Continuous Integration and Deployment (CI/CD)

Once you connect your data project and Workspace through Git you need to implement a Continuous Integration (CI) and Deployment (CD) workflow.

This page covers how CI and CD work using a walkthrough example, and how Releases work.

How Continuous Integration works

As you expand and iterate on your data projects, you'll want to continuously validate your API Endpoints. In the same way that you write integration and acceptance tests for source code in a software project, you can write automated tests for your Endpoints to be run on each Pull/Merge Request.

Continuous Integration helps with:

- Linting: Syntax and formatting on Datafiles.
- Correctness: Making sure you can push your changes to a Tinybird Workspace.
- Quality: Running fixture tests and/or data quality tests to validate the changes in the Pull Request.
- Regression: Running automatic regression tests to validate Endpoint performance (both in processed bytes and time spent) and data quality.

The following section uses the CI template, GitHub Actions, and the Tinybird CLI to demonstrate how to test your Endpoints on any new commit to a Pull Request.

The GitHub Action

This example demonstrates automating CI/CD processes using GitHub as the provider with a GitHub Action, but you can use any suitable platform. See Tinybird's ready-to-use CI and CD templates in this repository. Templates can be adapted to suit your Workspace needs.

0. GitHub Action configuration

CI WORKFLOW

```
name: Tinybird - CI Workflow

 on:
    workflow_dispatch:
    pull_request:
      branches:
       - main
       - master
      types: [opened, reopened, labeled, unlabeled, synchronize]

  concurrency: ${{ github.workflow }}-${{ github.event.pull_request.number }}

  jobs:
      ci: # ci using Branches from Workspace 'web_analytics_starter_kit'
```

Now, add a new `TB_ADMIN_TOKEN` secret with the Workspace admin token to the GitHub repository's Settings.

Let's review the generated CI workflow:

1. Trigger the CI workflow

RUN THE CI WORKFLOW ON EACH COMMIT TO A PULL REQUEST, WHEN LABELLING, OR WITH OTHER KINDS OF TRIGGERS

```
on:
   workflow_dispatch:
   pull_request:
     branches:
      - main
     types: [opened, reopened, labeled, unlabeled, synchronize]
```

Key points: The CI workflow is triggered when a new Pull Request is opened, reopened, synchronized or labels are updated, and the base branch has to be `main`.

2. Configure the CI job

```
ci: # ci using Branches from Workspace 'web_analytics_starter_kit'
   uses: tinybirdco/ci/.github/workflows/ci.yml@main
   with:
      data_project_dir: .
   secrets:
      tb_admin_token: ${{ secrets.TB_ADMIN_TOKEN }}  # set Workspace admin toker
      tb_host: https://api.tinybird.co
```

The `ci` job has a comment with a reference to the remote Workspace, which in this case is `web_analytics_starter_kit`.

If your data project directory is not in the root of the Git repository, change the `data_project_dir` variable.

About secrets:

- `tb_host`: The url of the region you want to use. By default, this is populated with the region of the Workspace you had on `tb init --git`
- `tb_admin_token`: The Workspace admin token. This grants all the permissions for a specific Workspace. You can find more information in the Auth Tokens docs or in the `Auth Tokens` section of the Tinybird UI.

The CI Workflow

The CI workflow is based on this generic Tinybird CI template that performs the following steps:

0. Workflow configuration

```
defaults:
   run:
     working-directory: ${{ inputs.data_project_dir }}
 if: ${{ github.event.action != 'closed' }}
 steps:
   - uses: actions/checkout@master
     with:
       fetch-depth: 300
       ref: ${{ github.event.pull_request.head.sha }}
   - uses: actions/setup-python@v5
     with:
       python-version: "3.11"
       architecture: "x64"
       cache: 'pip'
```

Key points: This sets the default `working-directory` to the `data_project_dir` variable, check outs the `main` branch to get the head commit, checks the `TB_ADMIN_TOKEN`, and installs Python 3.11.

1. Install the Tinybird CLI

```
- name: Install Tinybird CLI
  run: pip install tinybird-cli

- name: Tinybird version
  run: tb --version
```

The Tinybird CLI is required to interact with your Workspace, create a test Branch, and run the tests.

> Note that you could run this workflow locally by having a local data project and the CLI authenticated to your Tinybird Workspace.

2. Check the data project syntax and the authentication

```
- name: Check all the datafiles syntax
  run: tb check

- name: Check auth
  run: tb --host ${{ secrets.tb_host }} --token ${{ secrets.tb_admin_token }} au
```

Check the Datafiles syntax and the Tinybird authentication.

3. Create a new Tinybird Branch to deploy changes and run the tests

A Branch is an isolated copy of the resources in your Workspace at a specific point in time. It's designed to be temporary and disposable so that you can develop and test changes before deploying them to your Workspace.

A Branch is created on each CI job run. In this example, `GITHUB_RUN_ID` is used as a unique identifier for the Tinybird Branch name, so multiple tests can be run in parallel. If a Branch with this name was created before, it is removed and recreated again.

Branches are created using the `tb branch create` command. Once the Pull Request with your changes has been merged, the Tinybird Branch is deleted.

```
- name: Try to delete previous Branch
  run: |
    output=$(tb --host ${{ secrets.tb_host }} --token ${{ secrets.tb_admin_t
    BRANCH_NAME="tmp_ci_${_NORMALIZED_BRANCH_NAME}_${{ github.event.pull_req
```

```
      # Check if the branch name exists in the output
      if echo "$output" | grep -q "\b$BRANCH_NAME\b"; then
        tb \
        --host ${{ secrets.tb_host }} \
        --token ${{ secrets.tb_admin_token }} \
        branch rm $BRANCH_NAME \
        --yes
      else
        echo "Skipping clean up: The Branch '$BRANCH_NAME' does not exist."
```

Use the `_ENV_FLAGS` variable to configure which data to attach to the Branch. Use the `--last-partition --wait` flag to attach the most recently ingested data in the Workspace. This way, you can run the tests using the same data as in production. Alternatively, leave it empty and use fixtures.

4. Push the changes to the Tinybird Branch

You can deploy the changes in your current Pull Request to the test Branch previously created in two ways:

By default with `tb deploy`

The `tb deploy` command gets the `diff` resources between your current Pull Request branch and the base branch head commit, and pushes the resources to the Tinybird Workspace. This is the default behavior for all Pull Requests - you don't have to take any manual action.

Custom deploy command

Alternatively, for more complex changes, you can decide how to deploy the changes to the test Branch.

For this to work, you have to place an executable shell script file in `deploy/$VERSION/deploy.sh` with the CLI commands to push the changes. Learn more about this in the custom deployment strategies docs.

```
  - name: Deploy changes to the test Branch
    run: |
      source .tinyenv
      DEPLOY_FILE=./deploy/${VERSION}/deploy.sh
      if [ ! -f "$DEPLOY_FILE" ]; then
        echo "$DEPLOY_FILE not found, running default tb deploy command"
        tb --semver ${VERSION} deploy ${CI_FLAGS}
```

```
          tb release ls
        fi

    - name: Custom deployment to the test Branch
      run: |
        source .tinyenv
        DEPLOY_FILE=./deploy/${VERSION}/deploy.sh
```

Note the `$VERSION` variable is defined in the `.tinyenv` file. By default, it is `0.0.0`, but you can increment as needed.

5. Run the tests

You can now run your test suite!

There are three type of tests:

- Data fixture tests: These test very concrete business logic based on fixture data (see `datasources/fixtures`).
- Data quality tests: These test very concrete data scenarios.
- Regression tests: These test that requests to your API Endpoints are still working as expected. For these tests to work, you must attach production data(`last-partition`) when creating the test Branch.

To learn more about testing Tinybird data projects, refer to the Implementing test strategies docs.

Additionally, you can run a shell script with CLI commands after the default deployment with the `postdeploy.sh` file.

```
    - name: Post deploy
      run: |
        POSTDEPLOY_FILE=./deploy/${VERSION}/postdeploy.sh
        if [ -f "$POSTDEPLOY_FILE" ]; then
          if ! [ -x "$POSTDEPLOY_FILE" ]; then
            echo "Error: You do not have permission to execute '$POSTDEPLOY_FI
            echo "> chmod +x $POSTDEPLOY_FILE"
            echo "and commit your changes"
            exit 1
          else
            $POSTDEPLOY_FILE
          fi
        fi

  - name: Get regression labels
```

6. Delete the Branch

By default, Branches are not deleted until the Branch has been merged into the main Workspace. The following step runs after the tests:

```
  - name: Try to delete previous Branch
    run: |
       output=$(tb --host ${{ secrets.tb_host }} --token ${{ secrets.tb_admin_
       BRANCH_NAME="tmp_ci_${_NORMALIZED_BRANCH_NAME}_${{ github.event.pull_re

       # Check if the branch name exists in the output
       if echo "$output" | grep -q "\b$BRANCH_NAME\b"; then
          tb \
             --host ${{ secrets.tb_host }} \
             --token ${{ secrets.tb_admin_token }} \
             branch rm $BRANCH_NAME \
             --yes
       else
          echo "Skipping clean up: The Branch '$BRANCH_NAME' does not exist."
       fi
```

You can have up to simultaneous 3 Branches per Workspace at any time. Contact us at support@tinybird.co if you need to increase this limit.

How Continuous Deployment works

Once a Pull Request passes CI and has been reviewed and approved by a peer, it's time to merge it to your main Git branch.

Ideally, changes should be automatically deployed to a new release; this is called `Continuous Deployment` (or Continuous Delivery).

While efficient, this workflow comes with several challenges, most of them related to handling the current state of your Tinybird Workspace. For instance:

- As opposed to when you deploy a stateless application, deployments to a Workspace are incremental, based on the previous resources in the Workspace.
- Similar issues arising from state handling are certain resources or operations that are created or run programmatically: populating operations or permission handling.
- Deployments are also performed on the same live Release; you need to be aware of this and implement policy to avoid collisions from different Pull Requests deploying at the same time, or regressions.

> As deployments rely on Git commits to push resources, your Branches **must not** be out-of-date when merging. Use you Git provider to control branch freshness.

The CD workflow explained below is a guide relevant to many of the most common use cases. However, occasional complex deployments will require additional knowledge and expertise from the team deploying the change.

Continuous Deployment helps with:

- Correctness: Ensuring you can push your changes to a Tinybird Workspace.
- Deployment: Deploying the changes to the Workspace automatically.
- Data Operations: Centralizing data operations required after resources have been pushed to the Workspace.

The following section uses the generated CD template, GitHub Actions, and the Tinybird CLI to explain how to deploy Pull Request changes after merging.

The GitHub Action

CD WORKFLOW

```
name: Tinybird - CD Workflow

on:
   workflow_dispatch:
push:
   branches:
      - main
jobs:
   cd:  # deploy changes to workspace 'web analytics starter kit'
      uses: tinybirdco/ci/.github/workflows/cd.yml@main
      with:
         data_project_dir: .
      secrets:
```

This workflow deploys on merge to `main` to the Workspace defined by the `TB_ADMIN_TOKEN` set as secret in the GitHub repository's Settings.

If your data project directory is not in the root of the Git repository, you can change the `data_project_dir` variable.

About secrets:

- `tb_host`: The url of the region you want to use. By default, this is populated with the region of the Workspace you had on `tb init --git`.
- `tb_admin_token`: The Workspace admin token. This grants all the permissions for a specific Workspace. You can find more information in the Auth Tokens docs or in the `Auth Tokens` section of the Tinybird UI.

Let's review the generated CD workflow:

The CD Workflow

The CD workflow is based on this generic Tinybird CD template that performs the following steps.

0. Workflow configuration

Same as the CI workflow.

1. Install the Tinybird CLI and check authentication

The Tinybird CLI is required to interact with your Workspace.

> You can run this workflow locally by having a local data project and the CLI authenticated to your Tinybird Workspace.

This step is equivalent to, but not identical to, the CI workflow step 1.

```
- name: Install Tinybird CLI
  run: |
   if [ -f "requirements.txt" ]; then
     pip install -r requirements.txt
   else
     pip install tinybird-cli
   fi
  - name: Tinybird version
    run: tb --version

  - name: Check auth
    run: tb --host ${{ secrets.tb_host }} --token ${{ secrets.tb_admin_token }
```

2. Deploy changes

```
  - name: Deploy changes to the main Workspace
    run: |
       DEPLOY_FILE=./deploy/${VERSION}/deploy.sh
       if [ ! -f "$DEPLOY_FILE" ]; then
          echo "$DEPLOY_FILE not found, running default tb deploy command"
          tb --semver ${VERSION} deploy ${CD_FLAGS}
          tb release ls
        fi

  - name: Custom deployment to the main Workspace
    run: |
       DEPLOY_FILE=./deploy/${VERSION}/deploy.sh
       if [ -f "$DEPLOY_FILE" ]; then
          echo "$DEPLOY_FILE found"
          if ! [ -x "$DEPLOY_FILE" ]; then
```

This step deploys the changes to the chosen release. There are three options for deployment:

- Use `tb deploy`

If the changes you pushed to your test Branch were correctly deployed, use the default `tb deploy` command. Note that doesn't perform any data operations (such as populates) - that comes in the next step.

- Use a custom shell script

This option is for when you want to run custom commands as you did in the CI step. If the file `deploy/$VERSION/deploy.sh` with the CLI commands is defined, it can be run to deploy the changes to the chosen release.

> Note the `$VERSION` variable is defined in the `.tinyenv` file. By default, it is `0.0.0`, but you can increment as needed.

3. Post-deployment operations

```
- name: Post deploy
  run: |
     POSTDEPLOY_FILE=./deploy/${VERSION}/postdeploy.sh
     if [ -f "$POSTDEPLOY_FILE" ]; then
     if ! [ -x "$POSTDEPLOY_FILE" ]; then
        echo "Error: You do not have permission to execute '$POSTDEPLOY_FILE'. F
        echo "> chmod +x $POSTDEPLOY_FILE"
        echo "and commit your changes"
        exit 1
     else
        $POSTDEPLOY_FILE
     fi
     fi
```

Once the changes have been deployed to the Workspace, you can run post-deployment actions, such as if you had to migrate data between different versions of Data Sources, or perform a specific populate action.

Create a `deploy/${VERSION}/postdeploy.sh` (as described in the previous section with the CLI commands) and run it.

How Releases work

After the Continuous Deployment process is done, you can decide what to with your current Releases, or with your new Release after your recent changes.

The following section will use the generated Release template, GitHub Actions, and the Tinybird CLI to explain how to manually manage your Releases after the CD job.

The GitHub Action

As in the previous steps, this example uses GitHub as the provider.

This is how the GitHub Action looks:

RELASE WORKFLOW

```
name: Tinybird - Releases Workflow

on:
  workflow_dispatch:
      inputs:
        job_to_run:
          description: 'Select the job to run manually'
          required: true
          default: 'promote'

  jobs:
    cd:
      uses: tinybirdco/ci/.github/workflows/release.yml@main
      with:
        job_to_run: ${{ inputs.job_to_run }}
```

You must add a new `TB_ADMIN_TOKEN` secret with the Workspace admin token to the GitHub repository's Settings.

Let's review the generated Releases workflow:

1. Manual Releases workflow

ALLOW TO RUN THE WORKFLOW MANUALLY

```
  on:
    workflow_dispatch:
      inputs:
        job_to_run:
          description: 'Select the job to run manually'
          required: true
          default: 'promote'
```

The Releases workflow isn't triggered by any event or action: A Release, or any Release jobs, must be started manually. This can be achieved either from the UI, the CLI, or the REST API. As with any other Tinybird workflow, you can edit it for your needs.

2. Releases job configuration

```
  cd: # cd using Branches from Workspace 'web_analytics_starter_kit'
    uses: tinybirdco/ci/.github/workflows/release.yml@main
    with:
      job_to_run: ${{ inputs.job_to_run }}
    secrets:
      tb_admin_token: ${{ secrets.TB_ADMIN_TOKEN }}  # set the Workspace admin
      tb_host: https://api.tinybird.co
```

You to need to choose the job, in this case `promote`, `remove` or `rollback`, to run.

About secrets:

- `tb_host`: The url of the region you want to use. By default, this is populated with the region of the Workspace you had on `tb init --git`
- `tb_admin_token`: The Workspace admin token. This grants all the permissions for a specific Workspace. You can find more information in the Auth Tokens docs or in the `Auth Tokens` section of the Tinybird UI.

The Release Workflow

The Release workflow is based on this generic Tinybird Release template that provides three different jobs:

> All jobs require some configuration steps, for instance installing Tinybird CLI (like in any other workflow).

Promote

```
- name: Promote Release
  run: |
    source .tinyenv
    tb \
    --host ${{ secrets.tb_host }} \
    --token ${{ secrets.tb_admin_token }} \
    release promote \
    --semver $VERSION
```

In this job, the chosen Release is promoted to Live.

Remove

```
- name: Remove Release
  run: |
    source .tinyenv
    tb \
    --host ${{ secrets.tb_host }} \
    --token ${{ secrets.tb_admin_token }} \
    release rm  \
    --semver $VERSION --force --yes
```

In this job, the chosen Release is removed.

A Release **cannot** be removed if it is Live.

Rollback

```
  - name: Rollback Release
    run: |
      source .tinyenv
      tb \
      --host ${{ secrets.tb_host }} \
      --token ${{ secrets.tb_admin_token }} \
      release rollback --yes
```

In this job, a chosen Release can be rolled back to a previous Release.

Congrats! You've walked through Tinybird's default Continuous Integration, Continuous Delivery, and Release process. If you want to see how these processes are applied in different use cases on Tinybird, check out the repository of common use cases and view the `.github/workflows` in each example.

Company

Product

Pricing

Resources

Docs

Blog