

codefresh

Table of Content

- [What is Codefresh?](#)
- [Features and Benefits of Codefresh](#)
- [Understanding CI/CD](#)
- [Introduction to Continuous Integration \(CI\) and Continuous Deployment \(CD\)](#)
- [Importance of CI/CD in Modern Software Development](#)
- [Create Codefresh account](#)
- [Setting Up Git Integration with GitHub](#)
- [Runtime Environment in Codefresh](#)
- [Simple Pipeline](#)
- [Triggering Pipelines](#)
- [Advanced Pipeline Configuration](#)
- [Conditional Execution of Steps](#)
- [Conditional Steps and Branching Logic in Codefresh Pipeline](#)
- [Post Step Operations](#)
- [Setting Up Docker](#)
- [Docker Integration](#)

What is Codefresh?

Codefresh is a cutting-edge CI/CD platform specifically designed for Kubernetes and microservices-based applications. Unlike traditional CI/CD tools, Codefresh integrates seamlessly with Kubernetes, allowing developers to build, test, and deploy applications faster and with more flexibility.

Codefresh's unique selling points include:

- **Kubernetes-Native Pipelines:** Codefresh pipelines are built from the ground up with Kubernetes in mind, allowing for native integrations and optimizations.
- **Rapid CI/CD Workflows:** With parallel step execution and intelligent caching, Codefresh ensures that pipelines run as quickly as possible.
- **GitOps Integration:** Codefresh offers deep GitOps capabilities, enabling automated deployment and rollback of applications based on Git repository state.

Features and Benefits of Codefresh

- [Features](#)
- [Benefits](#)

Features

- **Visual Pipeline Editor:** Codefresh provides a user-friendly visual editor for creating and managing pipelines, making it easy to design complex workflows without needing to write YAML files from scratch.

- **Built-In Docker Support:** The platform has robust support for Docker, allowing you to build, test, and deploy Docker containers efficiently.
- **Helm Integration:** Codefresh offers out-of-the-box support for Helm, the popular Kubernetes package manager, simplifying Kubernetes application management.
- **Monitoring and Debugging:** With real-time logs, detailed metrics, and alerts, Codefresh provides excellent tools for monitoring and debugging your CI/CD pipelines.
- **Pre-Built Plugins:** Codefresh comes with numerous plugins and integrations for popular tools such as Slack, JIRA, and Datadog, which can be easily integrated into your pipelines.

Benefits

- **Speed:** Codefresh pipelines are optimized for speed, allowing for faster builds and deployments. This leads to shorter feedback loops and quicker releases.
- **Scalability:** The platform is built to handle large-scale microservices environments, making it ideal for enterprises and growing teams.
- **Ease of Use:** The visual pipeline editor and pre-built templates make Codefresh accessible even for teams with limited DevOps experience.
- **Flexibility:** Whether you're deploying to Kubernetes, a Docker registry, or using Helm, Codefresh offers the flexibility to fit various use cases.
- **Security:** Codefresh ensures that your pipelines are secure, with features like secret management and role-based access control.

Understanding CI/CD

- [What is CI/CD?](#)
- [Continuous Integration](#)
- [Continuous Deployment](#)

What is CI/CD?

CI/CD stands for Continuous Integration and Continuous Deployment. It is a set of practices that aims to improve software delivery by automating the integration of code changes, running tests, and deploying updates to production environments.

Continuous Integration

- **Definition:** CI is a development practice where developers frequently integrate their code into a shared repository. Each integration is automatically verified by building the application and running automated tests to detect errors as quickly as possible.
- **Goal:** The goal of CI is to provide rapid feedback on the state of the codebase, ensuring that any integration issues are identified and resolved early in the development process.
- **Tools:** Common CI tools include Jenkins, Travis CI, CircleCI, and, of course, Codefresh.

Continuous Deployment

- **Definition:** CD is an extension of CI that automates the deployment of code changes to production. It ensures that every code change passes through automated tests and, if successful, is automatically deployed to production.

- **Goal:** The goal of CD is to minimize the manual steps required to deploy new features, fixes, and updates, enabling faster and more reliable software releases.
- **Tools:** Popular CD tools include Jenkins, Codefresh, GitLab CI/CD, and Spinnaker.

Importance of CI/CD in Modern Software Development

- [Speed and Efficiency](#)
- [Quality and Reliability](#)
- [Collaboration and Transparency](#)
- [Scalability and Flexibility](#)

Speed and Efficiency

- **Faster Releases:** CI/CD allows teams to release software updates more frequently and with less effort, reducing time-to-market.
- **Continuous Feedback:** Automated testing and integration provide continuous feedback on the codebase, allowing developers to catch and fix issues early.

Quality and Reliability

- **Automated Testing:** By integrating automated tests into the CI/CD pipeline, teams can ensure that new code does not introduce bugs or regressions.
- **Consistent Deployments:** CD pipelines ensure that deployments are consistent across environments, reducing the risk of deployment errors.

Collaboration and Transparency

- **Improved Collaboration:** CI/CD fosters a culture of collaboration by integrating code from all team members frequently, making it easier to work together and avoid integration conflicts.
- **Transparency:** CI/CD tools provide visibility into the state of the codebase, the progress of builds, and the status of deployments, making the development process more transparent.

Scalability and Flexibility

- **Scalable Processes:** CI/CD pipelines can scale with your application, allowing you to handle growing codebases and teams without a drop in efficiency.
- **Flexibility:** CI/CD tools like Codefresh offer the flexibility to integrate with various tools and platforms, enabling teams to tailor their pipelines to their specific needs.

Create Codefresh account

This quick start guides you through creating an account in Codefresh:

After you select the IdP (identity provider), Codefresh requests permission to access your basic details, and for Git providers, to access your Git repositories. The Permissions window that is displayed differs according to the IdP selected.

The permissions requested by Codefresh are needed in order to build and deploy your projects.

Codefresh currently supports the following IdPs:

- GitHub
- Bitbucket
- GitLab
- Azure
- Google
- LDAP

If you need an IdP that is not in the list, please [contact us](#) with the details.

NOTE

For Git repositories, the login method is less important, as you can access Git repositories through [Git integrations](#), regardless of your sign-up process.

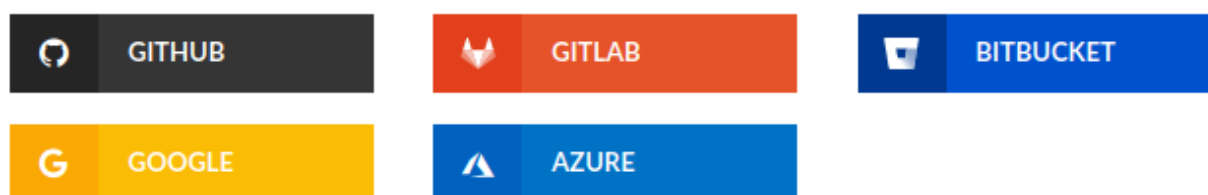
If you have multiple sign-up methods, as long as you use the same email address in all the sign-ups, Codefresh automatically redirects you to the account dashboard.

1. Select the identity provider (IdP) to use:
 1. Go to the [Codefresh Sign Up](#) page.
 2. Select the IdP for sign-up.



Sign up to Codefresh

Welcome! Please select a provider to create your Codefresh account.



By signing up you agree to our [terms of service](#) and [privacy policy](#)

2. Accept the permissions request for the selected IdP:
 - For GitHub: To continue, click **Authorize codefresh-io**.
 - For Bitbucket: To continue, click **Grant access**.
 - For GitLab: To continue, click **Authorize**.

Once you confirm the permissions for your Git provider, Codefresh automatically connects to your Git provider and fetches your basic account details, such as your email.

3. Review the details for your new account, make the relevant changes, and click **NEXT**.



Let's finish setting up your account

FIRST NAME

Super

✓

LAST NAME

Fresh

✓

USER NAME

superfresh

✓

COUNTRY

Select country

▼

COMPANY NAME

Company Name

EMAIL ADDRESS

super.fresh@codefresh.io

✓

PHONE NUMBER (OPTIONAL)

Phone Number (optional)

NEXT

4. Enter a name for your account, and click **NEXT**.



Almost done...

ACCOUNT NAME

superfresh



NEXT

5. Finally, answer the questions to personalize your account and click **FINISH**.



Let's personalize your account

WHAT BEST DESCRIBES YOUR ROLE?

Select...

▼

HOW MANY SOFTWARE ENGINEERS WORK AT YOUR COMPANY?

Select...

▼

WHERE DID YOU HEAR ABOUT CODEFRESH?

Select...

▼

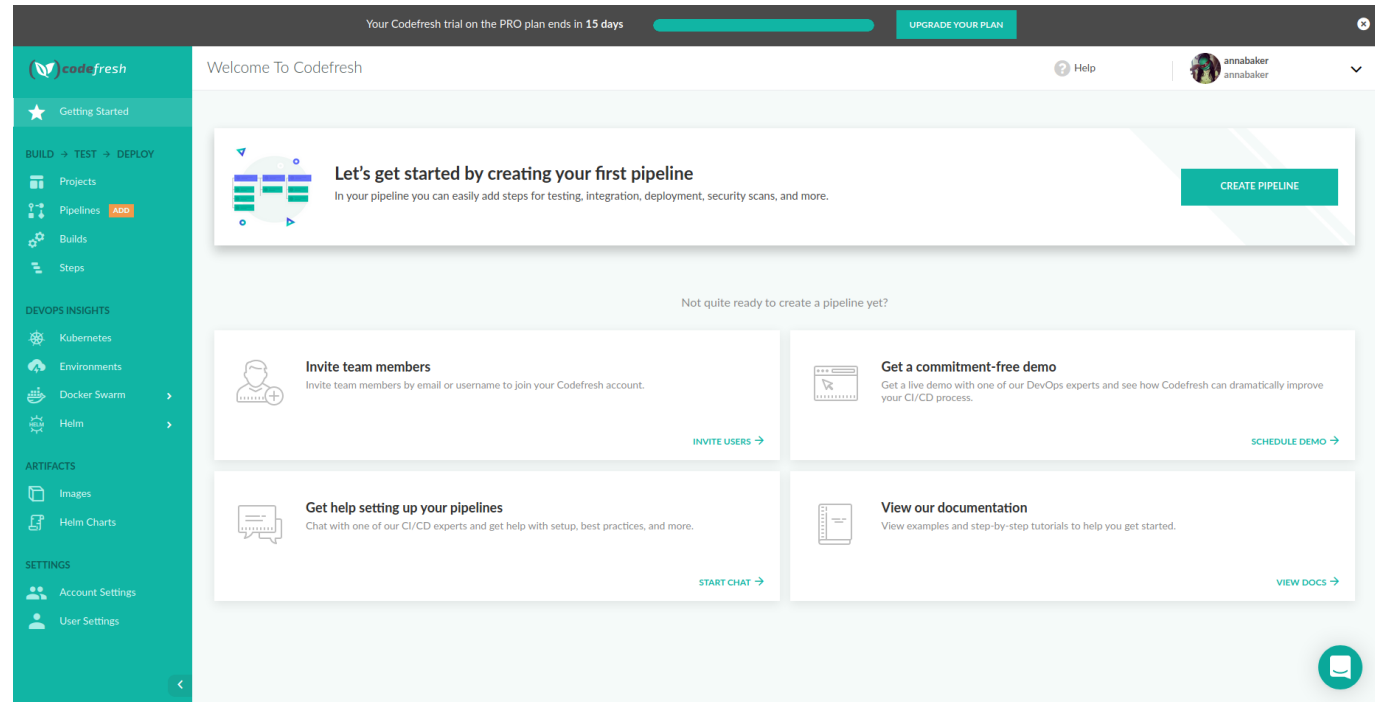
WHAT TECHNOLOGIES DO YOU PLAN ON USING CODEFRESH FOR?

Select all that apply

▼

FINISH

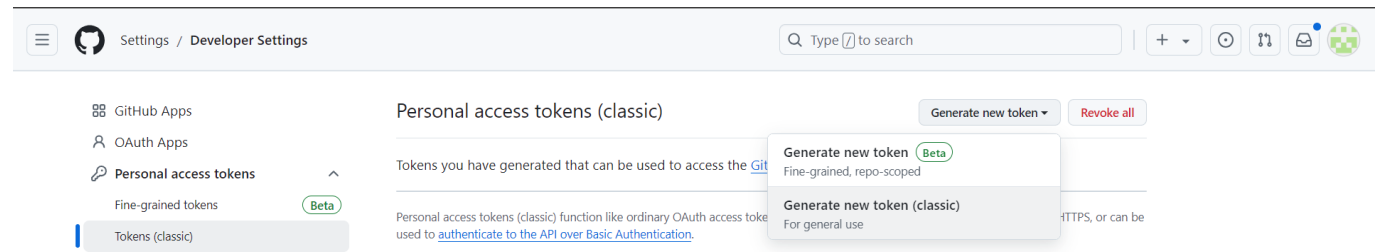
Congratulations! Your new Codefresh account is now ready.



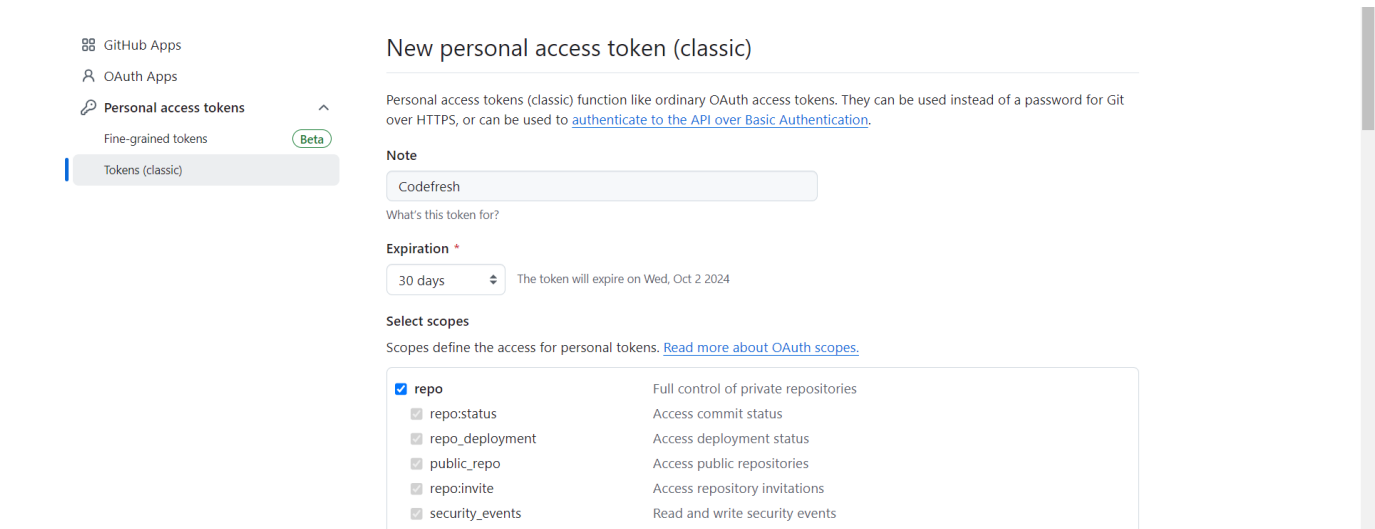
Setting Up Git Integration with GitHub

Step 1: Create a Personal Access Token on GitHub

- 1. Go to [GitHub](#) and log in to your account.
- 2. Navigate to **Settings** > **Developer settings** > **Personal access tokens**.
- 3. Click **Tokens(classic)**.



- 4. Select the required scopes, such as `repo` and `admin:repo_hook`.



☒ admin:repo_hook

Full control of repository hooks

☒ write:repo_hook

Write repository hooks

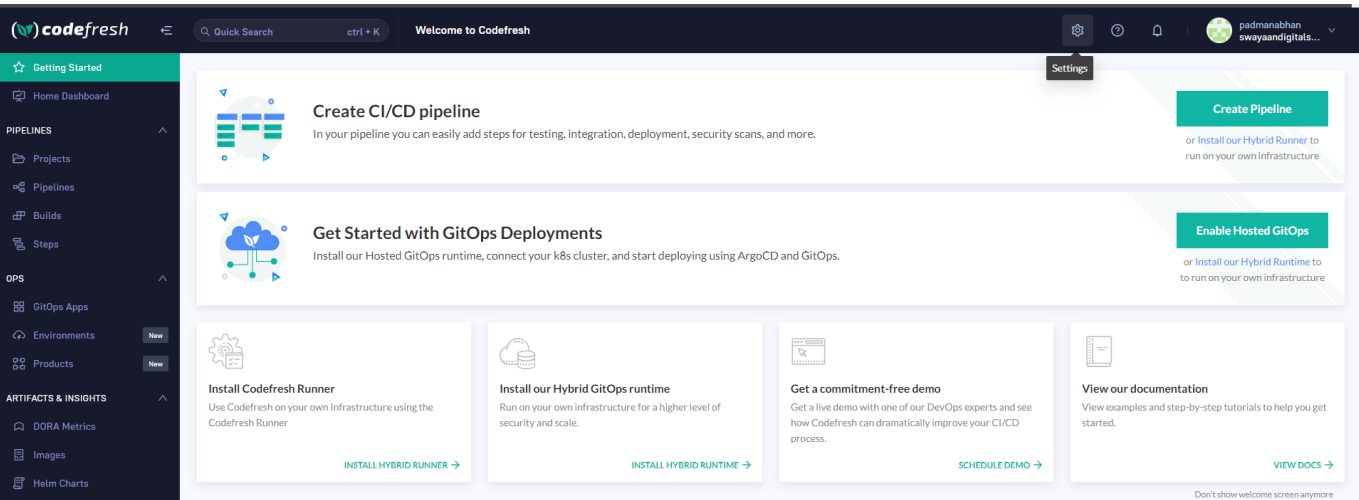
☒ read:repo_hook

Read repository hooks

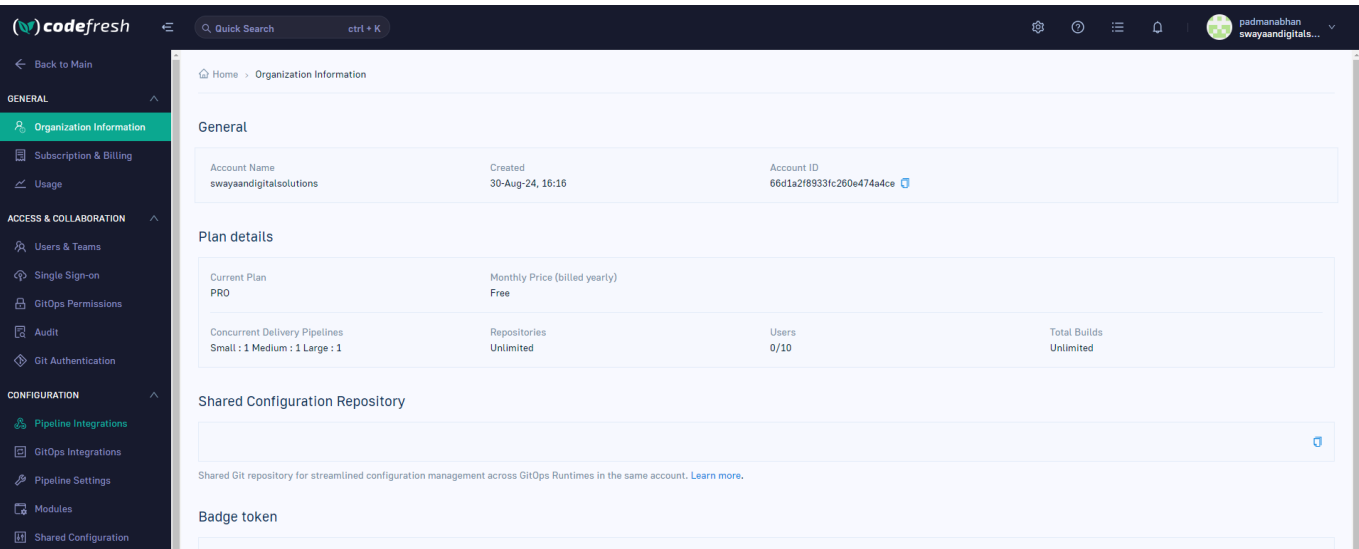
5. Click **Generate token** and save the token securely.

Step 2: Connect GitHub to Codefresh

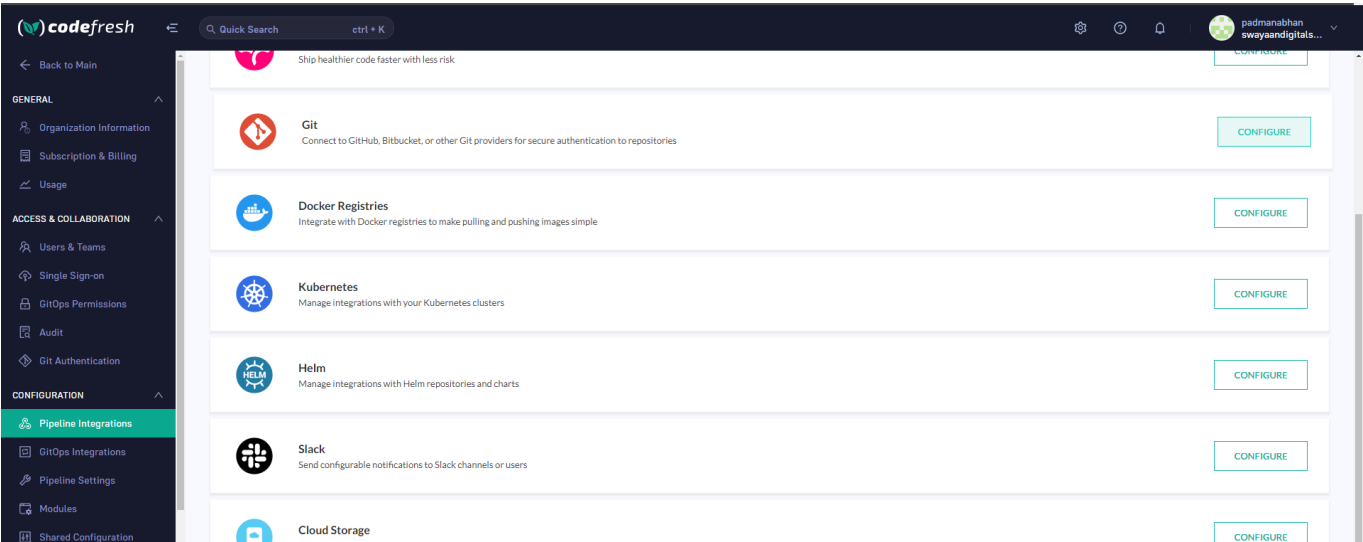
1. In the Codefresh UI, click on **Settings**.



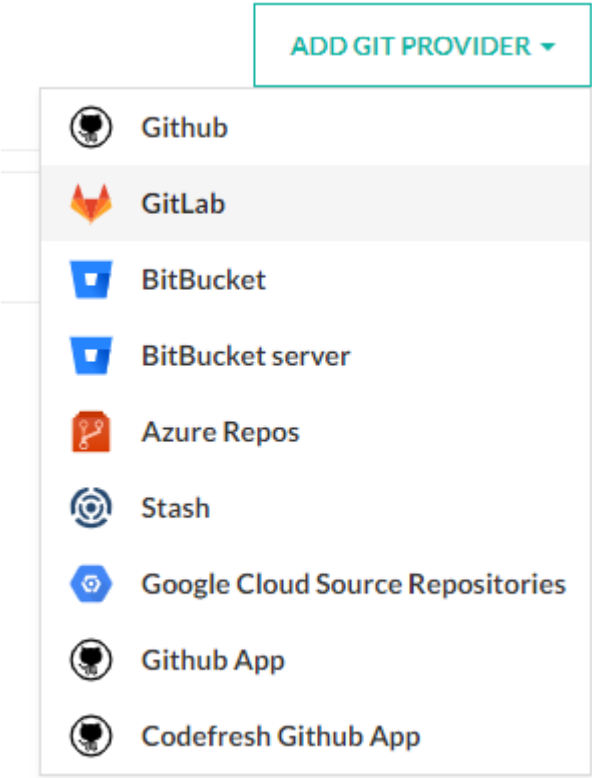
2. On sidebar bar under **configuration** select **Pipeline Integrations**.



3. Search for **Git Providers** and select **Configure**



4. Click on **Add Git Provider** and choose **GitHub**.



5. Enter your GitHub username and the Personal Access Token generated earlier.

[← INTEGRATIONS](#)

GIT

Manage integrations with any GIT provider

[ADD GIT PROVIDER ▾](#)

Github (github-1)
Enter the github connection details below

Allow access to all users ☒

NAME *

padmanabhansaravanan

Github is installed on-premise ☐ OFF

OAuth2

Access Token

[How to get the correct credentials](#)

ACCESS TOKEN *

.....

SAVE

CANCEL

TEST CONNECTION

6. Click **Test Connection** to ensure that the connection is successful.

7. Click **Save** to complete the integration.

Runtime Environment in Codefresh

Step 1: Access the Pipeline Runtimes Settings

1. **Click the Settings icon** on the toolbar.
2. From the sidebar, **select "Pipeline Runtimes."**

Step 2: Request a Cloud Build (SaaS Runtime Environment)

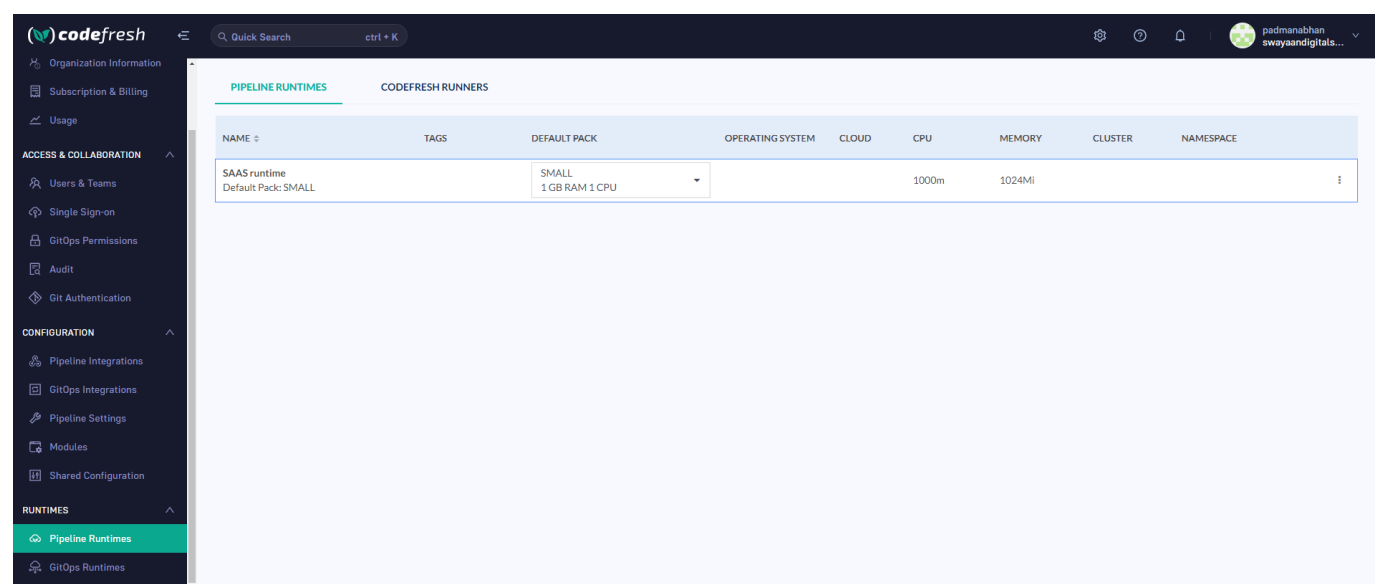
1. **Click the "Enable Cloud Builds" button** to request a Cloud Build (SaaS runtime environment).
2. This action will send an automated email request to the Codefresh support team.

Step 3: Await Approval

1. After sending the request, you should receive a response within 24 hours.
2. Once your request is approved, the SaaS runtime will appear in your list of Runtime Environments.

Step 4: Set the Default Runtime Environment

1. Once the Cloud Build runtime appears, you can select it as the default for all pipelines in your account.
2. To do this, click on the specific runtime environment you want to set as default, and mark it as the **default runtime environment**.
3. You can always override this setting for specific pipelines if needed.



Simple Pipeline

Creating a simple CI/CD pipeline in Codefresh that uses GitHub as the repository, and includes steps for cloning, Maven build, and test, involves the following steps. I'll guide you through each step of the process.

Step 1: Create a New Project

1. Access Projects:

- On the Codefresh dashboard, find the **Projects** tab on the left-hand side and click on it.
- Click on **New Project** to create a new project.

2. Name Your Project:

- Provide a name for your project, such as **ToDoAPI_Mongo_Project**.
- select icon of identity provider.
- Optionally, add a description to describe the project's purpose.

Create New Project

Projects are used to group pipelines and other settings together

PROJECT NAME

ToDoAPI_Mongo_Project

PROJECT TAGS

Add new tag

Tags can be used for filtering, managing permissions, and more

ICON



CANCEL

CREATE

Step 2: Create a New Pipeline

1. Navigate to Pipelines:

- Inside your project, go to the **Pipelines** tab.
- Click on **New Pipeline** to start creating a CI/CD pipeline.

2. Name Your Pipeline:

- Give your pipeline a name, such as `todoapi_mongo_pipeline`.


3. Choose a Git Repository:

- Under **Git Provider**, select **GitHub**.
- select user or git organization
- Select your repository from the list or add it manually if it's not listed.
- In this case, select `PadmanabhanSaravanan/todoapi_mongo`.

Create New Pipeline

Pipelines allow you to create build and deployment flows

PROJECT

 ToDoAPI_Mongo_Project ▾

Projects are used to group pipelines and other settings together

PIPELINE NAME

todoapi_mongo_pipeline

Select a name to describe this pipeline

ADD GIT REPOSITORY



This will add a Git clone step to your pipeline and create a commit trigger. You can add additional triggers to a pipeline, and enable/disable triggers after creation.

SELECT A GIT INTEGRATION

 github ▾

 [Configure or add new git integrations](#)

SELECT USER/GIT ORGANIZATION

 PadmanabhanSaravanan ▾

REPOSITORY

PadmanabhanSaravanan/todoapi_mongo ▾

TAGS

Add New Tag

CANCEL

CREATE

Step 3: Configure the Pipeline Steps

1. Add a **git-clone** Step:

- In the pipeline editor, click on **Add Step**.
- Choose **git-clone** from the list of available steps.
- Configure the step:
 - **Step Name:** `clone_repository`

- **Repository:** `PadmanabhanSaravanan/todoapi_mongo`
- **Branch:** `master` (or any branch you want to work with)
- **Stage:** `checkout`

2. Add a Maven Build Step:

- Click on **Add Step** again.
- Choose **freestyle** as the step type.
- Configure the step:
 - **Step Name:** `build_maven`
 - **Stage:** `build`
 - **Image:** `maven:3.8.1-jdk-11` (This is a Docker image that includes Maven and JDK 11)
 - **Working Directory:** `${{clone_repository}}`
 - **Commands:**

```
mvn clean install
```

- This command will compile your code and run all tests defined in your Maven project.

3. Add a Maven Test Step:

- Testing is often part of the Maven build process, but you can also separate it if needed.
- Add another **freestyle** step for testing:
 - **Step Name:** `test_maven`
 - **Stage:** `test`
 - **Image:** `maven:3.8.1-jdk-11`
 - **Working Directory:** `${{clone_repository}}`
 - **Commands:**

```
mvn test
```

Step 4: YAML File Example

If you prefer writing the pipeline configuration directly in YAML, here's a basic example that includes the steps we discussed:

```
version: '1.0'
stages:
  - checkout
  - build
  - test
```

```

steps:
  clone_repository:
    title: Cloning Repository
    type: git-clone
    repo: "PadmanabhanSaravanan/todoapi_mongo"
    revision: "master"
    stage: checkout

  build_maven:
    title: Maven Build
    type: freestyle
    stage: build
    image: maven:3.8.1-jdk-11
    working_directory: '${{clone_repository}}'
    commands:
      - mvn clean install

  test_maven:
    title: Maven Test
    type: freestyle
    stage: test
    image: maven:3.8.1-jdk-11
    working_directory: '${{clone_repository}}'
    commands:
      - mvn test

```

Step 5: Save and Run the Pipeline

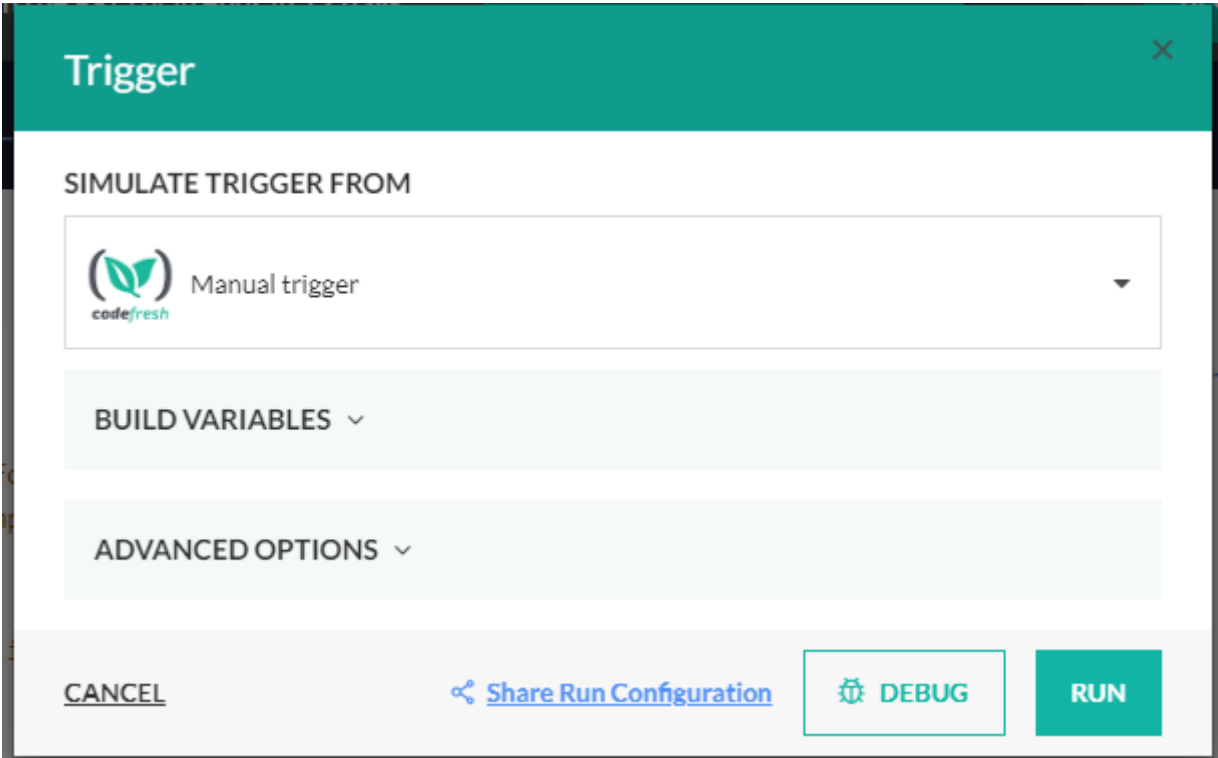
1. Save Your Pipeline:

- After configuring your steps, save the pipeline by clicking the **Save** button.



2. Run the Pipeline:

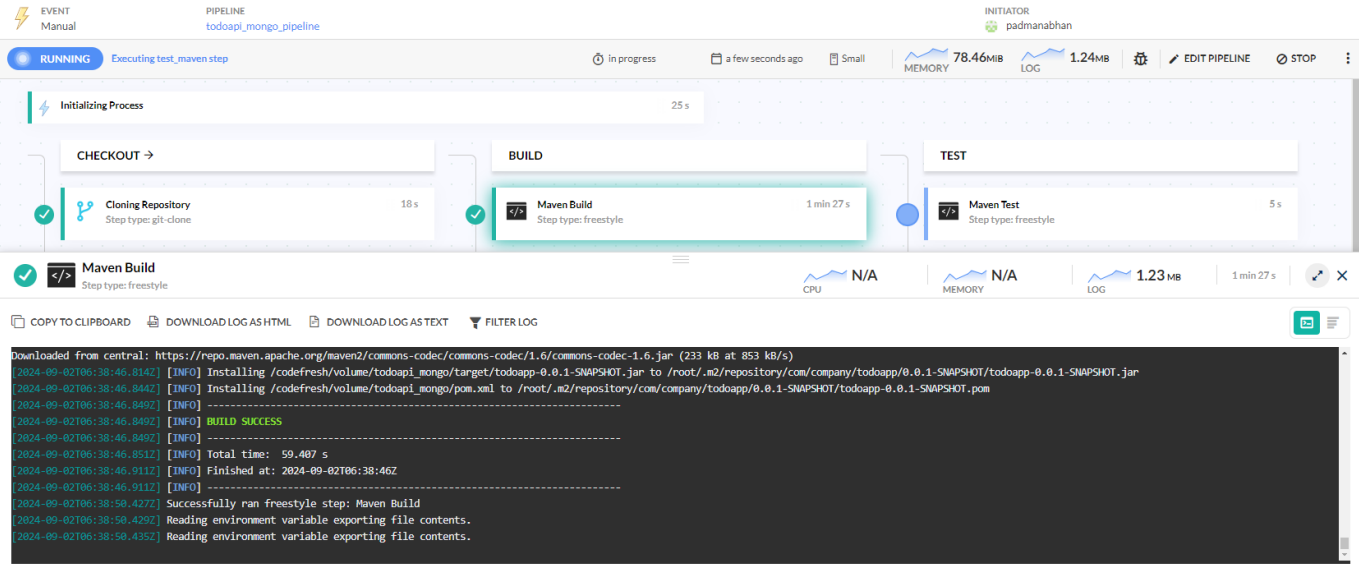
- Once saved, you can manually trigger the pipeline by clicking **Run**.
- Monitor the execution of each step in real-time through the pipeline execution view.



Step 6: Review Pipeline Execution

1. View Logs:

- For each step, you can view the logs to see the output of the commands that were run. This is useful for debugging and ensuring that everything works as expected.



YAML File Structure

1. Version

- `version: '1.0'`
- This defines the version of the pipeline schema you're using. It's important for backward compatibility and ensuring that the pipeline behaves as expected.

2. Stages

- **Stages** group together related steps. They define the order in which the pipeline steps will be executed and can be used to organize your pipeline into logical parts like `checkout`, `build`, `test`, and `deploy`.
- Example:

```
stages:  
  - checkout  
  - build  
  - package
```

- Here, there are three stages: `checkout`, `build`, and `package`. Each stage can have one or more steps.

3. Steps

- **Steps** are the individual tasks that the pipeline will execute. Each step is defined under a specific stage and can have its own configurations, like the type of step, the Docker image to use, commands to run, etc.
- Example of a step:

```
master_clone:  
  title: Cloning master repository...  
  type: git-clone  
  repo: "PadmanabhanSaravanan/todoapi_mongo"  
  revision: 'master'  
  stage: checkout
```

- In this example:
 - **title**: A descriptive name for the step.
 - **type**: Specifies what kind of step this is. For example, `git-clone` is used to clone a Git repository. You can specify a version for the step type like `git-clone:1.0.0`.
 - **repo**: The Git repository URL or identifier.
 - **revision**: The branch or tag to clone (e.g., `master`).
 - **stage**: Indicates the stage to which this step belongs.
- Another example of a step:

```
build_maven:  
  title: Building Maven Project  
  type: freestyle  
  stage: build  
  image: maven:3.8.1-jdk-11  
  working_directory: '${{master_clone}}'  
  commands:  
    - mvn clean install
```

- Here:
 - **type: freestyle**: A generic step where you can run any commands.
 - **image**: Specifies the Docker image to use for this step (e.g., a Maven image with Java 11).
 - **working_directory**: The directory in which to run the commands. It uses the output from the **master_clone** step.
 - **commands**: The shell commands to execute in this step.

4. Working Directory

- **working_directory**: This defines the directory where the step's commands will be executed. You can refer to the output of previous steps using `${{step_name}}`.

5. Image Configuration

- **image_name**: Specifies the name of the Docker image to build or use.
- **tag**: Defines the tag (e.g., **latest**, **master**, etc.) to apply to the Docker image.
- **dockerfile**: Indicates which Dockerfile to use if building a Docker image.

6. Deployment

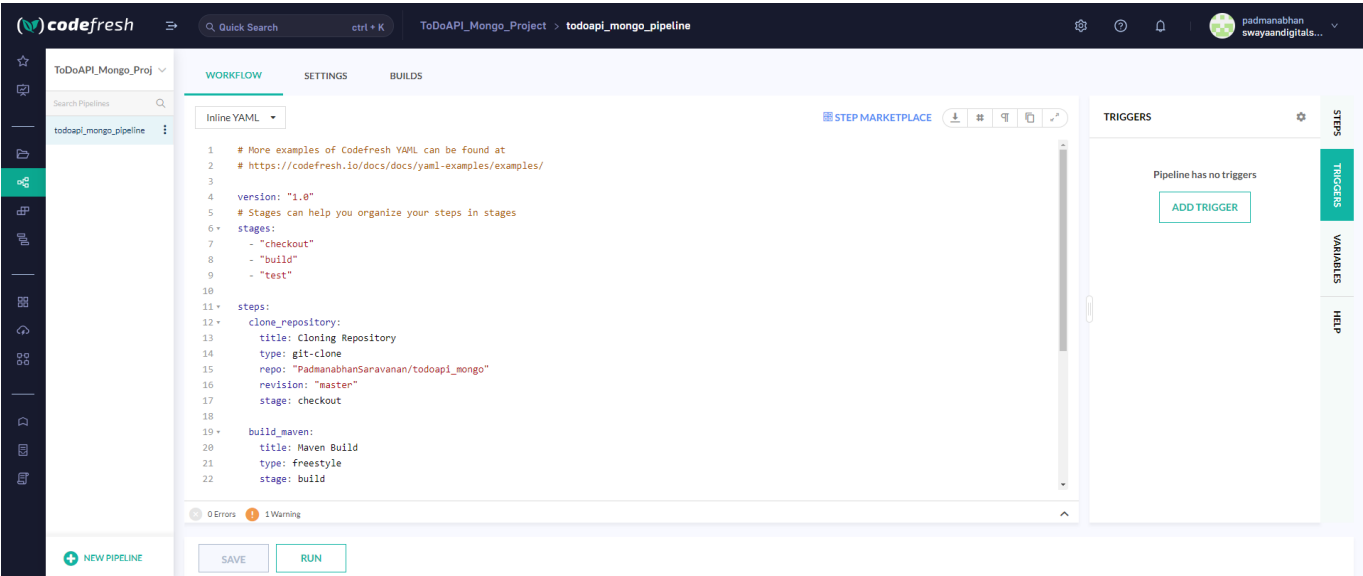
- After building and testing the application, a deployment step is often included. This step would push the Docker image to a container registry and deploy it to a Kubernetes cluster or other environments.
- Example of a deployment step:

```
deploy:
  title: Deploying to Kubernetes
  type: deploy
  stage: deploy
  image_name: padmanabhan1/todoapi-mongo
  working_directory: '${{master_clone}}'
  kubernetes:
    cluster: my-cluster
    namespace: default
    manifest: deployment.yaml
```

Triggering Pipelines

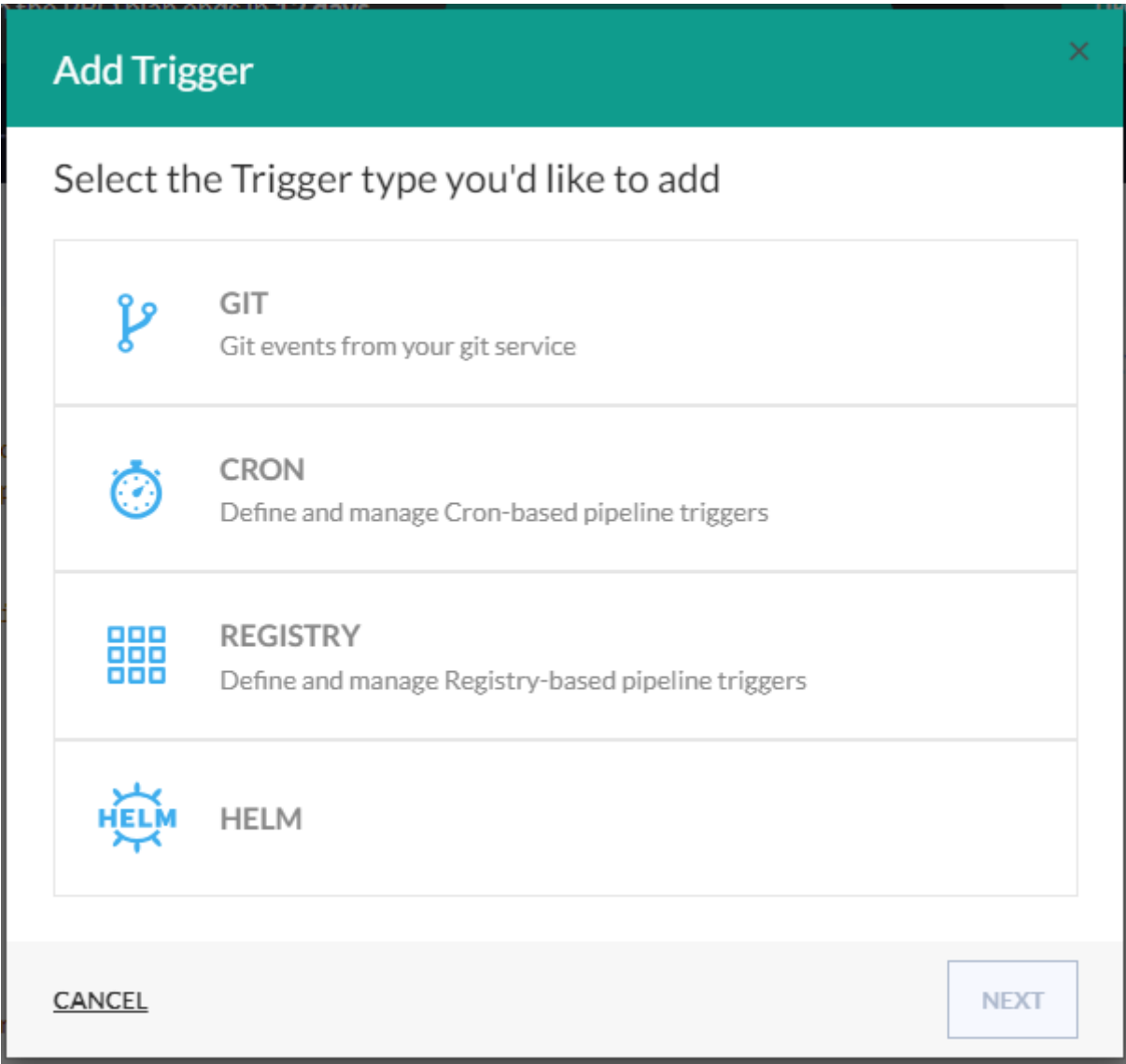
Step 1: Access the Pipeline

1. **Open the existing pipeline** where you want to add the trigger.
2. On the right-hand side of the pipeline page, find the **Triggers** section.



Step 2: Add a New Trigger

- 1. Click on **Add Triggers**.
- 2. In the **Add Trigger** window, select **Git** as the trigger type.
- 3. Click **Next**.



Step 3: Configure the Trigger

- 1. In the **Add Trigger - Git** window, name your trigger (e.g., **push**).
- 2. Select a **Git Integration** from the dropdown list.
- 3. Choose the **Git user or organization** and the **repository** where you want the trigger to be applied.
- 4. Under **Trigger By**, select **Push Commits** to trigger the pipeline whenever a commit is pushed to the selected repository.
- 5. Click **Next**.

Add Trigger - Git

Git Automated build


TRIGGER NAME *

push

DESCRIPTION


Trigger Pipleine on push event

SELECT A GIT INTEGRATION

 padmanabhansaravanan

Configure or add new git integrations

SELECT USER/GIT ORGANIZATION

 PadmanabhanSaravanan

REPOSITORY

PadmanabhanSaravanan/todoapi_mongo

TRIGGER BY *

☒ Push commits

☐ Any Pull request event

☐ Pull request closed

☐ Pull request closed (not merged)

☐ Pull request edited

☐ Pull request unassigned

☐ Pull request review request removed

☐ Pull request unlabeled

☐ Pull request comment added (restricted) ?

☐ Pull request review approved

☐ Pull request review changes requested

☐ Push tags

☐ Pull request opened

☐ Pull request merged

☐ Pull request reopened

☐ Pull request assigned

☐ Pull request review requested

☐ Pull request labeled

☐ Pull request synchronized

☐ Pull request comment added ?

☐ Pull request review commented

☐ Release published

☐ Pull request comment added (restricted) ?

☐ Pull request review approved

☐ Pull request review changes requested

☐ Release unpublished

☐ Release edited

☐ Release prereleased

☐ Pull request comment added ?

☐ Pull request review commented

☐ Release published

☐ Release created

☐ Release deleted

☐ Release released

Configure filter

BRANCH (REGEX EXPRESSION)

Regex Input ☒

./*/gi

SUPPORT PULL REQUEST EVENTS FROM FORKS? ?

☐ OFF

PR COMMENT (REGEX EXPRESSION)

./*/gi

PULL REQUEST TARGET BRANCH (REGEX EXPRESSION)

/yourPattern/gi (e.g /master/gi)

MODIFIED FILES (GLOB EXPRESSION) ?

Leave empty to execute on any file change

ADVANCED OPTIONS▶

? How to use Git triggers

BACK

NEXT

Step 4: Review and Save the Trigger

- 1. Review the configuration to ensure everything is set up correctly.
- 2. Click **Next** to finalize and add the trigger to your pipeline.

Your pipeline will now be automatically triggered whenever there is a push to the specified repository, keeping your CI/CD process automated and efficient.

Advanced Pipeline Configuration

- [Configuring Account Level Pipeline Settings](#)
- [Keeping Your Pipelines DRY with Shared Configuration](#)
- [Using Shared Secrets](#)

Configuring Account Level Pipeline Settings

Step 1: Access Account-Level Settings

1. Log In to Codefresh

- Go to [Codefresh](#) and log in to your account.

2. Navigate to Pipeline Settings

- Click on the **Settings** icon in the toolbar.
- From the sidebar, select **Configuration**.
- Click on **Pipeline Settings** to access global settings.

Step 2: Configure Project Settings

1. Auto-Create Projects for Teams

- **Enable/Disable Auto-Create Projects:**

- By default, this option is enabled, which automatically creates projects when teams are added to the account.
- To configure, toggle the **Auto-create projects for teams** setting.

Auto Create projects for each team

When creating a team, automatically create a project for the same team.

Once a project is created for the team, editing the team or deleting it does not affect the project.

By default, all teams can view pipelines from all projects.

A team can only create/edit/delete pipelines in the project assigned to their team.

You can change this in ["permissions"](#).

Enable Auto Create projects for each team



- **What It Does:**

- Creates a project with the same name and tag as the team.
- Sets up access-control rules for the team, granting Read access to the project and Pipeline rules with all privileges except Debug.

2. Pipeline Creation Options

- **Allow Pipeline Creation from Templates:**

- Enable or disable the option for users to [create new pipelines](#) from [pipeline templates](#).

- **Allow Pipeline Cloning:**

- Enable or disable the option for users to create pipelines by cloning existing ones.
- **Allowed Sources for Pipeline YAMLs:**
 - **Inline YAML:** Enable or disable the inline editor for YAML stored in Codefresh SaaS.
 - **YAML from Repository:** Enable or disable uploading YAMLs from connected Git repositories.
 - **YAML from External URLs:** Enable or disable loading YAMLs from external URLs.

Step 3: Configure Pipeline Scopes and Cluster Contexts

1. Pipeline Scopes

- **Define Scopes:**
 - Set account-level scopes for pipeline resources, including full access, read/write access, or CRUD permissions.

Pipeline Scopes

Set access levels for the pipeline by defining read, write, and delete permissions to Codefresh endpoints.

Configure account-level scopes

Configure

- **Override Custom Scopes:**
 - You can override the account-level scopes for specific pipelines by configuring custom scopes.

Step 4: Configure Build and Execution Settings

1. Pausing Build Executions

- **Enable/Disable Pausing:**
 - Allows pausing of builds for all pipelines, useful during maintenance.
- **Behavior:**
 - New pipelines will be paused immediately.
 - Existing pipelines will be paused after current builds complete.

Pause Build Execution

Pause all new builds from executing. Builds status will be set to pending until this option is disabled, after that builds will resume execution.

Pause Build Execution



2. Restarting from Failed Steps

- **Enable/Disable Restart from Failed Step:**

- Allow users to restart pipelines directly from failed steps instead of starting from the beginning.

Restart Pipeline

Permit restart from failed steps



3. Memory Usage Warnings

◦ Set Memory Thresholds:

- Configure memory usage thresholds for pipeline builds to display alerts when memory usage exceeds the selected threshold.

Restart Pipeline

Permit restart from failed steps



4. Default Behavior for Build Steps

◦ Configure Image Push Options:

- Define whether built images are automatically pushed to a registry or not.

Build step default behaviour

- ☐ By default require to explicitly define whether built images will be pushed or not
- ☒ By default automatically push built images to the default registry
- ☐ By default disable automatic push of built images

5. Default Behavior for Pending-Approval Steps

◦ Configure Manual Approval:

- Determine if manual confirmation is required after clicking Approve or Reject in pending-approval steps.

Pending Approval

Require confirmation dialog when Approving / Rejecting pending approval steps

None

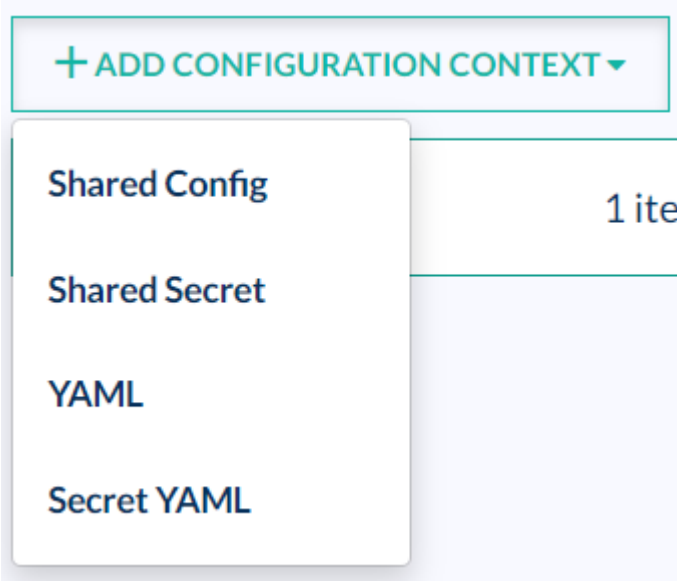


Keeping Your Pipelines DRY with Shared Configuration

Step 1: Access Shared Configuration Settings

1. In the Codefresh UI, click the **Settings** icon on the toolbar.

- 2. Select **Shared Configuration** from the sidebar.
- 3. Click **Add Shared Values** and choose the type of shared configuration context to add.



Step 2: Add Configuration Context

- 1. Enter a name for the shared configuration context.
- 2. Click **Save**.
- 3. Add one or more variables in **Key = Value** format. (Eg: **test = 12345**)
- 4. Optionally, toggle **Allow access to all users** ON to make it accessible to all users.
- 5. Click **Save**.



Types of Shared Configuration Contexts

- 1. **Shared Configuration**: For environment variables.
- 2. **Shared Secret**: For encrypted environment variables of sensitive data.
- 3. **YAML**: For Helm values or other generic YAML data.
- 4. **Secret YAML**: For encrypted YAML data.

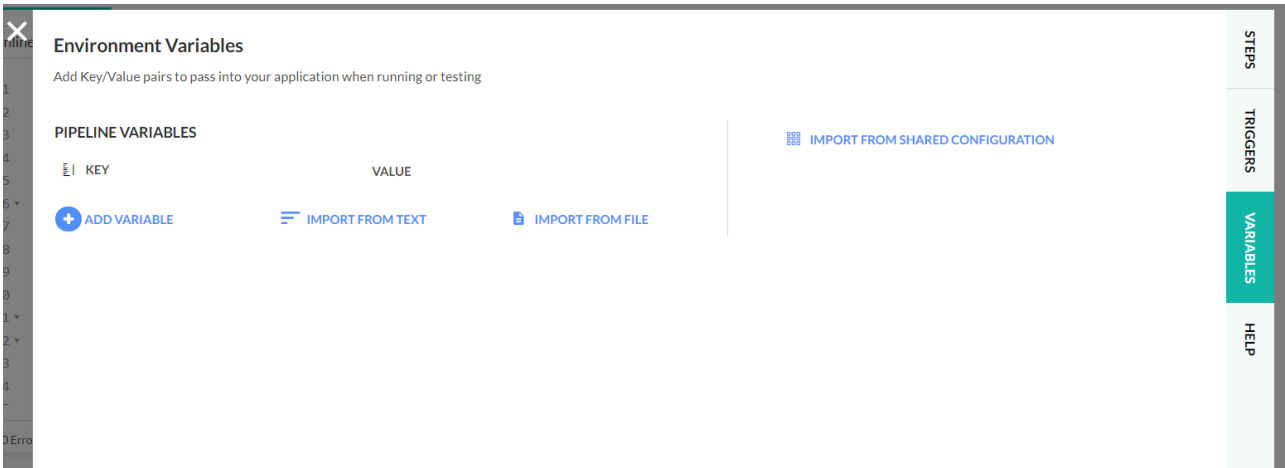
Note: RBAC (Role-Based Access Control) is supported for all types of shared configurations.

Using Shared Configuration in Pipelines

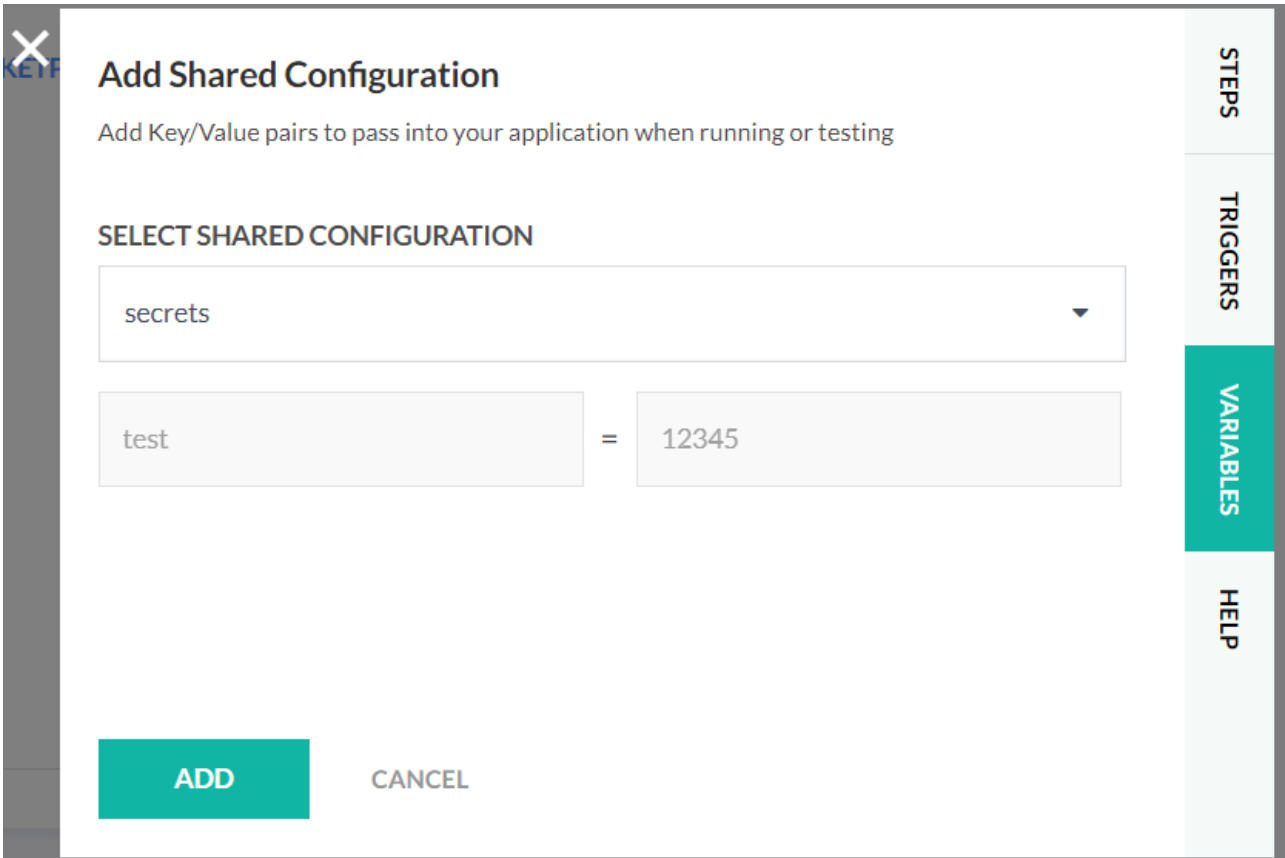
1. Importing Shared Configuration

- Open the pipeline editor.

- Go to the **Variables** tab on the right side.
- Click the gear icon and select **Open Advanced Options**.
- Click **Import from shared configuration** and select the snippet from the list.

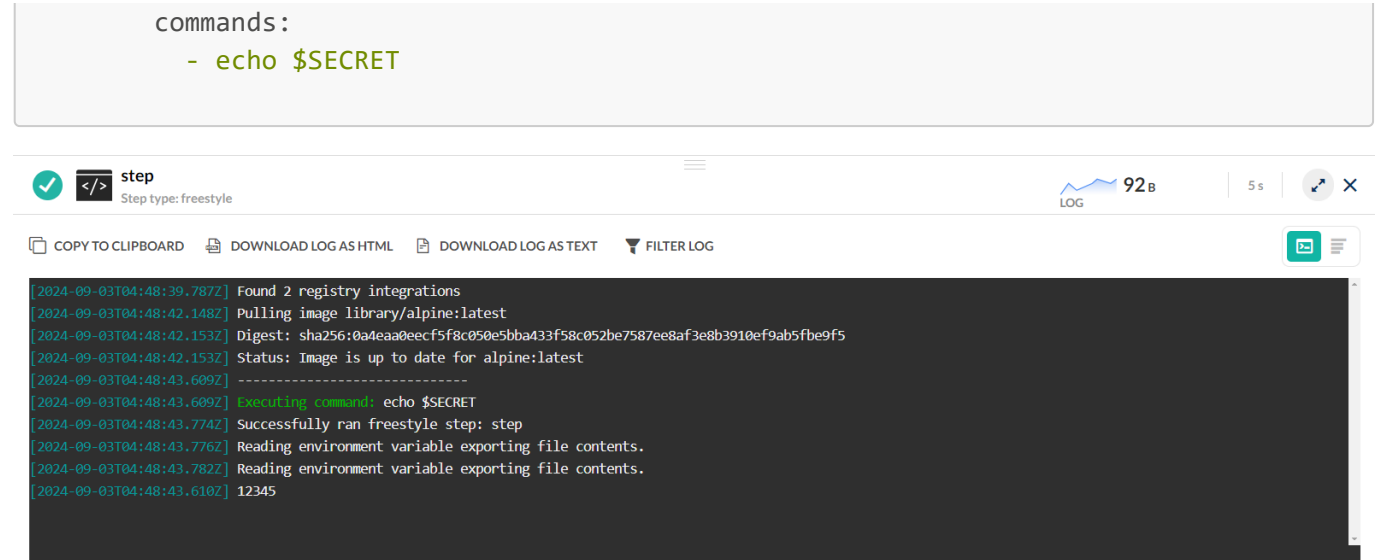


- Click **Add** to append the shared configuration values to your pipeline’s existing values.



2. **Example YAML**

```
version: '1.0'
steps:
  step:
    type: freestyle
    arguments:
      image: alpine
      environment:
        - SECRET=${{test}}
```



Using Shared Secrets

Shared secrets in Codefresh allow you to manage sensitive data such as API keys, passwords, and access tokens securely. These secrets can be shared across multiple pipelines, ensuring that sensitive information is handled consistently and securely.

Here's a step-by-step guide on how to use shared secrets in your Codefresh pipelines:

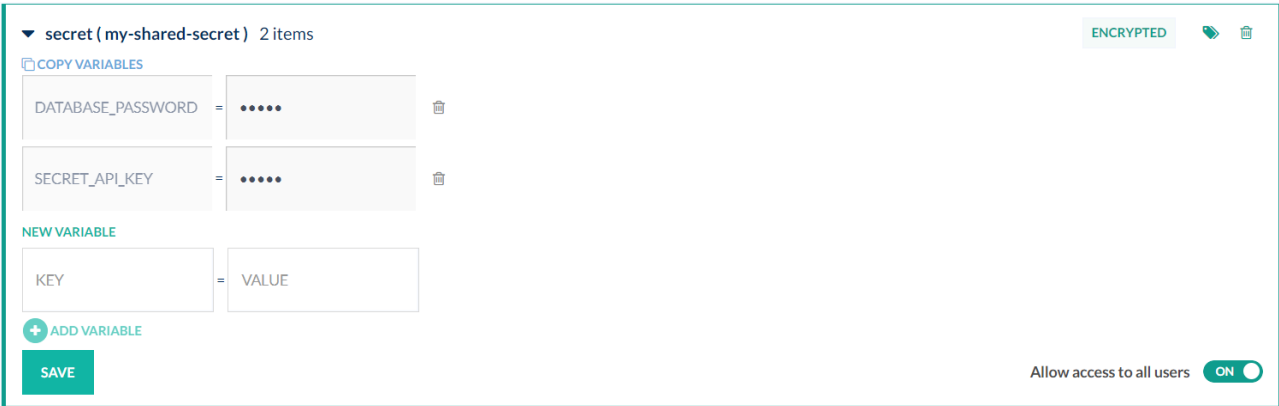
1. Create Shared Secrets

1. Access Shared Configuration Settings:

- In the Codefresh UI, click on the **Settings** icon on the toolbar.
- From the sidebar, select **Shared Configuration**.

2. Add a New Shared Secret:

- Click **Add Configuration Context** and select **Shared Secret**.
- Enter a name for the shared secret context (e.g., `my-shared-secret`).
- Add one or more secrets in the format `KEY=VALUE`.
- Toggle **Allow access to all users** if you want all users to access this secret (recommended to keep off for sensitive data).
- Click **Save**.



Example:

```
SECRET_API_KEY=your-api-key-here
DATABASE_PASSWORD=your-db-password
```

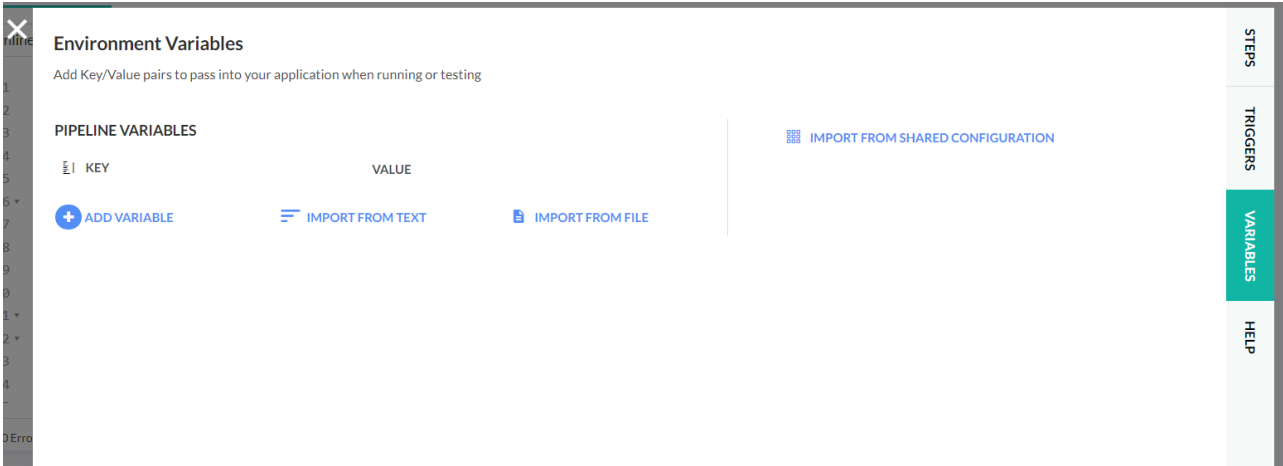
2. Use Shared Secrets in Your Pipeline YAML

1. Access Your Pipeline YAML File:

- Open the pipeline editor in Codefresh and navigate to the pipeline YAML file where you want to use the shared secrets.

2. Importing Shared Configuration

- Open the pipeline editor.
- Go to the **Variables** tab on the right side.
- Click the gear icon and select **Open Advanced Options**.
- Click **Import from shared configuration** and select the snippet from the list.



SECRET

Add Shared Configuration

Add Key/Value pairs to pass into your application when running or testing

Select shared configuration

my-shared-secret
ENCRYPTED

SECRET_API_KEY

=

.....

DATABASE_PASSWORD

=

.....

ADD

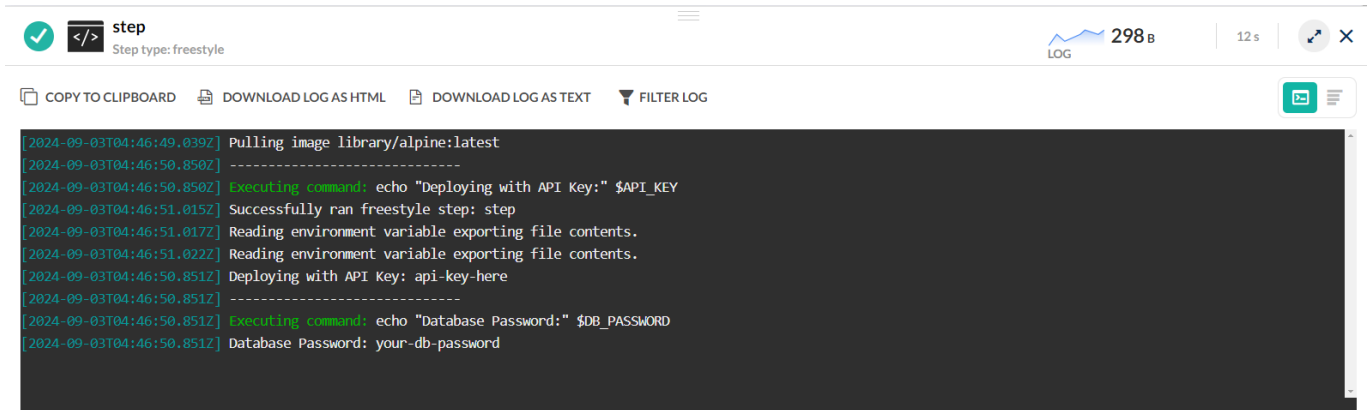
CANCEL

3. Reference Shared Secrets in Your Pipeline YAML:

Example:

```
version: '1.0'

steps:
  step:
    type: freestyle
    arguments:
      image: alpine
      environment:
        - API_KEY=${{SECRET_API_KEY}}
        - DB_PASSWORD=${{DATABASE_PASSWORD}}
      commands:
        - echo "Deploying with API Key:" $API_KEY
        - echo "Database Password:" $DB_PASSWORD
```



Conditional Execution of Steps

- [Branch Conditions](#)
- [Conditional expressions](#)

Branch Conditions

Usually, you'll want to define a branch condition, be it of the type **ignore** for blacklisting a set of branches or of the type **only** for allowlisting a set of branches. Each branch specification can either be an exact branch name, e.g. **master**, or a regular expression, e.g. **/hotfix\$/**. Case insensitive regexps (**/^FB-/i**) are also supported.

Here are some examples:

Only execute for the **master** branch

```
build_maven:
  title: Maven Build
  type: freestyle
  stage: build
  image: maven:3.8.1-jdk-11
  working_directory: '${{clone_repository}}'
  commands:
    - mvn clean install
  when:
    branch:
      only:
        - master
```

Only execute for feature branches

Only execute for branches whose name begins with **FB-** prefix, (feature branches):

```
build_maven:
  title: Maven Build
  type: freestyle
  stage: build
```

```
image: maven:3.8.1-jdk-11
working_directory: '${{clone_repository}}'
commands:
  - mvn clean install
when:
  branch:
    only:
      - /^FB-.*/i
```

Ignore specific branches

Ignore the development branch and main branch:

```
build_maven:
  title: Maven Build
  type: freestyle
  stage: build
  image: maven:3.8.1-jdk-11
  working_directory: '${{clone_repository}}'
  commands:
    - mvn clean install
  when:
    branch:
      ignore:
        - main
        - develop
```

Conditional expressions

Alternatively, you can use more advanced condition expressions.

This follows the standard condition expression syntax. In this case, you can choose to execute if **all** expression conditions evaluate to **true**, or to execute if **any** expression conditions evaluate to **true**.

NOTE

Use "" around variables with text to avoid errors in processing the conditions. Example:

```
"${{CF_COMMIT_MESSAGE}}"
```

Here are some examples.

AND conditional expression

Execute if the string `[skip ci]` is not part of the main repository commit message AND if the branch is `master`

```
build_maven:
  title: Maven Build
  type: freestyle
```



```

stage: build
image: maven:3.8.1-jdk-11
working_directory: '${{clone_repository}}'
commands:
  - mvn clean install
when:
  condition:
    all:
      noSkipCiInCommitMessage: 'includes(lower({% raw %}"${{CF_COMMIT_MESSAGE}}"
{% endraw %}), "skip ci") == false'
      masterBranch: '{% raw %}"${{CF_BRANCH}}"{% endraw %}' == "master"

```

OR conditional expression

Execute if the string `[skip ci]` is not part of the main repository commit message, OR if the branch is not a feature branch (i.e. name starts with FB-)

```

build_maven:
  title: Maven Build
  type: freestyle
  stage: build
  image: maven:3.8.1-jdk-11
  working_directory: '${{clone_repository}}'
  commands:
    - mvn clean install
  when:
    condition:
      any:
        noSkipCiInCommitMessage: 'includes(lower({% raw %}"${{CF_COMMIT_MESSAGE}}"
{% endraw %}), "skip ci") == false'
        notFeatureBranch: 'match({% raw %}"${{CF_BRANCH}}"{% endraw %}, "^FB-",
true) == false'

```

ALL conditions to be met

Each step in `codefresh.yml` file can contain conditions expressions that must be satisfied for the step to execute.

This is a small example of where a condition expression can be used:

```

build_maven:
  title: Maven Build
  type: freestyle
  stage: build
  image: maven:3.8.1-jdk-11
  working_directory: '${{clone_repository}}'
  commands:
    - mvn clean install

```

```
when:
  condition:
    all:
      executeForMasterBranch: "{% raw %}'${CF_BRANCH}' ==
'master'"
```

Conditional expression syntax

A conditional expression is a basic expression that is evaluated to true/false (to decide whether to execute or not to execute), and can have the following syntax:

Types

Type	True/False Examples	True/False
String	True: "hello"	<ul style="list-style-type: none">String with content = trueEmpty string = falseString with content = true
	False: ""	
String comparison is lexicographic.		
Number	True: 5	<ul style="list-style-type: none">Any number other than 0 = true.0 = false
	True: 3.4	
	True: 1.79E+308	
Boolean	True: true	<ul style="list-style-type: none">True = trueFalse = false
	False: false	
Null	False: null	Always false

Unary Operators

Operator	Operation
-	Negation of numbers
!	Logical NOT

Binary Operators

Operator	Operation
Add, String Concatenation	+
Subtract	-
Multiply	*
Divide	/

Operator	Operation
Modulus	%
Logical AND	&&
Logical OR	

Comparisons

Operator	Operation
==	Equal to
!=	Not equal to
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal

Here’s the updated [README.md](#) file that includes details on how to implement post-step operations in a Codefresh pipeline, based on the new reference provided.

Post Step Operations

This guide is based on the following pipeline structure:

```
version: '1.0'
stages:
  - checkout
  - build
  - test
  - notify

steps:
  clone_repository:
    title: Cloning Repository
    type: git-clone
    repo: "PadmanabhanSaravanan/todoapi_mongo"
    revision: "master"
    stage: checkout

  build_maven:
    title: Maven Build
    type: freestyle
    stage: build
    image: maven:3.8.1-jdk-11
    working_directory: '${{clone_repository}}'
```

```
commands:
  - mvn clean install
when:
  steps:
    - name: clone_repository
      on:
        - success

test_maven:
  title: Maven Test
  type: freestyle
  stage: test
  image: maven:3.8.1-jdk-11
  working_directory: '${{clone_repository}}'
  commands:
    - mvn test
  when:
    steps:
      - name: build_maven
        on:
          - success

notify_failure:
  title: Notify Build Failure
  type: freestyle
  stage: notify
  image: alpine:3.12
  commands:
    - echo "Build failed. Notifying user..."
    - echo "Sending failure notification..."
  when:
    steps:
      - name: build_maven
        on:
          - failure
```

Here's a step-by-step guide in the form of a README.md file that explains the pipeline configuration you provided:

Overview

The pipeline is divided into the following stages:

1. **Checkout:** Cloning the repository.
2. **Build:** Building the Maven project.
3. **Test:** Running unit tests.
4. **Notify:** Sending a notification if the build fails.

Version

The pipeline uses version **1.0** of the pipeline YAML syntax.

```
version: '1.0'
```

Stages

The pipeline is divided into four stages:

- **checkout**
- **build**
- **test**
- **notify**

```
stages:  
  - checkout  
  - build  
  - test  
  - notify
```

Steps

Each stage contains specific steps that define the actions to be taken. Below is a breakdown of each step.

1. Cloning the Repository

In this step, the pipeline clones the `todoapi_mongo` repository from GitHub. This step is crucial as it provides the source code for subsequent steps.

```
steps:  
  clone_repository:  
    title: Cloning Repository  
    type: git-clone  
    repo: "PadmanabhanSaravanan/todoapi_mongo"  
    revision: "master"  
    stage: checkout
```

- **title:** Describes the step.
- **type:** Specifies the type of step, in this case, `git-clone`.
- **repo:** The repository to clone.
- **revision:** The branch or commit to check out.
- **stage:** The pipeline stage where this step belongs.

2. Building the Maven Project

This step builds the Maven project using the `maven:3.8.1-jdk-11` Docker image. The step only runs if the `clone_repository` step succeeds.

```
build_maven:
  title: Maven Build
  type: freestyle
  stage: build
  image: maven:3.8.1-jdk-11
  working_directory: '${{clone_repository}}'
  commands:
    - mvn clean install
  when:
    steps:
      - name: clone_repository
        on:
          - success
```

- **image:** Specifies the Docker image to use.
- **working_directory:** The directory to execute the commands in.
- **commands:** The Maven commands to build the project.
- **when:** Defines the condition to run this step. It runs only if `clone_repository` succeeds.

3. Running Maven Tests

This step runs the unit tests in the Maven project. It is executed only if the `build_maven` step succeeds.

```
test_maven:
  title: Maven Test
  type: freestyle
  stage: test
  image: maven:3.8.1-jdk-11
  working_directory: '${{clone_repository}}'
  commands:
    - mvn test
  when:
    steps:
      - name: build_maven
        on:
          - success
```

- **commands:** The Maven command to run tests.
- **when:** This step is triggered only if `build_maven` succeeds.

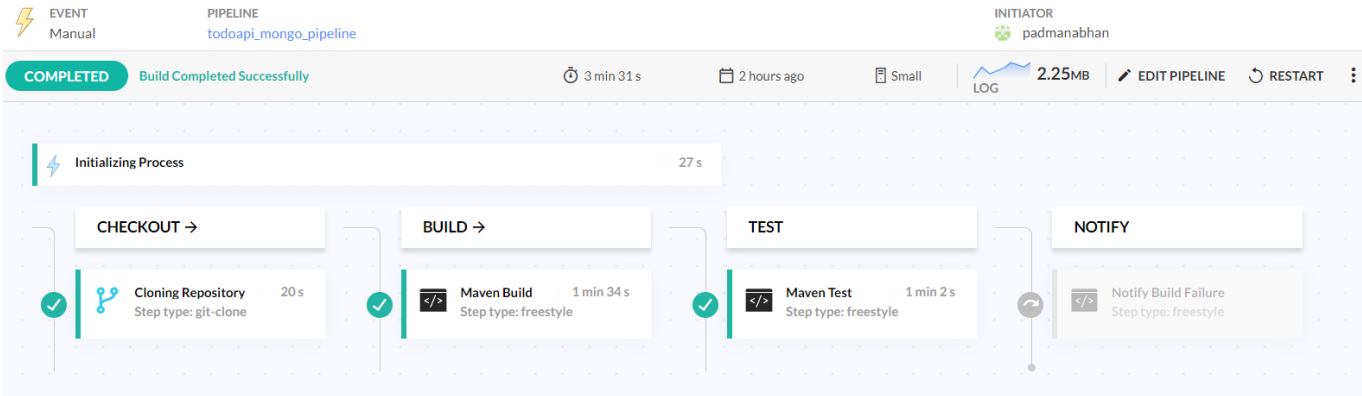
4. Notifying on Failure

If the `build_maven` step fails, this step sends a failure notification.

```
notify_failure:
  title: Notify Build Failure
  type: freestyle
```

```
stage: notify
image: alpine:3.12
commands:
  - echo "Build failed. Notifying user..."
  - echo "Sending failure notification..."
when:
  steps:
    - name: build_maven
      on:
        - failure
```

- **commands:** Commands to notify users about the build failure.
- **when:** This step is triggered only if `build_maven` fails.



Setting Up Docker

Here's a step-by-step guide for setting up Docker integration in Codefresh, similar to the GitHub integration guide you provided.

Setting Up Docker Integration with Codefresh

Step 1: Create a Docker Hub Account (If You Haven't Already)

1. Go to [Docker Hub](#) and log in to your account. If you don't have an account, click **Sign Up** and create one.

Step 2: Connect Docker Hub to Codefresh

1. In the Codefresh UI, click on **Settings**.
2. On the sidebar under **Configuration**, select **Pipeline Integrations**.
3. Scroll down to find **Docker Registries** and click on **Add Docker Registry**.
4. Choose **Docker Hub** from the list of available Docker registries.
5. Enter your Docker Hub username and the password.
6. Click **Test Connection** to verify the connection is successful.

7. Once the connection is confirmed, click **Save** to complete the integration.

Docker Integration

```
version: '1.0'
stages:
  - checkout
  - build
  - package

steps:
  master_clone:
    title: Cloning master repository...
    type: git-clone
    repo: "PadmanabhanSaravanan/todoapi_mongo"
    revision: 'master'
    stage: checkout

  build_maven:
    title: Building Maven Project
    type: freestyle
    stage: build
    image: maven:3.8.1-jdk-11 # Use a Maven Docker image with Java 11
    working_directory: '${{master_clone}}'
    commands:
      - mvn clean install # This builds the project and generates the JAR file

  build_my_app:
    title: Building Docker Image
    type: build
    stage: package
    image_name: padmanabhan1/todoapi-mongo
    working_directory: '${{master_clone}}'
    tag: 'master'
    dockerfile: Dockerfile
```

Here's a step-by-step guide explaining the CI/CD pipeline configuration you provided. This guide is designed to help you understand each part of the pipeline and how it contributes to building and packaging your project.

The pipeline consists of the following stages:

1. **Checkout:** Clone the master branch of the repository.
2. **Build:** Build the Maven project.
3. **Package:** Package the project into a Docker image.

Pipeline Configuration

Version

The pipeline uses version **1.0** of the pipeline YAML syntax.

```
version: '1.0'
```

Stages

The pipeline is organized into three stages:

- **checkout**: For cloning the repository.
- **build**: For building the Maven project.
- **package**: For packaging the project into a Docker image.

```
stages:  
  - checkout  
  - build  
  - package
```

Steps

Each stage contains specific steps that define the actions to be taken. Below is a breakdown of each step.

1. Cloning the Repository

The first step in the pipeline is to clone the **todoapi_mongo** repository from GitHub. This step ensures that you have the latest code from the master branch to work with.

```
master_clone:  
  title: Cloning master repository...  
  type: git-clone  
  repo: "PadmanabhanSaravanan/todoapi_mongo"  
  revision: 'master'  
  stage: checkout
```

- **title**: A descriptive name for the step.
- **type**: Specifies the type of step, in this case, **git-clone**.
- **repo**: The URL or name of the repository to clone.
- **revision**: The branch or commit to check out (in this case, **master**).
- **stage**: The pipeline stage where this step belongs.

Explanation:

- This step clones the repository into the pipeline's workspace. The **revision: 'master'** ensures that the latest code from the **master** branch is used for the build.

2. Building the Maven Project

The second step is to build the Maven project. This step compiles the code, runs the tests, and generates a JAR file.

```
build_maven:
  title: Building Maven Project
  type: freestyle
  stage: build
  image: maven:3.8.1-jdk-11 # Use a Maven Docker image with Java 11
  working_directory: '${{master_clone}}'
  commands:
    - mvn clean install # This builds the project and generates the JAR file
```

- **type:** Specifies the step as `freestyle`, meaning it allows for custom commands.
- **image:** The Docker image used to execute the commands. Here, `maven:3.8.1-jdk-11` is used, which includes Maven and Java 11.
- **working_directory:** The directory where the commands will be executed. `${{master_clone}}` references the directory created in the previous step.
- **commands:** The Maven command `mvn clean install` is used to clean the target directory and build the project.

Explanation:

- The `mvn clean install` command compiles the Java code, runs unit tests, and packages the compiled code into a JAR file. The JAR file is typically located in the `target/` directory after a successful build.

3. Building the Docker Image

The final step is to package the application into a Docker image. This step uses the Dockerfile in your repository to build the image.

```
build_my_app:
  title: Building Docker Image
  type: build
  stage: package
  image_name: padmanabhan1/todoapi-mongo
  working_directory: '${{master_clone}}'
  tag: 'master'
  dockerfile: Dockerfile
```

- **type:** Specifies the step as `build`, which is used for building Docker images.
- **image_name:** The name of the Docker image to be created.
- **working_directory:** The directory where the Dockerfile is located, referencing the directory created in the clone step.
- **tag:** A tag to assign to the Docker image. Here, it's tagged as `master`.
- **dockerfile:** The path to the Dockerfile used to build the Docker image.

Explanation:

- This step builds a Docker image using the Dockerfile located in the repository. The image is tagged with `master`, which can be used to identify this specific build. The resulting Docker image is named `padmanabhan1/todoapi-mongo`.