**Managing Configuration in Microservices with Spring Cloud**

**1) Reading Configuration from Classpath**

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

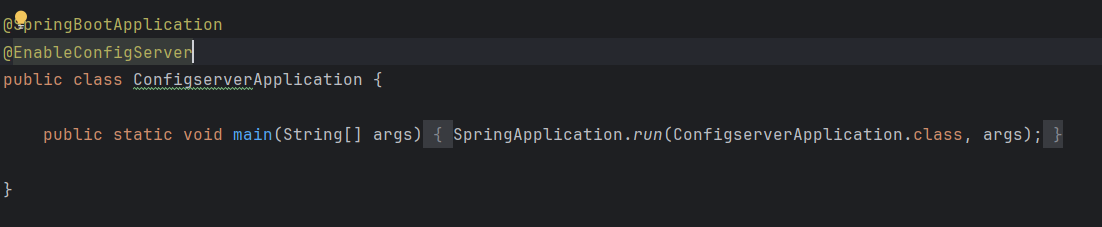
* Start by creating a new Spring Boot project and name it “ConfigServer.” This project will serve as a dedicated configuration server to manage the configurations of all microservices.

**Step 2: Add Dependencies**

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

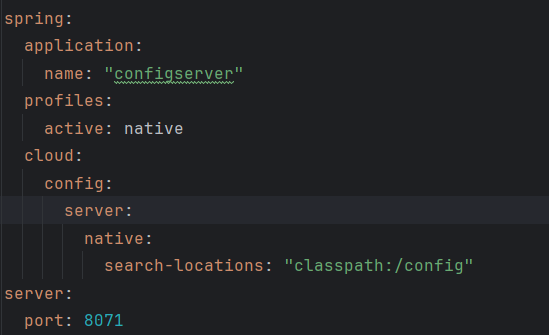
**Step 3: Enable Config Server**

* 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.



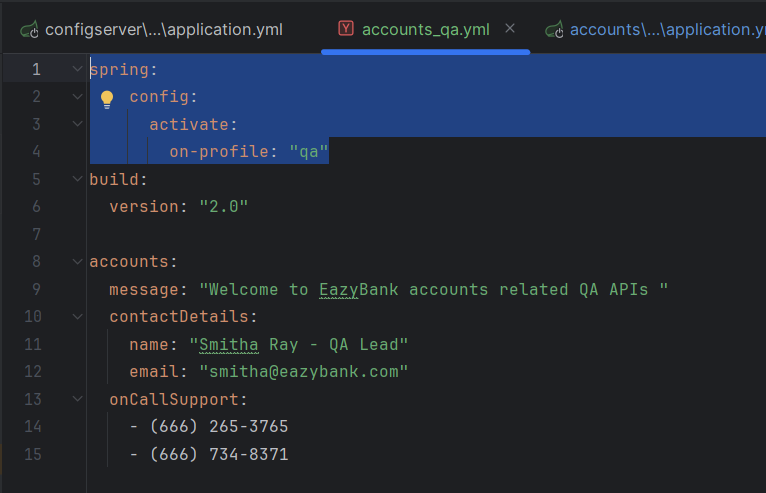
**Step 4: Configure application.yml**

* 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.



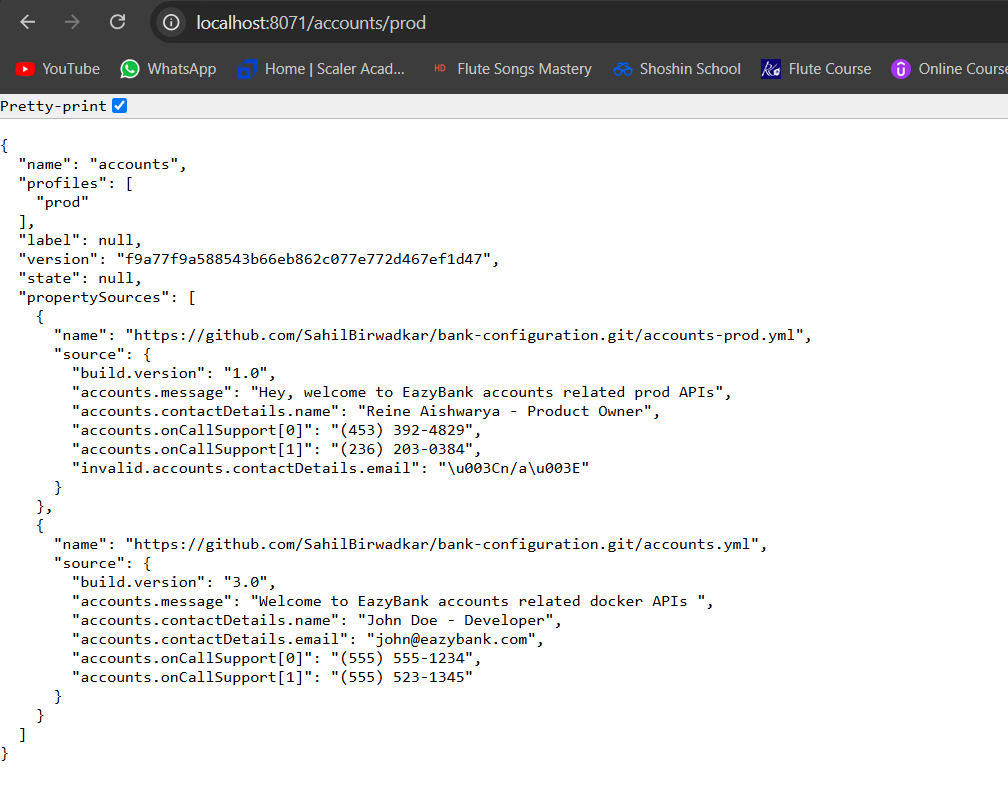
**Step 5: Remove Unnecessary Configuration from Microservices**

* 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**

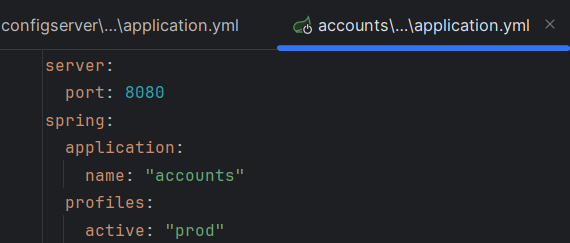
* Start the Config Server on port 8071.
* To verify the setup, navigate to http://localhost:8071/accounts/prod in your browser. This URL requests the configuration details (such as contact-info) for the accounts microservice in the prod environment.
* If configured correctly, this endpoint should return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

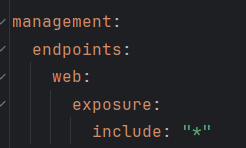
**Managing Dynamic Configuration Refresh in Spring Cloud**

After successfully centralizing and securing configuration management using GitHub and Spring Cloud, a new challenge arises: applying configuration updates without repeatedly restarting the Config Server. To address this, we can use Spring Boot Actuator for runtime configuration refresh. Here are three effective methods:

**1) Refresh Configuration at Runtime Using the /refresh Actuator Endpoint**

To dynamically refresh configuration without restarting the application, we can use the /refresh endpoint provided by Spring Boot Actuator. Follow these steps:

* **Enable the /refresh Endpoint**: In the microservice’s application.yml (e.g., accounts microservice), expose the refresh endpoint by adding the following configuration:

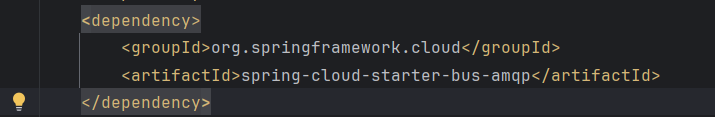


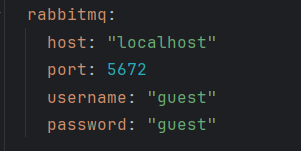
* **Trigger Configuration Refresh**: After making changes to the configuration file in GitHub, send a POST request to the /refresh endpoint (e.g., http://localhost:8080/actuator/refresh), where 8080 is the port number of the microservice (such as accounts).
* **Effect of Refresh**: This endpoint checks for changes in the configuration specific to the current microservice. If updates are detected, it applies them without needing to restart the application. After calling /refresh, the next request to the application will reflect the updated configuration values.
* In this method we’ve to call /refresh api in each microservice. And that can be tedious job if we’ve 100 Microservices. And hence there’s another approach we can use which is as below

**2) Refresh Configurations at Runtime Using Spring Cloud Bus**

To avoid manually calling the /refresh endpoint on each microservice (a time-consuming task when managing many microservices), Spring Cloud Bus offers a solution for broadcasting configuration changes across all services. This setup ensures that configuration updates are applied automatically and simultaneously across the system.

* **Spring Cloud Bus Integration**: Spring Cloud Bus utilizes a message broker (e.g., RabbitMQ or Kafka) to propagate configuration changes to all microservices. By listening for state changes, Spring Cloud Bus can broadcast a refresh event whenever a configuration update is detected.
* **Automatic Refresh Across Microservices**: When you push an update to a configuration file in GitHub, a refresh event triggered in the Config Server will automatically notify all subscribed microservices. This approach eliminates the need to call the /refresh endpoint manually on each service, as all services listen for broadcasted refresh events.
* **Configuration Setup:**

1. **Dependencies:** In the pom.xml file of both the Config Server and all microservices, add the necessary Spring Cloud Bus dependencies: 
2. **Message Broker Configuration**: In application.yml for both the Config Server and each microservice, configure the message broker (RabbitMQ/Kafka) settings:

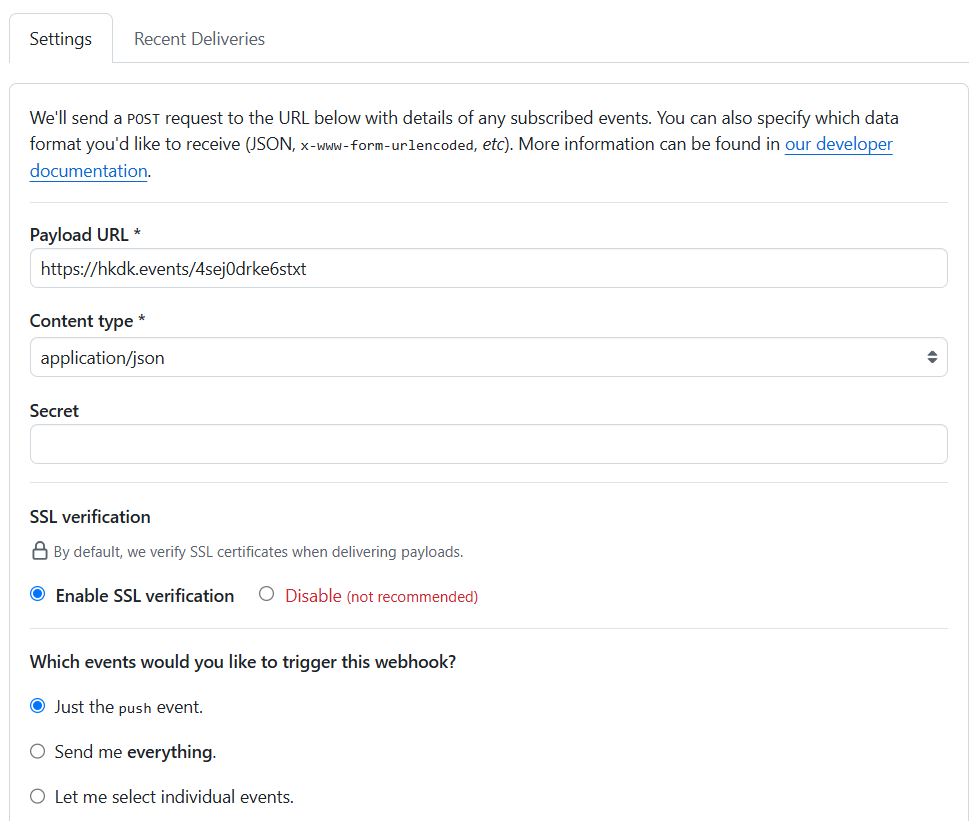


1. **Trigger Refresh**: After updating configurations in GitHub, a single /refresh call on the Config Server (or webhook event) will broadcast the configuration changes to all microservices automatically.

**3) Refresh Configuration at Runtime Using Spring Cloud Bus & Spring Cloud Config Monitor**

Spring Cloud Config Monitor simplifies configuration updates by listening for webhook events from GitHub (or any other version control system). This setup automatically triggers a refresh across all microservices whenever configuration files are updated in the repository, minimizing the need for manual refresh calls.

* **How Spring Cloud Config Monitor Works:** The Config Monitor listens for webhooks from your GitHub repository. When you commit a change to any configuration file, the webhook sends a notification to the Config Server, which then triggers a configuration refresh for all microservices via Spring Cloud Bus.
* **Configuration Requirements:**
  1. **Add Required Dependencies:** In the pom.xml file of the Config Server, include both Spring Cloud Bus and Spring Cloud Config Monitor dependencies:
  2. **Webhook Setup**: Configure a webhook in your GitHub repository to notify the Config Server whenever configuration changes occur. Set the webhook’s **Payload URL** to the monitor endpoint of your Config Server (e.g., http://your-config-server-url/monitor). Note that if you’re developing locally, this URL must be accessible from the internet.
     + - **Using Hookdeck for Local Development**: Hookdeck provides an HTTPS URL that can forward incoming webhook requests to your local server, making it easier to work with local environments. Create an HTTPS URL through Hookdeck and specify it as the payload URL in GitHub. Hookdeck will forward GitHub notifications to your local Config Server, even if it’s running on localhost.
       - **Webhook Payload Configuration**: In the GitHub webhook settings, set the content type to application/json, and select Just the push event to monitor for changes whenever you push updates to the configuration files.

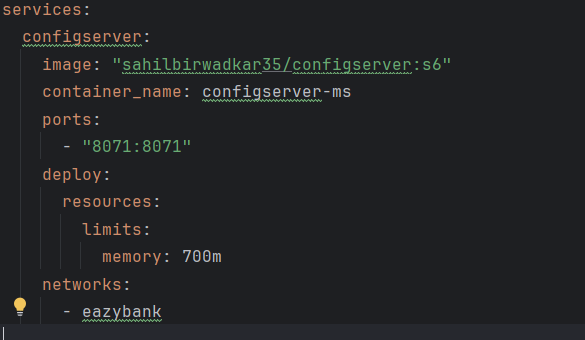


* 1. **Automatic Trigger on Changes**: Once the webhook is configured, any commit that modifies configuration files in the GitHub repository will automatically trigger the Config Monitor, which will then instruct Spring Cloud Bus to propagate these changes across all microservices. This way, configurations are refreshed instantly without requiring any manual refresh requests, ensuring all services remain up-to-date.

**Docker Compose**

To containerize your microservices, including the Config Server, and manage them with Docker Compose, you’ll need to ensure that each microservice is configured properly within a Docker environment and that dependencies, like the Config Server, are correctly referenced. Here’s a step-by-step guide to setting this up based on your requirements.

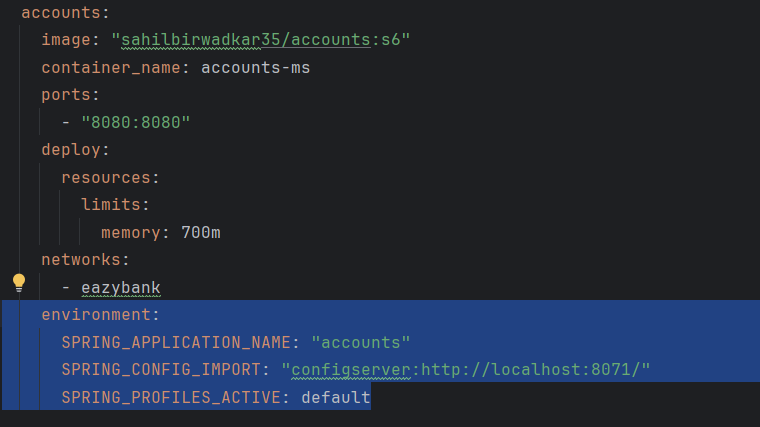
**Step 1: Add Config Server Properties in docker-compose.yml**

First, define the Config Server in docker-compose.yml with appropriate configuration settings, including port mappings, environment variables, and health checks.

**Step 2: Add Configuration Server URL in Each Microservice**

In each microservice’s configuration within docker-compose.yml, set up environment variables for SPRING\_APPLICATION\_NAME and SPRING\_PROFILES\_ACTIVE, and configure them to point to the Config Server. Instead of using localhost, use the configserver service name defined in Docker Compose.

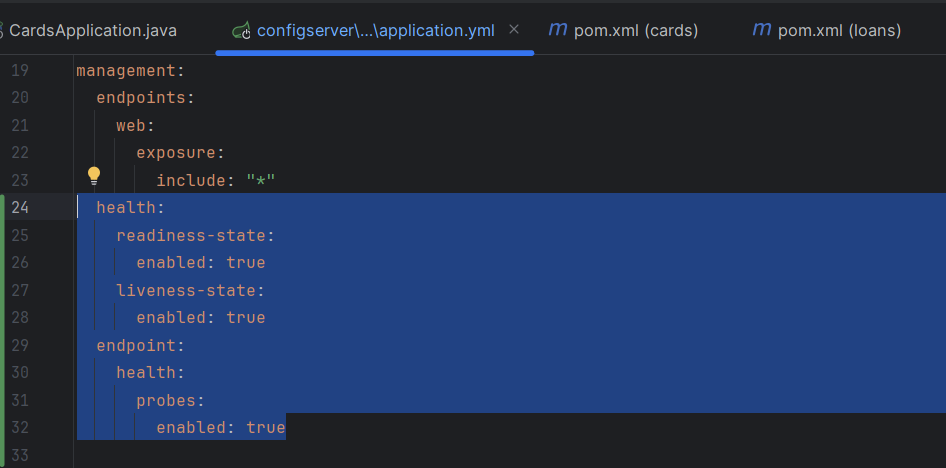
For example, for a microservice called accounts:



**Explanation**

* **SPRING\_PROFILES\_ACTIVE=docker**: Sets the active profile to docker, ensuring that the configuration properties are compatible with Docker Compose.
* **SPRING\_CONFIG\_IMPORT=optional:configserver:http://configserver:8071/**: Specifies the Config Server’s URL, using the configserver service name instead of localhost. This makes the connection work in a Docker network, where services communicate by name.
* **depends\_on with condition: service\_healthy**: Ensures the accounts service waits until configserver is healthy before starting.

**Step 3: Add Health Check to Each Microservice**

Add health checks to your microservices to monitor their readiness and liveness, enabling Docker to determine their status. You can configure each microservice to have a similar healthcheck section as the Config Server.

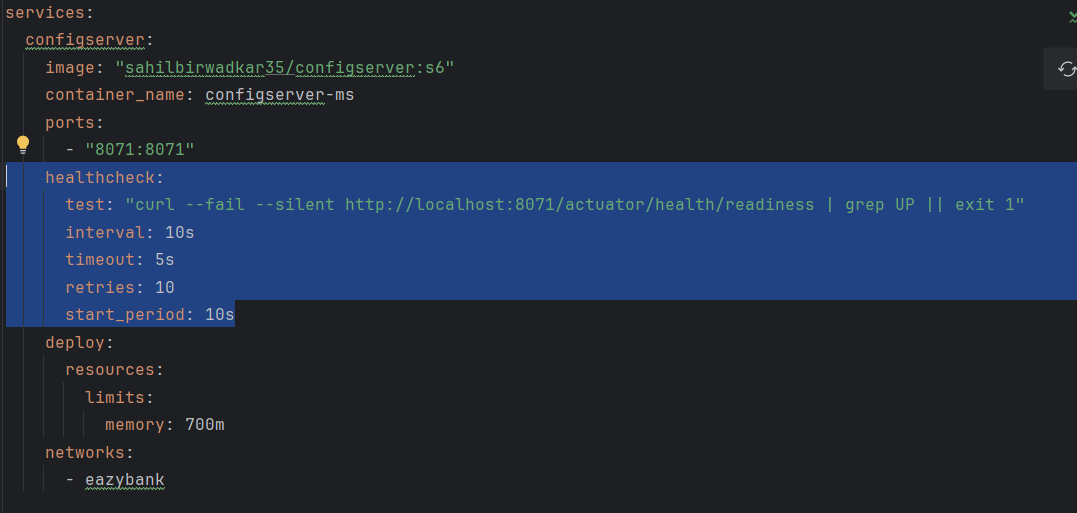
**Testing Config Server Health**

You can check if the Config Server is running and healthy by calling:

* http://localhost:8071/actuator/health/readiness
* http://localhost:8071/actuator/health/liveness

This will confirm whether the Config Server and other microservices are up and ready to serve requests.

By following these steps, your microservices and Config Server are containerized and properly orchestrated with Docker Compose, ready for local testing and deployment into environments like Kubernetes.



Now we’ve added above piece of code for healthcheck under docker-compose.yml file to check readiness of the configserver.

**Explanation of healthcheck Configuration**

1. **test: "curl --fail --silent http://localhost:8071/actuator/health/readiness | grep UP || exit 1"**
   * This command checks the container's readiness by making an HTTP request to the endpoint http://localhost:8071/actuator/health/readiness.
   * curl --fail --silent:
     + --fail: curl will exit with a non-zero status if the HTTP request fails (e.g., if the server returns a 4xx or 5xx status).
     + --silent: suppresses any output except for errors.
   * The response is piped to grep UP, searching for the text "UP" in the response (typically, a healthy status from a Spring Boot actuator endpoint).
   * || exit 1: If "UP" is not found, the command exits with status 1, marking the health check as failed.

2. **interval: 10s**

* Docker waits 10 seconds between each health check attempt. This is how often Docker checks the health status.

3. **timeout: 5s**

* Each health check attempt has a timeout of 5 seconds. If the health check does not complete within this time, Docker marks it as failed.

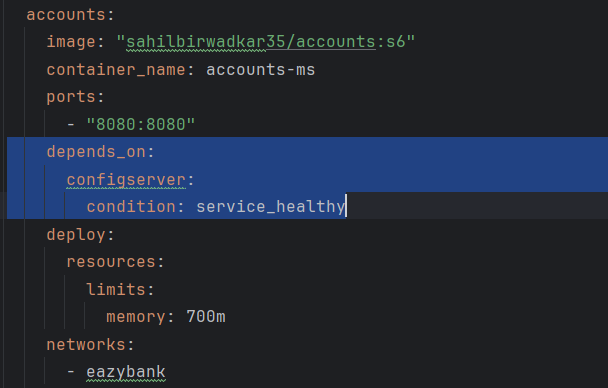
4. **retries: 10**

* Docker will retry the health check 10 times before considering the container unhealthy. If the health check fails consecutively 10 times, the container is marked as unhealthy.

5. **start\_period: 10s**

* This is a grace period of 10 seconds after the container starts, during which Docker will not consider failed health checks as actual failures. This gives the container time to initialize before checks are enforced.

After configuring and checking all readiness and liveness of the config server, we’ve to specify the condition under other microservice as below:



With the health check in place, you can configure the accounts service to wait for configserver to be healthy before starting. This is done using the depends\_on configuration with the condition: service\_healthy.

To optimize your Docker-Compose setup, you can use a common\_config.yml file to define shared configurations like networks, depends\_on, memory\_size, etc., which can then be referenced in docker-compose.yml. This reduces redundancy and makes your configuration cleaner.

Here’s how to set this up:

**Step 1: Create common\_config.yml**

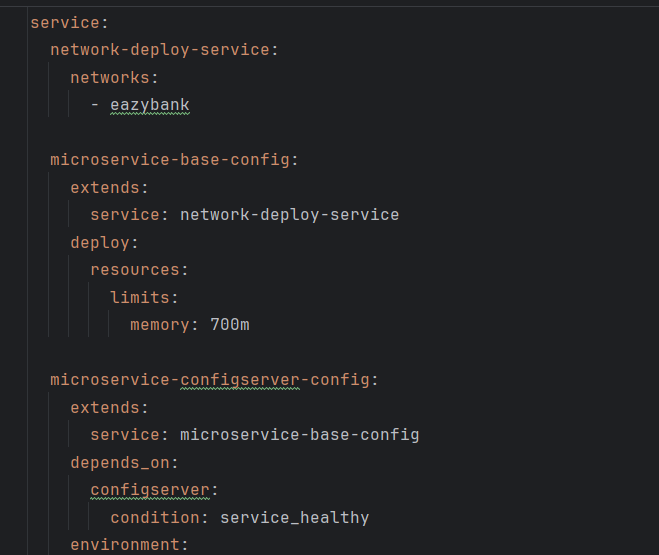
In common\_config.yml, define common settings for each service that should be reused. You can also use YAML anchors (& for anchor and \* for referencing) to define configurations once and reuse them.

This file defines:

* A common network (app\_network) shared by all services.
* Common resource limits and memory reservations.
* A depends\_on condition that references configserver to ensure it is healthy before other services start.

**Step 2: Reference common\_config.yml in docker-compose.yml**

* In docker-compose.yml, you can import and reuse configurations from common\_config.yml to avoid repetition.



Sample Common\_Config.yml

**Explanation**

* **\*common\_settings**: Imports all properties defined as a common settings in common\_config.yml. This means the accounts service will inherit networks, depends\_on, and memory configurations defined in common\_config.yml file.
* **Service-specific properties**: Any specific properties, like image, environment, and healthcheck, are defined under each service as needed.