**What is Container?**

A container is a lightweight, standalone, executable package that includes everything required to run a piece of software, such as code, libraries, and plugins.

Containers are isolated from one another, meaning each container operates independently with its own libraries, dependencies, and configurations. This isolation allows containers to run securely and reliably, without interference from other containers on the same system.

Because containers package all necessary components (like binaries and libraries), they can operate consistently across different environments—whether on a developer’s laptop, in testing, or in production. This portability helps eliminate the common "it works on my machine" problem seen in traditional software deployment.

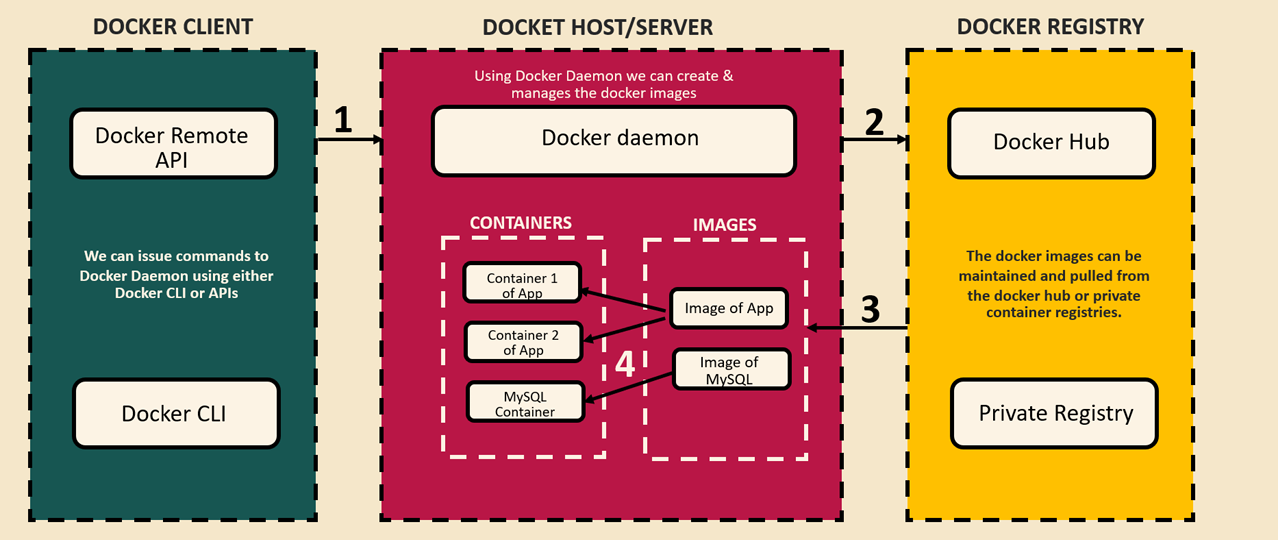
**What is Containerization?**

Containerization is the process of packaging software and its dependencies into a container. This approach ensures consistency across different environments, whether in development, testing, or production, by isolating applications and their configurations.

**What is Docker?**

Docker is a popular open-source platform for developing, shipping, and running applications in containers. Docker provides tools and a runtime environment for creating and managing containers, along with a registry to store and share container images.

**Docker Architecture**



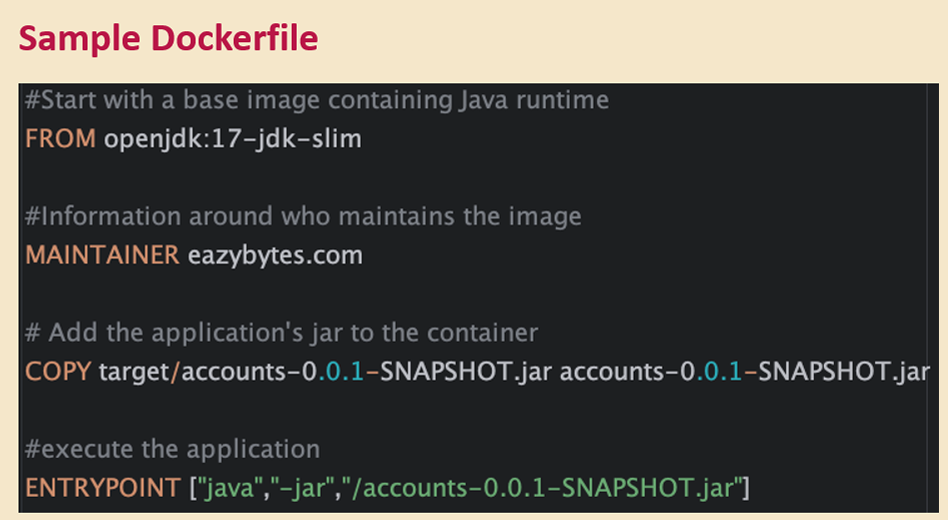
Docker architecture is composed of three primary components:

1. **Docker Client**: The Docker Client allows users to interact with the Docker Host (or Server) to issue instructions on containerizing applications. It includes two main parts:
   1. **Docker CLI**: The command-line interface used to execute Docker commands.
   2. **Docker Remote API**: An API that facilitates communication between the Docker Client and Docker Daemon on the host.
2. **Docker Server/Host**: The Docker Server, or Host, houses a Docker Daemon process that runs continuously in the background, accepting commands from the Docker CLI. For example, when instructed, it can create Docker images by packaging applications (such as Spring Boot, Maven, or microservices) along with their dependencies. This resulting package, known as a Docker image, can then be used to create Docker containers. The Docker Server manages the full lifecycle of these images and containers, handling tasks like creation, deployment, and deletion.
3. **Docker Registry**: Once a Docker image is finalized and ready for deployment, it can be pushed to a Docker Registry—a remote repository where images are stored and managed. Docker Hub, for instance, is a widely used registry where images can be stored securely and retrieved for deployment across various environments (e.g., development, testing, production). The Docker Registry functions similarly to code repositories like GitHub, providing centralized storage and accessibility for Docker images.

There are three main methods to create Docker images.

1. Dockerfile

Here, we’ll cover the first method using a **Dockerfile**, which requires manually specifying all configurations.

1. **Build the Application**  
   Run the command “mvn clean install“ from the directory containing the pom.xml file. This command will build the project and create a fat JAR file in the target folder.
2. **Write Dockerfile Instructions**  
   Define the necessary instructions in the Dockerfile to configure the Docker image. 
3. **Build the Docker Image**  
   Navigate to the directory where the Dockerfile is located and execute:

“docker build -t eazybytes/accounts:s4 .”

This command will create a Docker image with the name eazybytes/accounts:s4.

1. **Run the Docker Container**  
   To start a container based on the newly created image, run:

“docker run -p 8080:8080 eazybytes/accounts:s4”

This command launches the container with port mapping, making the application accessible on port 8080.

1. Buildpacks
2. Google Jib

**Configuration Management in Microservices**

In microservices, managing configurations across multiple environments can be challenging. Each environment—such as development, staging, and production—requires different configurations. If we bundle all configurations together, it would necessitate creating separate Docker images for each environment, which is not ideal. A best practice is to use the same Docker image across all environments, including production.

Additionally, some properties, like database credentials, need to be injected into the microservice at startup, as they should not be hard-coded in configuration files or embedded in business logic. Maintaining a single configuration file for each environment can quickly become complex and unmanageable.

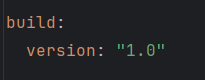
To address these challenges, there are three effective methods for configuration management in microservices:

1. **Configuring Spring Boot with Environment-Specific Profiles**: Use profiles to load environment-specific properties directly within Spring Boot.
2. **External Configuration with Spring Boot**: Externalize configurations outside the application, allowing them to be customized per environment without modifying the code.
3. **Using a Centralized Configuration Server with Spring Cloud Config**: Implement a dedicated configuration server to manage all microservice configurations centrally, simplifying maintenance and ensuring consistency across environments.

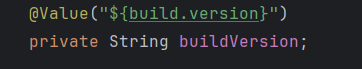
To read properties from configuration files in a Spring Boot application, we have the following options:

1. **Using the @Value Annotation**  
   The @Value annotation allows us to inject property values directly from the application.yml or application.properties file. Specify the property’s path in the annotation.  
   **Example:**  
   If the build version is defined in application.yml as build.version: 1.0, you can access this value by annotating a variable with @Value("${build.version}").

Application.yml



AccountsController.java



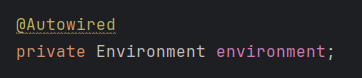


1. **Using the Environment Interface**

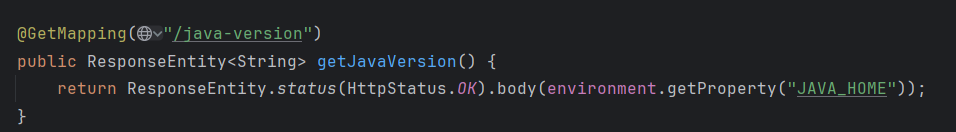
The **Environment interface** provides a programmatic way to access property values in Spring Boot applications. This approach is helpful when you need dynamic access to properties, allowing you to retrieve them at runtime without hardcoding property paths.

**Example:**  
Suppose we want to retrieve the Java version installed on the system. We can achieve this by using the Environment interface as follows:

* 1. **Declare an Environment Variable**  
     Inject the Environment object as a dependency in your component or service.



* 1. **Retrieve the Property Value**  
     Use the getProperty() method to fetch the property by its name.



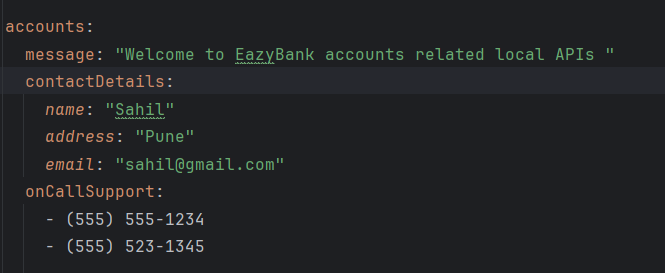
In this example, the java.version property provides the Java version. This approach makes configuration management more flexible and enables dynamic property retrieval based on application logic.

1. **Using the @ConfigurationProperties Annotation**

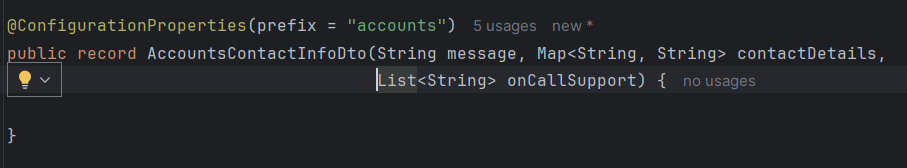
Unlike the @Value annotation and Environment interface, which are suitable for retrieving single properties, the **@ConfigurationProperties** annotation allows us to bind multiple related properties into a single class (DTO). This approach is particularly useful when you have a group of properties that you want to manage together, making your configuration code cleaner and more organized.

To use @ConfigurationProperties, follow these steps:

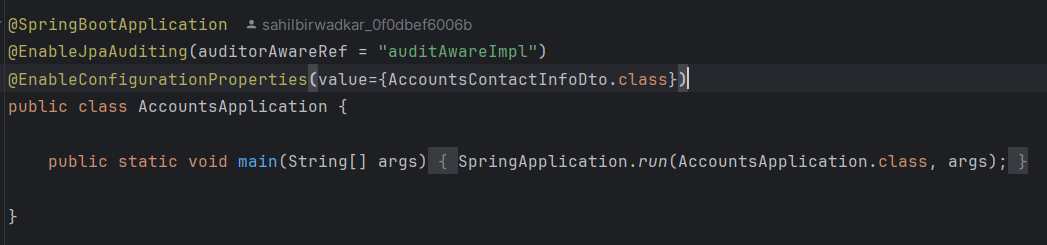
1. **Define Properties in application.yml**  
   Organize related properties under a common prefix. For instance, if we want to configure account-related settings, we could define them like this:



1. **Create a DTO Class**  
   Create a class that will hold the properties. Annotate it with @ConfigurationProperties and specify the common prefix (e.g., "accounts") that you used in application.yml.



1. **Enable Configuration Properties in the Application**  
   Make sure @EnableConfigurationProperties is added in your main application class or that @ConfigurationProperties classes are annotated with @Component, so Spring Boot can detect and bind them automatically.



So far, we've defined properties that are common across all environments, without any environment-specific configuration. To handle environment-specific settings, Spring Boot introduces the concept of **profiles**.

**Spring Profiles** allow us to group configurations and properties based on different environments, such as development, testing, and production. By creating distinct profiles (e.g., dev, QA, and prod), we can specify unique values for each environment. Spring Boot will activate the appropriate profile based on the current execution context, ensuring the right configuration is loaded for the environment in use.

With profiles, we can configure our application to run seamlessly across multiple environments with the same code base but different settings. Additionally, profiles allow us to control bean creation, enabling beans to be instantiated only when a specific profile is active.

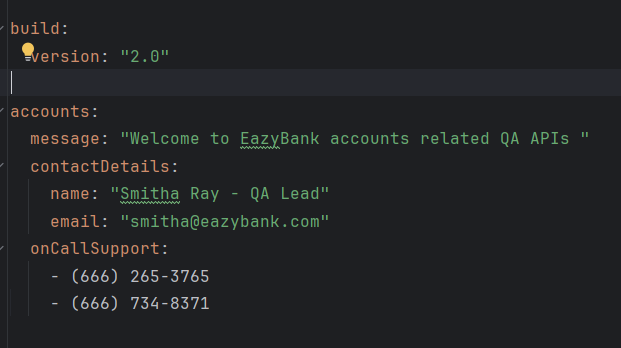
This powerful feature in Spring Boot ensures that both application properties and bean lifecycles can be customized based on the active profile, making it easier to manage environment-specific configurations in a clean, flexible way.

To demonstrate how profiles work, we'll create two additional configuration files: application\_qa.yml and application\_prod.yml, alongside our default application.yml file.

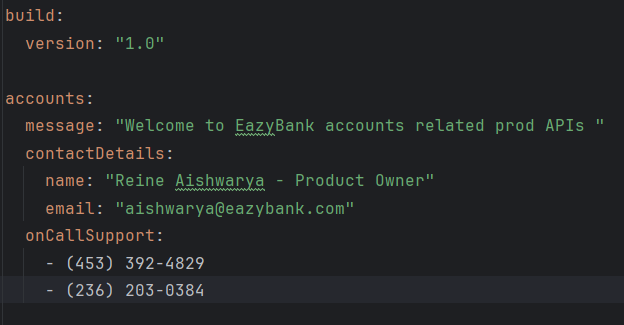
In this setup:

* Common properties like the port number and database configuration will remain in the main application.yml file, as they are shared across environments.
* Environment-specific properties, such as build version and account-related settings, will be defined in application\_qa.yml and application\_prod.yml.

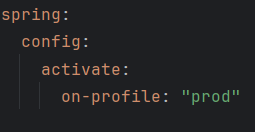
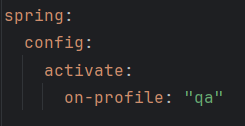
application\_qa.yml



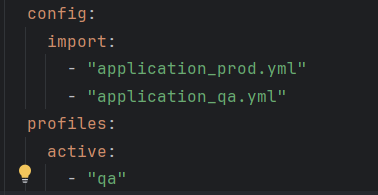
application\_prod.yml



To determine which YAML file is loaded based on the active profile, we can specify the profile activation condition within each configuration file. This setup allows Spring Boot to activate the appropriate configuration automatically based on the profile.

application\_prod.yml application\_qa.yml

By default, Spring Boot loads the properties from application.yml. To activate a specific profile (for instance, to test with the QA configuration), we need to specify the active profile in our local application.yml file as shown below:

Here:

* config.import specifies a list of configuration files that can be loaded.
* profiles.active determines which profile (and corresponding configuration file) Spring should use.

This setup allows Spring Boot to dynamically load the appropriate configuration file based on the active profile, making it easier to switch between environments during development and testing.

**Managing Configuration in Microservices with Spring Cloud**

**1) Reading Configuration From Classpath**

**Step 1: Create a Spring Boot Project for Config Server**

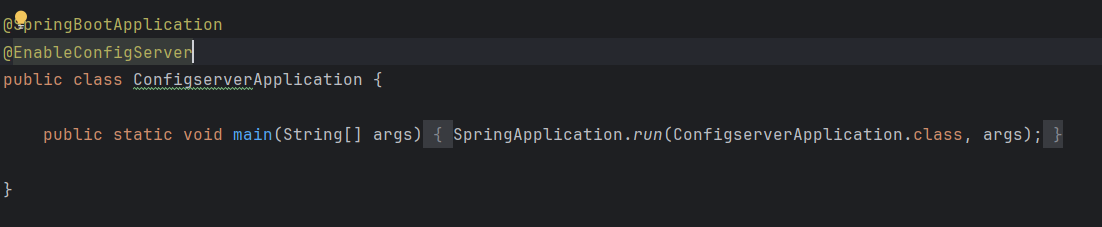
* Start by creating a new Spring Boot project and name it “ConfigServer.” This project will serve as a dedicated configuration server to manage the configurations of all microservices.

**Step 2: Add Dependencies**

* In the pom.xml file, include dependencies for Spring Cloud and Spring Boot Actuator.

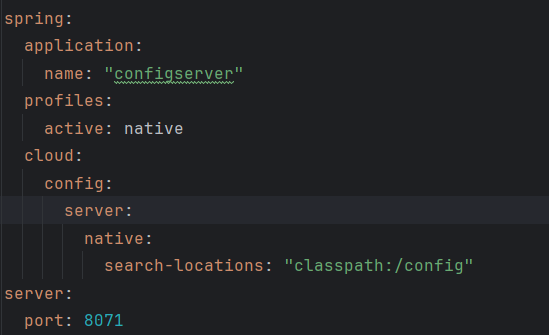
**Step 3: Enable Config Server**

* In the ConfigServer application, add the @EnableConfigServer annotation. This annotation designates the application as a configuration server responsible for managing the configurations of other microservices.



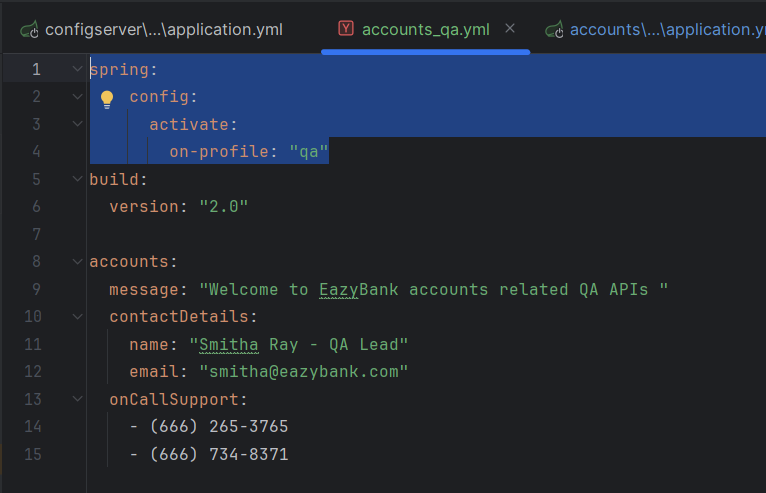
**Step 4: Configure application.yml**

* Set the application.yml file with a port number (e.g., 8071) and an application name (configserver). This name will help other microservices identify which server to connect to for configuration.
* Add profile configurations by setting the active profile to native. This informs Spring Boot to use the native profile for local property management.
* Create a config folder and add environment-specific application.yml files for each microservice. Name each file according to the microservice and environment. For instance, if the microservice is named accounts, the configuration file for the production environment should be named accounts-prod.yml, while the file for the QA environment should be accounts-qa.yml.
* Specify the file path where the properties are stored. For example, if the YAML configuration files for different environments (production/QA) are stored under classpath, add this location in the search-locations variable. This setup enables the configuration server to locate and serve the necessary properties to each microservice.



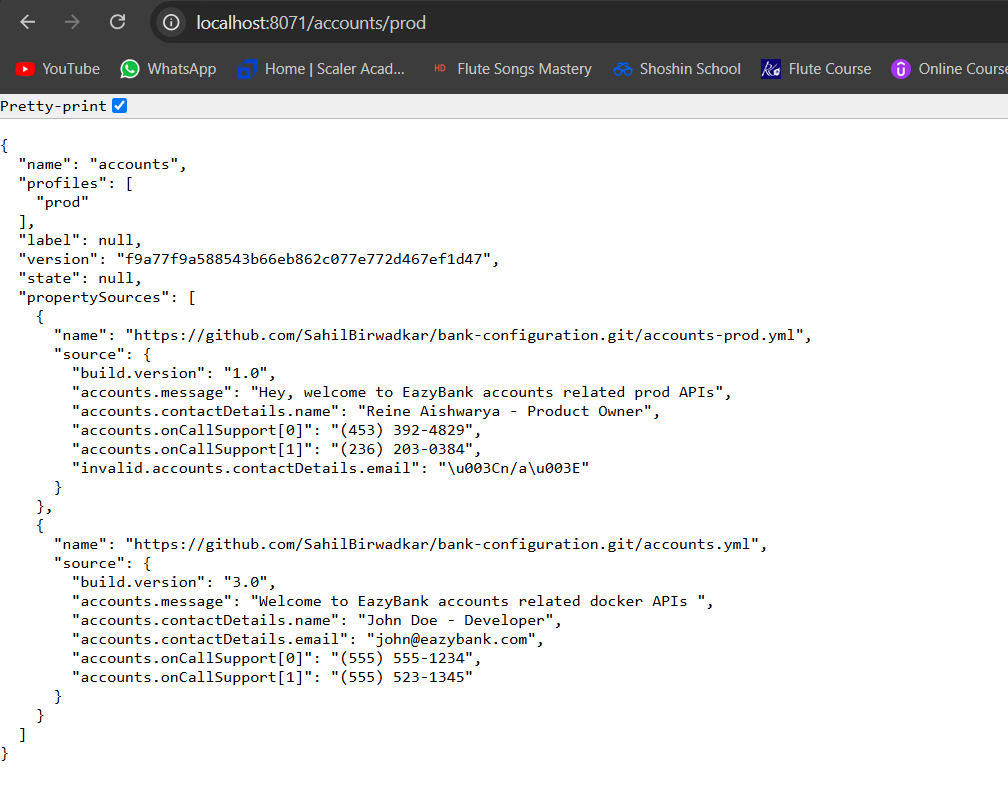
**Step 5: Remove Unnecessary Configuration from Microservices**

* In this step, remove any redundant configuration settings from all microservices, such as spring.config.activate.on-profile: “qa”. Since we are now using Spring Cloud Config to centralize and manage configurations, these individual settings within each microservice are no longer needed.
* By removing these configurations, we ensure that each microservice relies solely on the configuration provided by the central Config Server, streamlining the configuration process across all services.
* Keep environment-specific properties like build version and contact information, as these will vary based on the environment and should remain unique to each service.



**Step 6: Run the Config Server**

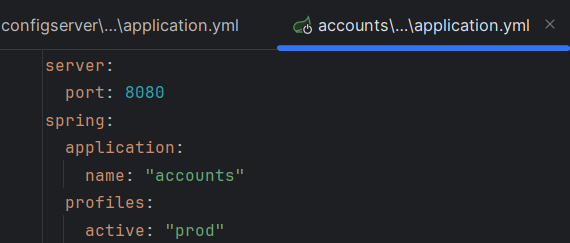
* Start the Config Server on port 8071.
* To verify the setup, navigate to http://localhost:8071/accounts/prod in your browser. This URL requests the configuration details (such as contact-info) for the accounts microservice in the prod environment.
* If configured correctly, this endpoint should return the configuration details specific to the requested microservice and environment.



**Step 7: Add Config Server Details to Other Microservices**

* In the application.yml file of each microservice, add the configuration details of the Config Server. This setup ensures that each microservice connects to the Config Server to fetch its configuration upon startup.
* To control startup dependencies, ensure the Config Server starts before other microservices by specifying it in the configuration. You can make this connection optional by adding optional: true to the spring.config.import statement, allowing microservices to start even if the Config Server is temporarily unavailable.



* Specify a unique application name for each microservice in the application.yml file, matching the file name format used in the config folder of the Config Server. For example, for the accounts microservice, set application.name: accounts so the Config Server can correctly identify and serve the matching configuration file (like accounts-prod.yml or accounts-qa.yml) based on the environment.

**2) Reading Configuration From Git**

**Step 8: Push All Configuration to GitHub**

* Push all configuration files to a private GitHub repository to centralize and secure access to configuration files. This setup prevents developers from accessing configuration files locally, keeping sensitive information safe.
* In the application.yml file of the Config Server application, configure the following properties:
  + Activate Git Profile: Set the spring.profiles.active property to git to enable Git-based configuration.
  + Git Configuration: In the spring.cloud.config.server section, specify git as a child property. Define the uri as the URL of the GitHub repository where the configuration files are stored. Also, set the branch property to the branch containing the configuration files, such as main or prod.

