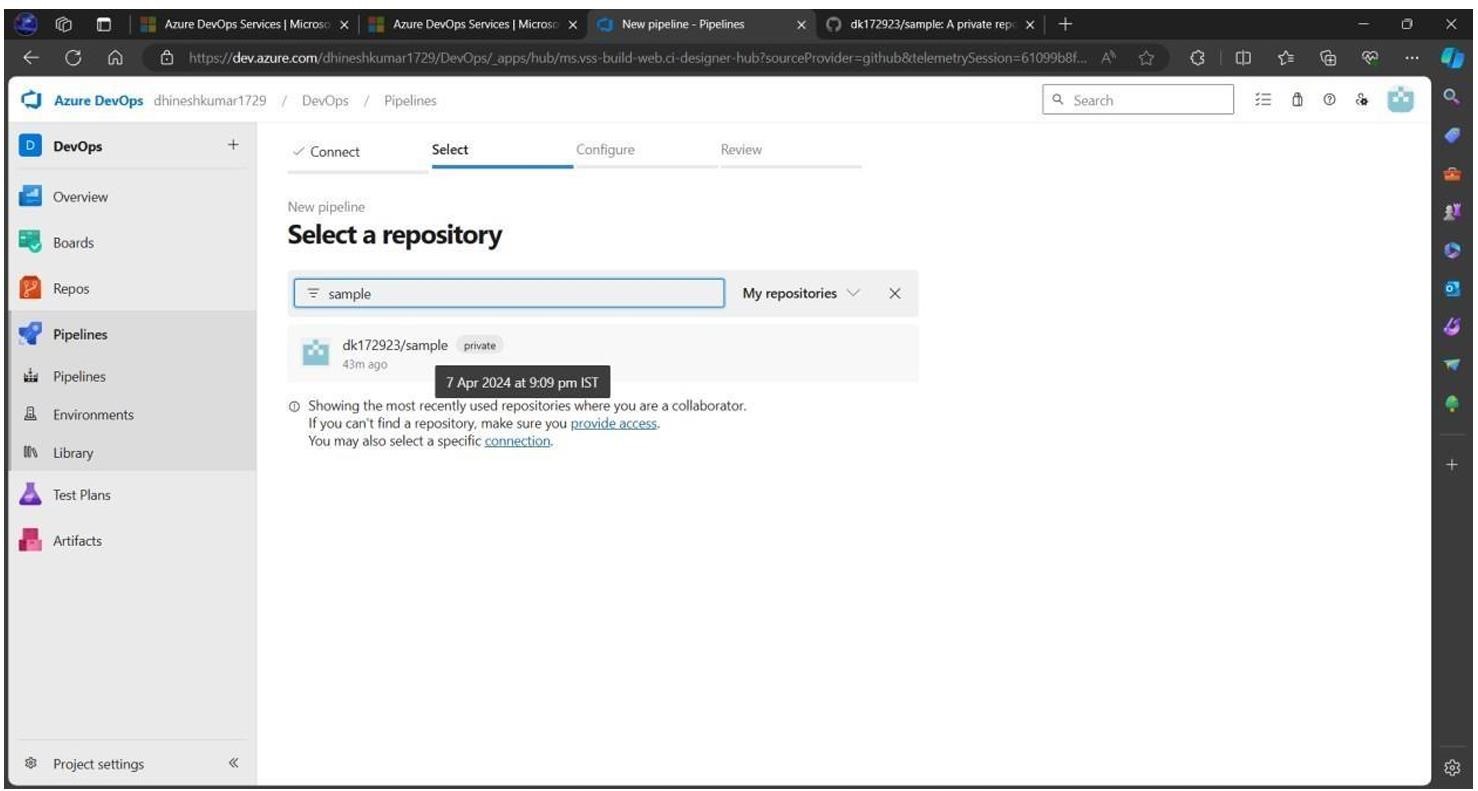
Click on Pipelines on the left pane of the Azure DevOps website.



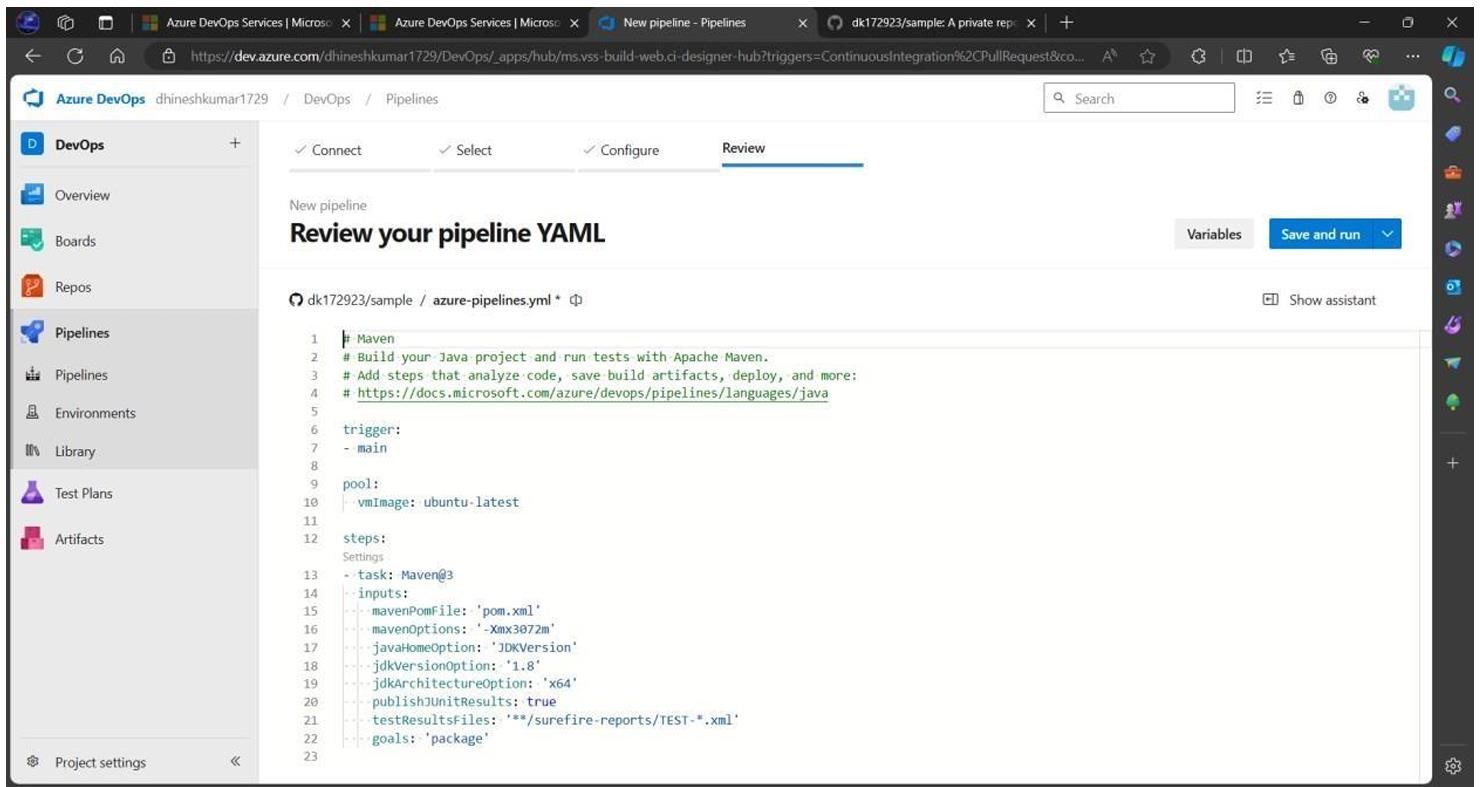
1. Click on the Create Pipeline button.
2. Choose the GitHub YAML option.



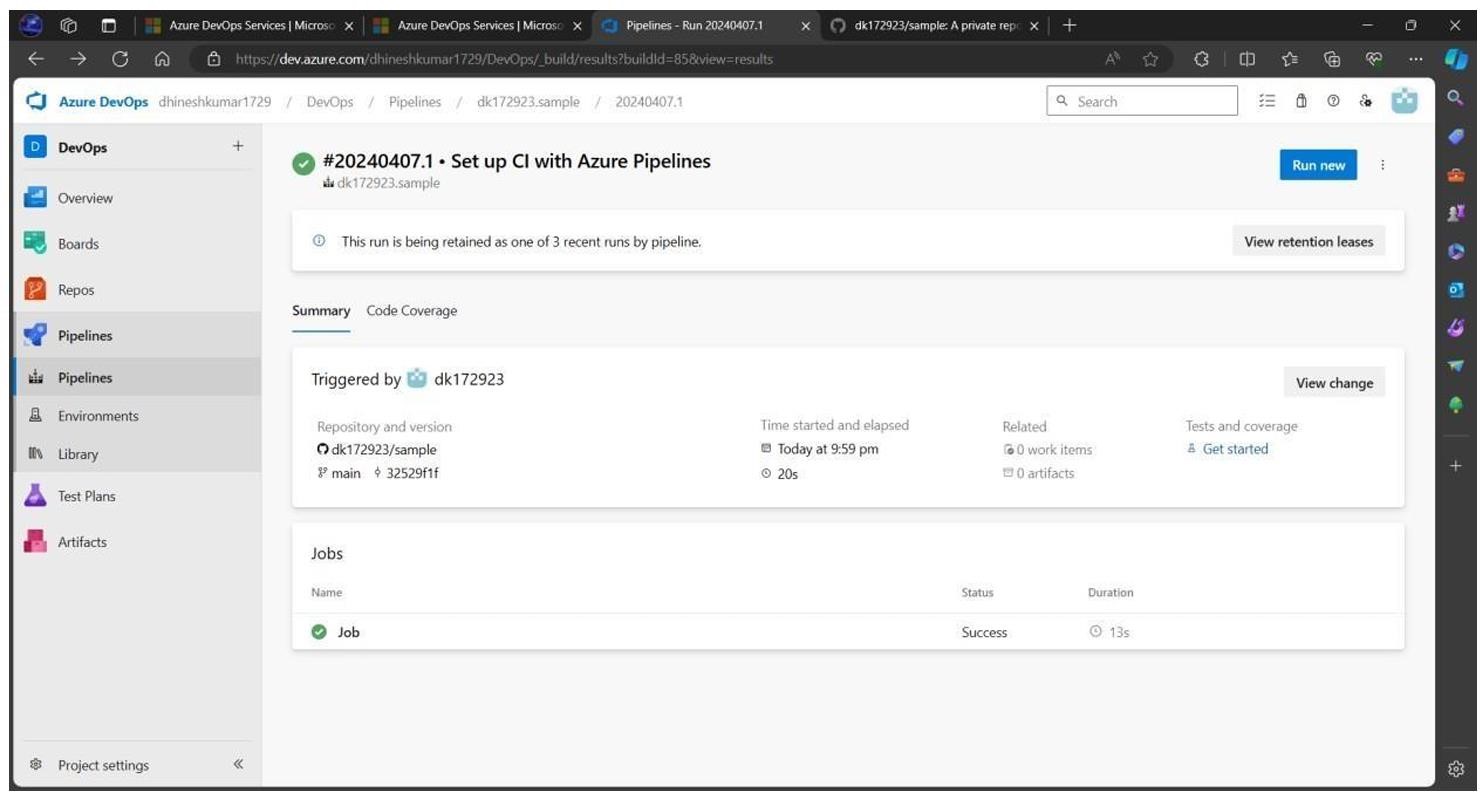
1. Select the created repository “sample” from the available repositories.
2. Give permissions if prompted to do so and sign in to your GitHub account if needed.



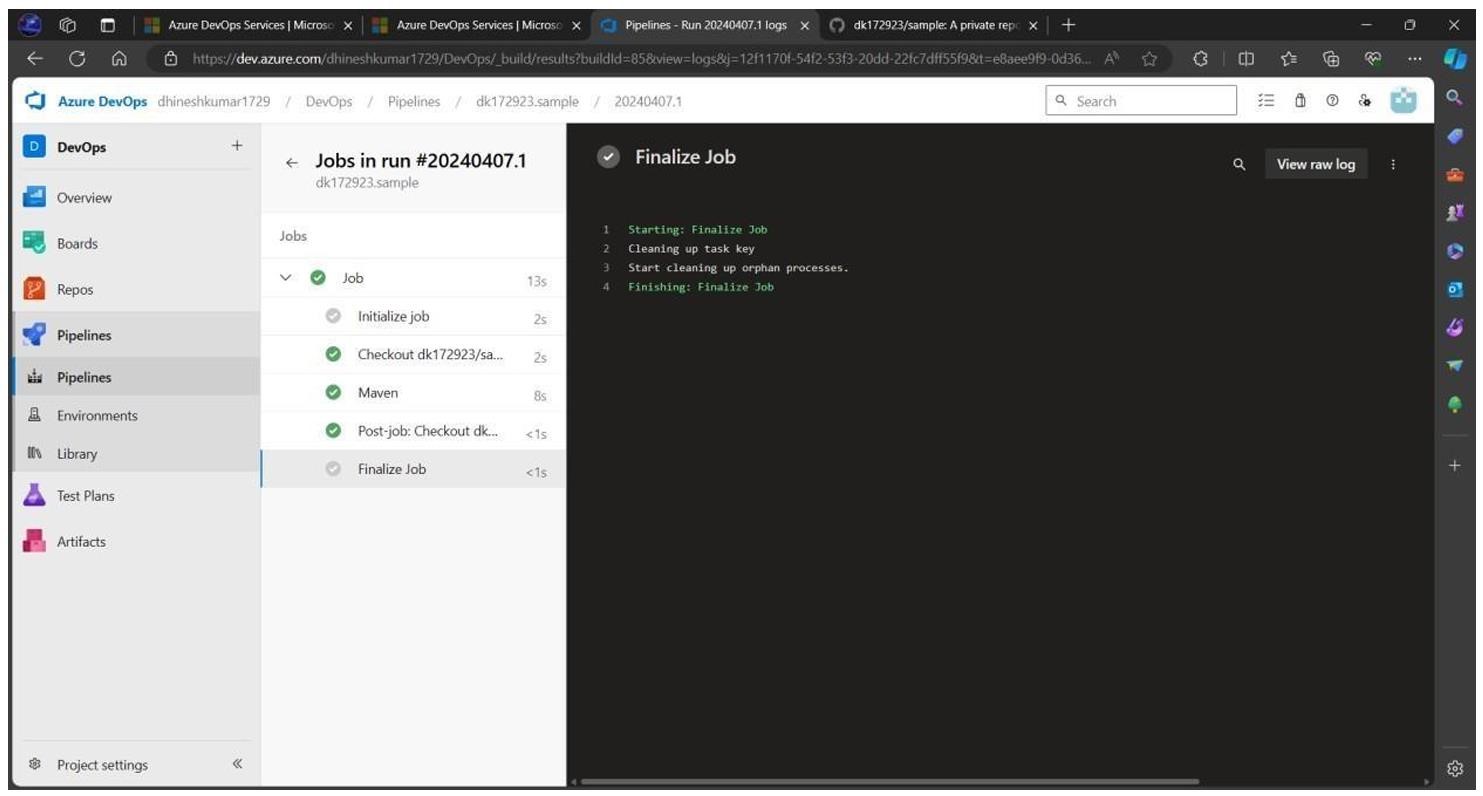
1. Choose Maven Pipeline from the given options.
2. A YAML file is automatically created based on the configuration details found in the pom.xml file in the repository.



1. Press the Save and Run button.
2. Create a commit message and press the run button again.
3. Click on the created Job from the Jobs table.



A successful job completion would be like this.



Below is a detailed section that explains how to run unit tests and generate reports in an Azure

DevOps pipeline after your Maven project has been built. This section explains what happens during the Maven build process, how test results are generated, and how to publish these results as part of your pipeline execution.

**Running Unit Tests and Generating Reports with Maven in Azure DevOps**

When you build your Maven project using Azure Pipelines, the build process usually includes running unit tests with the Maven Surefire plugin. This plugin executes tests (typically written with JUnit) and produces test result files in XML format. Azure Pipelines can then pick up these XML files and present them as part of the build summary. Below are the steps and details to ensure that your unit tests are executed and the reports are published.

1. **Maven Surefire Plugin and Test Reports What Happens During the Maven Build?** 
   * **Maven Surefire Plugin:**

When you run the command mvn clean package (or mvn test), the Surefire plugin automatically executes the unit tests found in the src/test/java directory.

* + **Test Report Generation:**

By default, the Surefire plugin creates XML reports in the target/surefire-reports/ directory. These files usually have names like TEST-<TestClassName>.xml.

* + **Importance of Test Reports:**

These XML files contain detailed information on test execution, including the number of tests run, passed, failed, and any error messages or stack traces.

**Configuring Your Azure Pipeline to Publish Test Results**After your Maven build runs and tests are executed, you need to add a step in your Azure Pipeline YAML file that locates these test reports and publishes them in Azure DevOps. This is accomplished by using the **PublishTestResults@2** task.

**YAML Configuration Example for Maven**

Below is a sample snippet of a YAML pipeline configuration that includes both the Maven build step and a step to publish test results: trigger:

main

pool:

vmImage: 'ubuntu-latest'

steps:

# Step 1: Maven Build and Test Execution

task: Maven@3

inputs:

mavenPomFile: 'pom.xml' # Ensure your pom.xml is at the repository root goals: 'clean package' options: '' # Add any additional Maven options if needed

# Step 2: Publish Unit Test Results - task: PublishTestResults@2

inputs:

testResultsFiles: '\*\*/target/surefire-reports/TEST-\*.xml' mergeTestResults: true testRunTitle: 'Maven Unit Test Results'

**Explanation of the YAML Steps:**

1. **Maven Build Step:**

**Task:** Maven@3 o **Input:**

**mavenPomFile:** Points to your pom.xml.

**goals:** The goals clean package ensure that your code is cleaned, compiled, tested, and packaged. The Surefire plugin runs tests during this process.

**Outcome:** Maven executes the tests and creates XML reports under the target/surefire-reports/ folder.

1. **Publish Test Results Step:**
   * **Task:** PublishTestResults@2 o **Input:**
     + **testResultsFiles:** Uses a glob pattern (\*\*/target/surefire-reports/TEST\*.xml) to find all XML test result files.
     + **mergeTestResults:** Set to true so that if there are multiple result files, they are merged into a single report.
     + **testRunTitle:** Provides a title for the test run that will appear in the Azure DevOps test results view.

**Outcome:** Azure Pipelines reads the test result XML files and displays a summary in the pipeline's **Tests** tab. You’ll be able to see metrics like the number of tests passed, failed, or skipped, and review detailed error messages for any failures.

1. **Running and Verifying the Pipeline**

After committing the YAML file to your repository, the pipeline is triggered (either automatically or manually):

1. **Trigger the Pipeline:**
   * Once your YAML file is saved in your repository, Azure Pipelines will pick up the changes. If not automatically triggered, you can click **“Run pipeline”** manually.
   * **Screenshot Tip:** Capture the pipeline run screen where you see the stages (e.g., Checkout, Maven Build, Publish Test Results).
2. **Monitor the Build Output:**

Navigate to the **“Build”** or **“Logs”** section of the pipeline run. o Confirm that the Maven task logs show the execution of tests and that the Surefire reports are created.

1. **Review Test Reports:**
   * Once the build completes, click on the **“Tests”** tab (often found on the pipeline summary page) to review the detailed test results.
   * **Screenshot Tip:** Capture the test report summary, showing the number of tests executed, passed, and failed. You may also capture detailed logs for any failed tests.

# Experiment 11: Creating Release Pipelines: Deploying Applications to Azure App Services, Managing Secrets and Configuration with Azure Key Vault, Hands-On: Continuous Deployment with Azure Pipelines

1. **Overview**

In this experiment, you will:

* + **Deploy your build artifact** (e.g., a JAR or WAR file from a Maven/Gradle project) to an Azure App Service.
  + **Manage secrets and configuration** securely using Azure Key Vault.
  + **Set up a release pipeline** in Azure DevOps that automatically deploys your application when a new artifact is available (continuous deployment).

This experiment bridges the gap between build automation (CI) and release automation (CD) while ensuring secure management of sensitive information.

1. **Prerequisites**

Before you begin, ensure that you have:

* + An **Azure DevOps account** with a project set up (see Experiment 9 and Experiment 10).
  + A **build artifact** (e.g., your Maven/Gradle artifact) available from a build pipeline.
  + An **Azure Subscription** with an **Azure App Service** instance already created to host your application.
  + An **Azure Key Vault** instance created in your Azure Subscription for storing secrets (e.g., connection strings, API keys).
  + Appropriate permissions to create and manage resources in Azure DevOps and your Azure Subscription.

1. **Creating a Release Pipeline in Azure DevOps**
2. **Create a New Release Pipeline** 
   1. **Log in to Azure DevOps:**

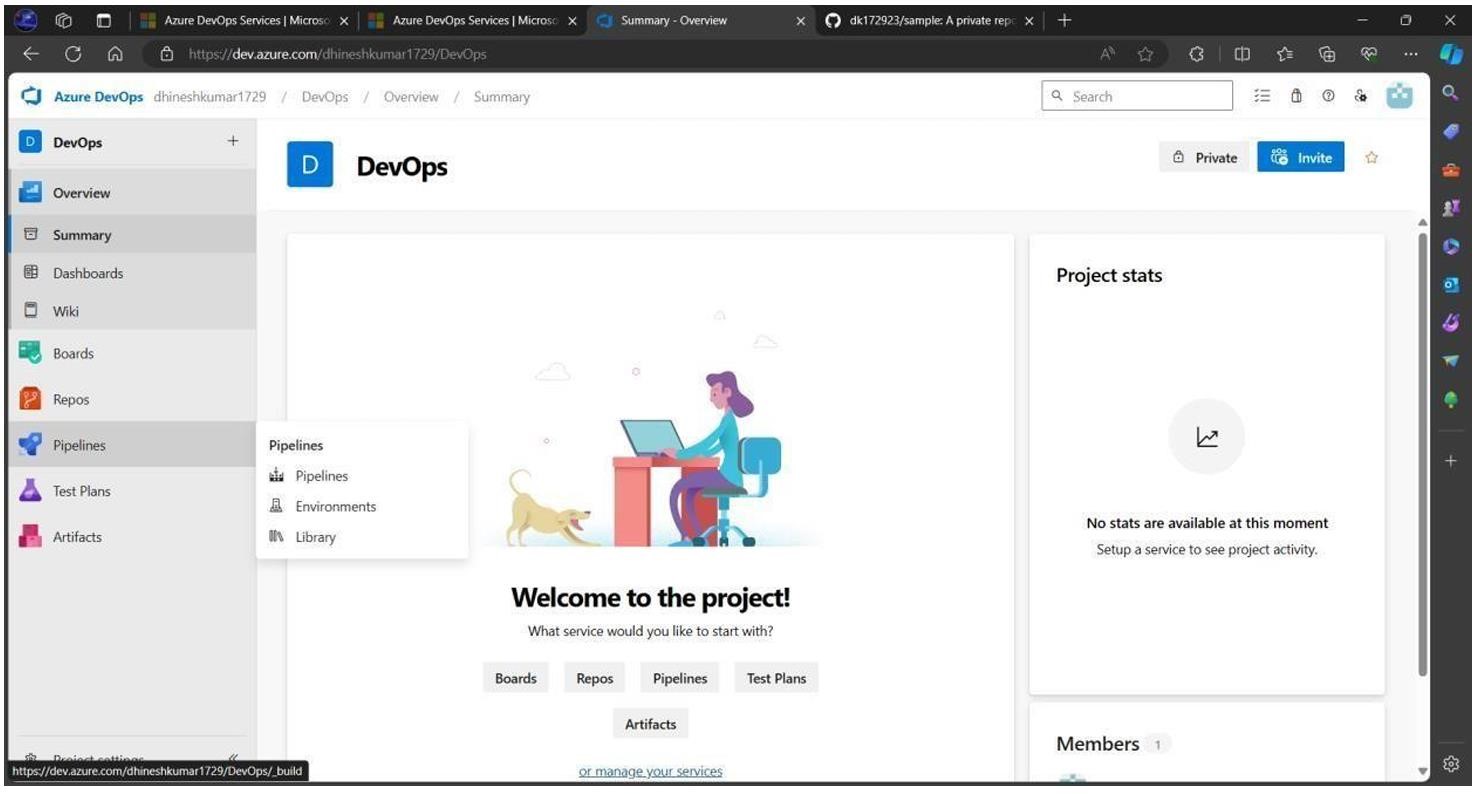
Open your web browser and navigate to your Azure DevOps project (e.g., [https://dev.azure.com/YourOrganization)](https://dev.azure.com/YourOrganization).

**Navigate to the Releases Section:**

* + - In the left-hand menu, click on **“Pipelines”** and then **“Releases”**.

3. **Create a New Pipeline:**

* + - Click on **“New pipeline”**.
    - When prompted, select **“Empty job”** (or start with a template if one suits your needs).



1. **Add an Artifact** 
   1. **Link Your Build Artifact:**
      * In the release pipeline editor, click on **“Add an artifact”**. o **Source Type:** Choose the source (e.g., “Build”).
      * **Source (Build Pipeline):** Select the build pipeline that produces your Maven/Gradle artifact.
      * **Default Version:** Use the latest version (or specify a branch/tag as needed). o Click **“Add”**.

1. **Define a Stage for Deployment** 
   1. **Add a New Stage:**

Click on **“Add a stage”** and select **“Empty job”**. o Rename the stage (e.g., Development or Production).

**Configure the Stage:**

* + - Click on the stage to open its settings. o In the **“Tasks”** view for the stage, click **“+”** to add a new task.

1. **Deploying to Azure App Services A. Add the Azure App Service Deploy Task** 
   1. **Add Deployment Task:**
      * In the stage tasks, click on **“+ Add”** and search for **“Azure App Service**

**Deploy”**.

* + - Click **“Add”** to include it in your stage.
  1. **Configure the Azure App Service Deploy Task:**
     + **Azure Subscription:**
       - Click on **“Authorize”** (if required) and select your Azure subscription. o **App Service Type:** Choose **“Web App on Windows”** or **“Web App on**

**Linux”** based on your deployment target.

* + - **App Service Name:** Enter the name of your Azure App Service instance.
    - **Package or Folder:**
      * Specify the path to your build artifact (for example, $(System.DefaultWorkingDirectory)/\_HelloMaven-

CI/drop/HelloMaven-1.0-SNAPSHOT.jar or the folder containing your package). Adjust the path based on your artifact’s location.

* + - **Deployment Options:** Configure any additional options (e.g., deployment slot, resource group) as needed.

1. **Managing Secrets and Configuration with Azure Key Vault A. Create and Configure an Azure Key Vault (in Azure Portal)** 
   1. **Create a Key Vault (if not already created):**
      * Log in to the [Azure Portal.](https://portal.azure.com/) o Click on **“Create a resource”** and search for **“Key Vault”**.

Follow the prompts to create a new Key Vault (enter a name, select subscription, resource group, and region).

**Add Secrets to Your Key Vault:**

* + - Once the Key Vault is created, navigate to it.
    - Click on **“Secrets”** and then **“Generate/Import”**. o Create new secrets (e.g., DBConnectionString, APIKey) and note their names.

**B. Integrate Key Vault with Azure DevOps**

1. **Create a Variable Group Linked to Key Vault:**
   * In your Azure DevOps project, navigate to **“Pipelines”** and then **“Library”**. o Click on **“+ Variable group”**. o Name the variable group (e.g., KeyVault-Secrets).
   * Enable **“Link secrets from an Azure Key Vault as variables”**. o Click **“Authorize”** and select your Azure subscription. o Select your Key Vault from the dropdown list. o Choose the secrets you want to import into the variable group.
   * Save the variable group.
2. **Use the Variable Group in Your Release Pipeline:**
   * In your release pipeline, click on the **“Variables”** tab.
   * Click **“Variable groups”** and then **“Link variable group”**. o Select the variable group you created (KeyVault-Secrets). o Now, you can reference these secrets as variables in your deployment tasks (for example, $(DBConnectionString)).

**6. Enabling Continuous Deployment A. Configure Continuous Deployment Trigger**

1. **Enable Continuous Deployment:**
   * In your release pipeline, click on the **“Triggers”** tab.
   * Under **“Artifact filters”** or **“Continuous deployment trigger”**, enable the toggle to allow the release pipeline to trigger automatically when a new build artifact is available.
   * Optionally, you can specify branch filters or artifact version filters.
2. **Save the Release Pipeline:**
   * Click **“Save”** to apply all changes.

**B. Test the Continuous Deployment Pipeline**

1. **Trigger a Build:**
   * Commit a change to your code repository to trigger your build pipeline (or trigger it manually).
   * Verify that the build pipeline creates a new artifact.
2. **Automatic Release:**
   * Once the new artifact is published, the release pipeline should automatically trigger a new release.
   * Monitor the release pipeline execution to ensure that:
     + The artifact is deployed to the Azure App Service.
     + The deployment task uses the configuration and secrets from Key Vault.

**Experiment 12: Practical Exercise and Wrap-Up: Build and Deploy a Complete DevOps Pipeline, Discussion on Best Practices and Q&A**

1. **Overview**

In this experiment, you will create an end-to-end DevOps pipeline that demonstrates the following processes:

* + **Version Control:** Code is maintained in a Git repository (e.g., GitHub or Azure Repos).
  + **Continuous Integration (CI):**
    - A CI tool (Jenkins and/or Azure Pipelines) automatically checks out the code, builds it using Maven/Gradle, runs unit tests, and archives the build artifact.
  + **Artifact Management:** The artifact (e.g., a JAR file) is archived and made available for deployment.
  + **Continuous Deployment (CD):**
    - Deployment automation is handled either by an Ansible playbook or an Azure Release pipeline, deploying the artifact to a target environment (such as an Azure App Service or a local server).
  + **Secrets and Configuration Management:** Securely manage configuration details and secrets using Azure Key Vault.
  + **Pipeline as Code:** Use YAML (for Azure Pipelines) or a Jenkinsfile (for Jenkins) to define your build and release processes.

After setting up and running the pipeline, we will discuss best practices for designing and maintaining such pipelines and open the floor for a Q&A discussion.

1. **Prerequisites**

Before starting, ensure you have completed or have access to the following:

* + **Source Code Repository:**
    - A Java project (e.g., “HelloMaven” or “HelloGradle”) hosted on GitHub or Azure Repos.
    - The project should follow a standard structure (with pom.xml for Maven or build.gradle for Gradle, and appropriate src/main/java and src/test/java directories).
  + **Jenkins and/or Azure DevOps Setup:**
    - Jenkins installed and configured on your local machine or a cloud server (refer to Experiments 5 and 6), or an Azure DevOps project set up with a build pipeline (Experiments 9 and 10).
  + **Ansible Installed:**
    - Ansible is installed on your control machine (or Jenkins server) with a basic inventory file (see Experiment 7).
  + **Azure Resources:** (Optional but recommended for cloud deployment) o An Azure App Service instance created to host your application. o An Azure Key Vault instance set up to store sensitive data (e.g., connection strings).
  + **Access Credentials:**
    - Permissions to commit code to the repository. o Administrative access on Jenkins/Azure DevOps. o Proper permissions on Azure to deploy to App Services and to manage Key Vault secrets.

1. **Step-by-Step Pipeline Setup**

**Step 1: Code Repository Preparation**

1. **Ensure Your Project Is in Version Control:**

* + - Navigate to your project folder (e.g., “HelloMaven”) and initialize Git if not already done:
    - cd /path/to/HelloMaven o git init o git add .
    - git commit -m "Initial commit of HelloMaven project" o Push the project to your remote repository (e.g., GitHub):
    - git remote add origin https://github.com/yourusername/HelloMaven.git o git push -u origin main

**Step 2: Continuous Integration with Jenkins and/or Azure Pipelines**

1. **Jenkins CI Pipeline Setup** 
   1. **Create a New Jenkins Job:**
      * Log in to Jenkins and click **“New Item”**.
      * Enter a name (e.g., HelloMaven-CI) and select **“Freestyle project”** or

**“Pipeline”**.

* + - *Screenshot Tip:* Capture the new job creation screen.
  1. **Configure Source Code Management:**
     + Under the **“Source Code Management”** section, select **“Git”**.
     + Enter your repository URL (e.g., https://github.com/yourusername/HelloMaven.git) and set branch specifier to \*/main.
  2. **Add Build Steps:**
     + **For Maven Projects:**

Add a build step “Invoke top-level Maven targets” and set the goals to:

clean package

o **For Pipeline-as-Code (Jenkinsfile):**

Create a Jenkinsfile in your repository with stages for checkout, build, test, and archive. For example:

pipeline {

agent any

stages {

stage('Checkout') {

steps {

git url: 'https://github.com/yourusername/HelloMaven.git', branch: 'main'

}

}

stage('Build') {

steps {

sh 'mvn clean package'

}

}

stage('Test') {

steps {

sh 'mvn test'

}

}

stage('Archive') {

steps {

archiveArtifacts artifacts: 'target/\*.jar', fingerprint: true

}

}

}

}

* 1. **Run the Jenkins Job:**
     + Click **“Build Now”** and monitor the console output. Ensure that the build is successful and that test reports are generated.

1. **Azure DevOps Build Pipeline Setup** 
   1. **Create a New Pipeline:**
      * Log in to your Azure DevOps project. o Navigate to **“Pipelines”** and click **“New pipeline”**.
      * Select your repository source (GitHub or Azure Repos) and choose your repository.
   2. **Define Your YAML Pipeline:**

trigger:

- main

pool:

vmImage: 'ubuntu-latest'

steps:

- task: Maven@3

inputs:

mavenPomFile: 'pom.xml'

goals: 'clean package'

- task: PublishTestResults@2

inputs:

testResultsFiles: '\*\*/target/surefire-reports/TEST-\*.xml'

mergeTestResults: true

testRunTitle: 'Maven Unit Test Results'

**Run the Pipeline and Verify Test Reports:**

* + - Commit and run the pipeline. o Navigate to the **“Tests”** tab to view the summary of executed tests

**Step 3: Artifact Management**

1. **Artifact Archiving in Jenkins:**

* + - Ensure your Jenkins job archives the artifact (JAR file) using the **“Archive the artifacts”** post-build action.

2. **Artifact in Azure Pipelines:**

* + - The build task produces an artifact that can be downloaded or referenced by subsequent release pipelines.

**Step 4: Deployment Automation with Ansible and Azure Release Pipeline**

1. **Using Ansible for Deployment** 
   1. **Write an Ansible Playbook:**

---

- name: Deploy Maven Artifact

hosts: deployment

become: yes

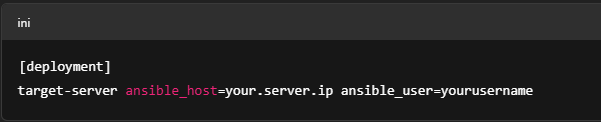
tasks:

- name: Copy the artifact to the target directory

copy:

src: "/var/lib/jenkins/workspace/HelloMaven-CI/target/HelloMaven-1.0-SNAPSHOT.jar"

dest: "/opt/deployment/HelloMaven.jar"

* 1. **Configure Your Ansible Inventory:**
  2. **Integrate Ansible into Your Jenkins/Azure Pipeline:**
     + Add a post-build (or post-release) step to execute the Ansible playbook: o ansible-playbook -i /path/to/hosts.ini /path/to/deploy.yml

1. **Using Azure Release Pipeline to Deploy to Azure App Services** 
   1. **Create a New Release Pipeline in Azure DevOps:**
      * Navigate to **“Pipelines > Releases”**.
      * Click **“New pipeline”** and select an empty job.
   2. **Link Your Build Artifact:**
      * Click **“Add an artifact”** and select your build pipeline as the source.
   3. **Add a Deployment Stage for Azure App Service:**
      * Create a new stage (e.g., Production). o Add the **“Azure App Service Deploy”** task. o Configure the task with your Azure subscription, target App Service, and the path to the artifact.
   4. **Integrate Key Vault (Optional for Managing Secrets):**
      * Create and link a Variable Group that pulls secrets from Azure Key Vault (refer to previous experiments).
      * Reference these secrets in your deployment tasks (e.g., connection strings).
   5. **Enable Continuous Deployment:**
      * In the release pipeline’s triggers, enable continuous deployment so that a new release is automatically created when a new artifact is available.
   6. **Run the Release Pipeline and Verify Deployment:**
      * Trigger the release pipeline (either manually or automatically).

o Verify that the application is deployed to your target environment (e.g., by browsing to the Azure App Service URL or checking the deployment directory on the target server).

**Step 5: End-to-End Pipeline Demonstration**

1. **Trigger a Complete Run:**

* + - Make a change in your code repository (e.g., modify a welcome message in your Java application) and commit it.
    - This should trigger the CI pipeline (Jenkins or Azure Pipelines), which builds, tests, and archives the artifact. o The artifact triggers the CD process (via Ansible or Azure Release), and the application is deployed automatically.

2. **Verify the Entire Workflow:**

* + - Check the CI pipeline output (build success and test results). o Confirm that the artifact is archived.
    - Review the CD pipeline logs (deployment success, any post-deployment notifications, etc.). o Optionally, log into the target environment and verify that the new version of your application is running. **Step 6: Discussion on Best Practices and Q&A Best Practices:**
  + **Pipeline as Code:**

Use YAML (or a Jenkinsfile) to define your build and release pipelines. This allows you to version control your pipeline configuration alongside your code.

* + **Automate Everything:**

Automate code checkout, builds, tests, artifact archiving, and deployments. Reduce manual interventions to minimize human error.

* + **Idempotence:**

Ensure that your deployment scripts (whether Ansible playbooks or release tasks) are idempotent—running them multiple times produces the same result.

* + **Secure Secrets Management:**

Use Azure Key Vault (or a similar tool) to securely store sensitive data (e.g., API keys, connection strings) and reference these values in your pipelines.

* + **Monitoring and Logging:**

Integrate logging and monitoring into your pipeline. Review test reports, deployment logs, and set up notifications for build failures.

* + **Modular and Scalable Design:**

Break down your pipeline into clear stages (checkout, build, test, deploy) and design it to handle multi-environment deployments (development, staging, production).

* + **Continuous Improvement:**

Regularly review and refine your pipeline. Use metrics and feedback to optimize build times, reduce failures, and ensure high-quality releases.

**Q&A Discussion Points:**

* + **Troubleshooting Pipeline Failures:**

What are common reasons for build/test failures, and how can you diagnose issues from the logs?

* + **Handling Rollbacks:**

What strategies can be implemented for rolling back deployments if a release fails in production?

* + **Scaling Pipelines:**

How do you handle scaling the pipeline for larger projects or multiple microservices?

* + **Integrating Additional Tools:**

What additional tools (e.g., code quality analyzers, security scanners) could be integrated into this pipeline?

* + **Real-World Challenges:**

Discussion of challenges faced in actual DevOps implementations and strategies to overcome them.

*Open the Floor for Questions:*

Encourage participants to ask questions regarding any part of the pipeline—from technical configuration details to high-level best practices.