Slim CI/CD in Bitbucket Pipelines

1. General CI/CD rationale
2. Problem: dbt CI/CD is well-documented for GitHub and GitLab, both of which have dbt Cloud support, but some of us have to use Bitbucket for one reason or another[1], and dbt Cloud doesn't support Bitbucket.
3. Solution: Use native Bitbucket features - Pipelines and Downloads - to implement CI on every pull request and CD on every push to main.

Steps

Step 1: Database prep

1. Create a CI user with proper grants in your database
2. Either create a CI schema(s) or grant `create on [dbname]` to your CI user
3. Create a prod user with proper grants in your database (and possibly mask them under a parent role?)

Step 2: Repository prep

1. Create requirements.txt
2. Create .ci/profiles.yml and specify two targets, `ci` and `prod`. Use environmental variables as content instead of your secret values.
3. Create bitbucket-pipelines.yml, with the following contents:
    a. pull-requests: section for slim CI with artifact download
    b. branches::main: section for CD and artifact creation + upload

Step 3: Environment prep

1. Create a Bitbucket App Password with repository:write privileges
2. Set the following Bitbucket repository secrets:
    a. BITBUCKET_USER: Your username, as found in… (it's not your sign-up e-mail!)
    b. BITBUCKET_APP_PASSWORD (which you just created)
    c. All the variables in your .ci/profiles.yml
3. Enable Bitbucket Pipelines in Repository settings.

Step 4: Test

1. File a pull request. It should do a full run + test because there's no artifacts yet.
2. Approve the pull request. It should deploy to your production database and upload your artifacts.
3. Change one model and file another pull request. It should execute only runs and tests related to that model.

[1] Maybe an affirmative case for Bitbucket can be made, but I assume most people are wedded to it because that's where Jira lives?

Attached files

requirements.txt

```
dbt-postgres ~= 1.0  # change to adapter of choice
```

bitbucket-pipelines.yml

```
image: python:3.8

pipelines:
  pull-requests:
    '**':  # run on any branch
      - step:
          name: Set up and build
          caches:
            - pip
          script:
            # Set up dbt environment + dbt packages. Rather than passing
            # profiles.yml to dbt commands explicitly, we'll store it where dbt
            # expects it:
            - pip install -r requirements.txt
            - mkdir ~/.dbt
            - cp .ci/profiles.yml ~/.dbt/profiles.yml
            - dbt deps

            # The following step downloads dbt artifacts from the Bitbucket
            # Downloads, if available. (They are uploaded there by the CD
            # process -- see "Upload artifacts for slim CI runs" step below.)
            #
            # curl loop ends with "|| true" because we want downstream steps to
            # always run, even if the download fails. Running with "-L" to
            # follow the redirect to S3, -s to suppress output, --fail to avoid
            # outputting files if curl for whatever reason fails and confusing
            # the downstream conditions.
            #
            # ">-" converts newlines into spaces in a multiline YAML entry. This
            # does mean that individual bash commands have to be terminated with
            # a semicolon in order not to conflict with flow keywords (like
            # for-do-done or if-else-fi).
            - >-
              export
API_ROOT="https://api.bitbucket.org/2.0/repositories/$BITBUCKET_REPO_FULL_NAME/downloa
ds";
              mkdir target-deferred/;
              for file in manifest.json run_results.json; do
```

```
                curl -s -L --request GET \
                  -u "$BITBUCKET_USERNAME:$BITBUCKET_APP_PASSWORD" \
                  --url "$API_ROOT/$file" \
                  --fail --output target-deferred/$file;
              done || true
            - >-
              if [ -f target-deferred/manifest.json ]; then
                export DBT_FLAGS="--defer --state target-deferred/ --select
+state:modified";
              else
                export DBT_FLAGS="";
              fi

            # Finally, run dbt commands with the appropriate flag that depends
            # on whether state deferral is available. (We're skipping `dbt
            # snapshot` because only production role can write to it and it's
            # not set up otherwise.)
            - dbt seed
            - dbt run $DBT_FLAGS
            - dbt test $DBT_FLAGS
            # - dbt snapshot $DBT_FLAGS

  branches:
    main:
      - step:
          name: Deploy to production
          caches:
            - pip
          artifacts:  # Save the dbt run artifacts for the next step (upload)
            - target/*.json
          script:
            - pip install -r requirements.txt
            - mkdir ~/.dbt
            - cp .ci/profiles.yml ~/.dbt/profiles.yml
            - dbt deps
            - dbt seed --target prod
            - dbt run --target prod
            - dbt snapshot --target prod
      - step:
          name: Upload artifacts for slim CI runs
          script:
            - pipe: atlassian/bitbucket-upload-file:0.3.2
              variables:
                BITBUCKET_USERNAME: $BITBUCKET_USERNAME
                BITBUCKET_APP_PASSWORD: $BITBUCKET_APP_PASSWORD
                FILENAME: 'target/*.json'
```

.ci/profiles.yml

```yaml
your_project:
  target: ci
  outputs:
    ci:
      type: postgres
      host: "{{ env_var('DB_CI_HOST') }}"
      port: "{{ env_var('DB_CI_PORT') | int }}"
      user: "{{ env_var('DB_CI_USER') }}"
      password: "{{ env_var('DB_CI_PWD') }}"
      dbname: "{{ env_var('DB_CI_DBNAME') }}"
      schema: "{{ env_var('DB_CI_SCHEMA') }}"
      threads: 16
      keepalives_idle: 0
    prod:
      type: postgres
      host: "{{ env_var('DB_PROD_HOST') }}"
      port: "{{ env_var('DB_PROD_PORT') | int }}"
      user: "{{ env_var('DB_PROD_USER') }}"
      password: "{{ env_var('DB_PROD_PWD') }}"
      dbname: "{{ env_var('DB_PROD_DBNAME') }}"
      schema: "{{ env_var('DB_PROD_SCHEMA') }}"
      threads: 16
      keepalives_idle: 0
```

Related links

- Julia Schottenstein's blogpost on CI/CD: https://blog.getdbt.com/adopting-ci-cd-with-dbt-cloud/
- Joel Labes' post on slim CI: https://discourse.getdbt.com/t/how-we-sped-up-our-ci-runs-by-10x-using-slim-ci/2603

Article: Slim CI/CD in Bitbucket Pipelines without dbt Cloud

Continuous Integration (CI) sets the system up to test everyone's pull request before merging. Continuous Deployment (CD) deploys each approved change to production. "Slim CI" refers to running/testing only the changed code, thereby saving compute. In summary, CI/CD automates dbt pipeline testing and deployment.

dbt Cloud, a much beloved method of dbt deployment, supports GitHub- and Gitlab-based CI/CD out of the box. It doesn't support Bitbucket, AWS CodeCommit/CodeDeploy, or any number of other services. But you need not give up hope even if you're tethered to an unsupported platform.

Although this article uses Bitbucket Pipelines as the compute service and Bitbucket Downloads as the storage service, this article should serve as a blueprint for creating a dbt-based Slim CI/CD *anywhere*. The idea is always the same:

1. Deploy your product and save the deployment artifacts.
2. Use the artifacts to allow dbt to determine the stateful changes and run only those (thereby achieving "slimness").

Steps required

To accomplish this, we'll need to prepare three parts of our pipeline to work together:

1. Database,
2. Repository,
3. Bitbucket environment.

Step 1: Database preparation

In general, we want the following:

1. Create a CI user with proper grants in your database, including the ability to create the schema(s) they'll write to (`create on [dbname]`).
2. Create a prod user with proper grants in your database.

The specifics will differ based on the database type you're using. To see what constitutes "proper grants", please consult the dbt Discourse classic "[The exact grant statements we use in a dbt project]()" and Matt Mazur's "[Wrangling dbt Database Permissions]()".

In my case, I created:

1. A dev_ci Postgres user which had been granted a previously created role_dev role (same as all other development users). role_dev has connect and create grants on the database.

2. A dbt_bitbucket user which has been granted a previously created role_prod role (same as dbt Cloud prod environment). The role_prod role must have write access to your production schemas.

```sql
create role role_dev;
grant create on database [dbname] to role_dev;
-- Grant all permissions required for the development role
create role role_prod;
grant create on database [dbname] to role_prod;
– Grant all permissions required for the production role

create role dev_ci with login password ‘[password]’;
grant role_dev to dev_ci;
create schema dbt_ci;
grant all on schema dbt_ci to role_dev;
alter schema dbt_ci owner to role_dev;

create role dbt_bitbucket with login password ‘[password]’;
grant role_prod to dbt_bitbucket;
```

Finally - and this might be a Postgres-only step - I had to make sure that the regular scheduled dbt Cloud jobs connected with a dbt_cloud user with a role_prod grant would be able to drop and re-create views + tables during their run, which they could not if dbt_bitbucket had previously created and owned them. To do that, I needed to mask both roles:

```sql
alter role dbt_bitbucket set role role_prod;
alter role dbt_cloud set role role_prod;
```

That way, any tables and views created by either user would be owned by “user” role_prod.

Step 2: Repository preparation

Next, we’ll need to configure the repository. Within the repo, we’ll need to configure:

1. The pipeline environment,
2. The database connections, and
3. The pipeline itself.

Pipeline environment: requirements.txt

You’ll need at least your dbt-adapter package, ideally pinned to a version. Mine is just

```
dbt-[adapter] ~= 1.0
```

Database connections: profiles.yml

You shouldn't ever commit secrets in a plain-text file, but you can reference environmental variables (which we'll securely define in Step 3).

```yaml
your_project:
  target: ci
  outputs:
    ci:
      type: postgres
      host: "{{ env_var('DB_CI_HOST') }}"
      port: "{{ env_var('DB_CI_PORT') | int }}"
      user: "{{ env_var('DB_CI_USER') }}"
      password: "{{ env_var('DB_CI_PWD') }}"
      dbname: "{{ env_var('DB_CI_DBNAME') }}"
      schema: "{{ env_var('DB_CI_SCHEMA') }}"
      threads: 16
      keepalives_idle: 0
    prod:
      type: postgres
      host: "{{ env_var('DB_PROD_HOST') }}"
      port: "{{ env_var('DB_PROD_PORT') | int }}"
      user: "{{ env_var('DB_PROD_USER') }}"
      password: "{{ env_var('DB_PROD_PWD') }}"
      dbname: "{{ env_var('DB_PROD_DBNAME') }}"
      schema: "{{ env_var('DB_PROD_SCHEMA') }}"
      threads: 16
      keepalives_idle: 0
```

Pipeline itself: bitbucket-pipelines.yml

This is where you'll define the steps that your pipeline will take. In our case, we'll use the Bitbucket Pipelines format, but the approach will be similar for other providers.

There are two pipelines we need to configure:

1. Continuous Deployment (CD) pipeline, which will deploy and also store the run artifacts,
2. Continuous Integration (CI) pipeline, which will retrieve them for state-aware runs.

The entire file is accessible in a Gist, but we'll take it step-by-step to explain what we're doing and why.

Continuous Deployment: Transform by latest master and keep the artifacts

Each pipeline is a speedrun of setting up the environment and the database connections, then running what needs to be run. In this case, we also save the artifacts to a place we can retrieve them from - here, it's the Bitbucket Downloads service, but it could just as well be AWS S3 or another file storage service.

```yaml
image: python:3.8

pipelines:
  branches:
    main:
      - step:
          name: Deploy to production
          caches:
            - pip
          artifacts:  # Save the dbt run artifacts for the next step (upload)
            - target/*.json
          script:
            - pip install -r requirements.txt
            - mkdir ~/.dbt
            - cp .ci/profiles.yml ~/.dbt/profiles.yml
            - dbt deps
            - dbt seed --target prod
            - dbt run --target prod
            - dbt snapshot --target prod
      - step:
          name: Upload artifacts for slim CI runs
          script:
            - pipe: atlassian/bitbucket-upload-file:0.3.2
              variables:
                BITBUCKET_USERNAME: $BITBUCKET_USERNAME
                BITBUCKET_APP_PASSWORD: $BITBUCKET_APP_PASSWORD
                FILENAME: 'target/*.json'
```

Reading the file over, you can see that we:

1. Set the container image to Python 3.8,
2. Specify that we want to execute the workflow on each change to the branch called `main` (if yours is called something different, you'll want to change this),
3. Specify that this pipeline is a two-step process,
4. Specify that in the first step called "Deploy to production", we want to:
    a. Use whatever pip cache is available, if any,
    b. Keep whatever JSON files are generated in this step in target/,
    c. Run the dbt setup by first installing dbt as defined in requirements.txt, then adding profiles.yml to the location dbt expects them in, and finally running dbt deps to install any dbt packages,
    d. Run dbt seed, run, and snapshot, all with `prod` as specified target.
5. Specify that in the first step called "Upload artifacts for slim CI runs", we want to use the Bitbucket "pipe" (pre-defined action) to authenticate with environment variables and upload all files that match the glob `target/*.json`.

In summary, anytime anything is pushed to main, we'll ensure our production database reflects the dbt transformation, and we've saved the resulting artifacts to defer to.

But wait – what are artifacts and why should I defer to them? dbt artifacts are metadata of the last run - what models and tests were defined, which ones ran successfully, and which failed. If a future dbt run is set to **defer** to these metadata, it means that it can select models and tests to run based on their state, including and especially their difference from the reference metadata. See Artifacts, Selection methods: "state", and Caveats to state comparison for details.

Slim Continuous Integration: Retrieve the artifacts and do a state-based run

The Slim CI pipeline looks similar to the CD pipeline, with a couple of differences explained in the code comments. The deferral to artifacts is a key element to making the CI "slim".

```
pipelines:
  pull-requests:
    '**':  # run on any branch that's referenced by a pull request
      - step:
          name: Set up and build
          caches:
            - pip
          script:
            # Set up dbt environment + dbt packages. Rather than passing
            # profiles.yml to dbt commands explicitly, we'll store it where dbt
            # expects it:
            - pip install -r requirements.txt
            - mkdir ~/.dbt
            - cp .ci/profiles.yml ~/.dbt/profiles.yml
            - dbt deps

            # The following step downloads dbt artifacts from the Bitbucket
            # Downloads, if available. (They are uploaded there by the CD
            # process -- see "Upload artifacts for slim CI runs" step above.)
            #
            # curl loop ends with "|| true" because we want downstream steps to
            # always run, even if the download fails. Running with "-L" to
            # follow the redirect to S3, -s to suppress output, --fail to avoid
            # outputting files if curl for whatever reason fails and confusing
            # the downstream conditions.
            #
            # ">-" converts newlines into spaces in a multiline YAML entry. This
            # does mean that individual bash commands have to be terminated with
            # a semicolon in order not to conflict with flow keywords (like
            # for-do-done or if-else-fi).
            - >-
```

```
              export
API_ROOT="https://api.bitbucket.org/2.0/repositories/$BITBUCKET_REPO_FULL_NAME/downloa
ds";
              mkdir target-deferred/;
              for file in manifest.json run_results.json; do
                curl -s -L --request GET \
                  -u "$BITBUCKET_USERNAME:$BITBUCKET_APP_PASSWORD" \
                  --url "$API_ROOT/$file" \
                  --fail --output target-deferred/$file;
              done || true
            - >-
              if [ -f target-deferred/manifest.json ]; then
                export DBT_FLAGS="--defer --state target-deferred/ --select
+state:modified";
              else
                export DBT_FLAGS="";
              fi

            # Finally, run dbt commands with the appropriate flag that depends
            # on whether state deferral is available. (We're skipping `dbt
            # snapshot` because only production role can write to it and it's
            # not set up otherwise.)
            - dbt seed
            - dbt run $DBT_FLAGS
            - dbt test $DBT_FLAGS
```

In short, we:

1. Set up the pipeline trigger condition to trigger on any pull request,
2. Set up dbt,
3. Retrieve the files from Bitbucket Downloads via API and credentials,
4. Set flags for state deferral if the retrieval was successful,
5. Run dbt with the default target (which we'd defined in `` `profiles.yml` `` as `` `ci` ``).

Step 3: Bitbucket environment preparation

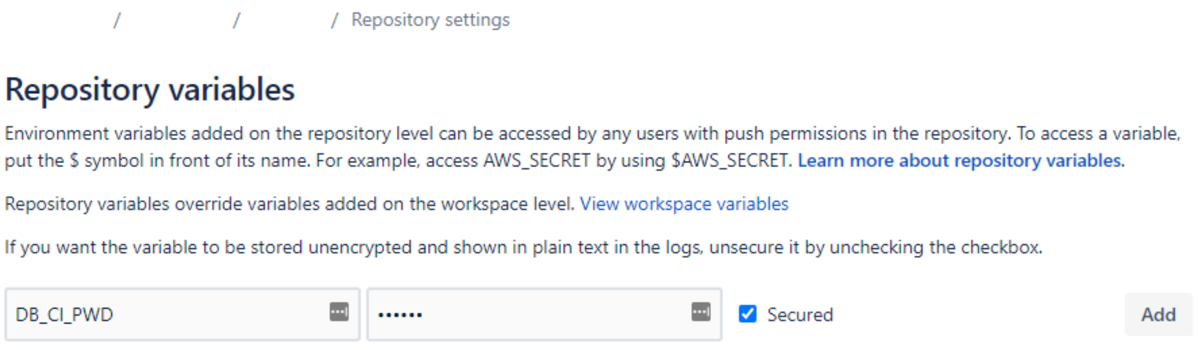
Finally, we need to make sure that all the steps that require authentication succeed:

1. Database authentication, and
2. Bitbucket Downloads authentication.

Database authentication

1. Determine the values of all of the variables in .ci/profiles.yml (`DB_{CI,PROD}_{HOST,PORT,USER,PWD,DBNAME,SCHEMA}`)

2. Go to Repository > Repository Settings > Repository Variables and define them there, making sure to store any confidential values as “Secured”.


/ / / Repository settings

Repository variables

Environment variables added on the repository level can be accessed by any users with push permissions in the repository. To access a variable, put the $ symbol in front of its name. For example, access AWS_SECRET by using $AWS_SECRET. Learn more about repository variables.

Repository variables override variables added on the workspace level. View workspace variables

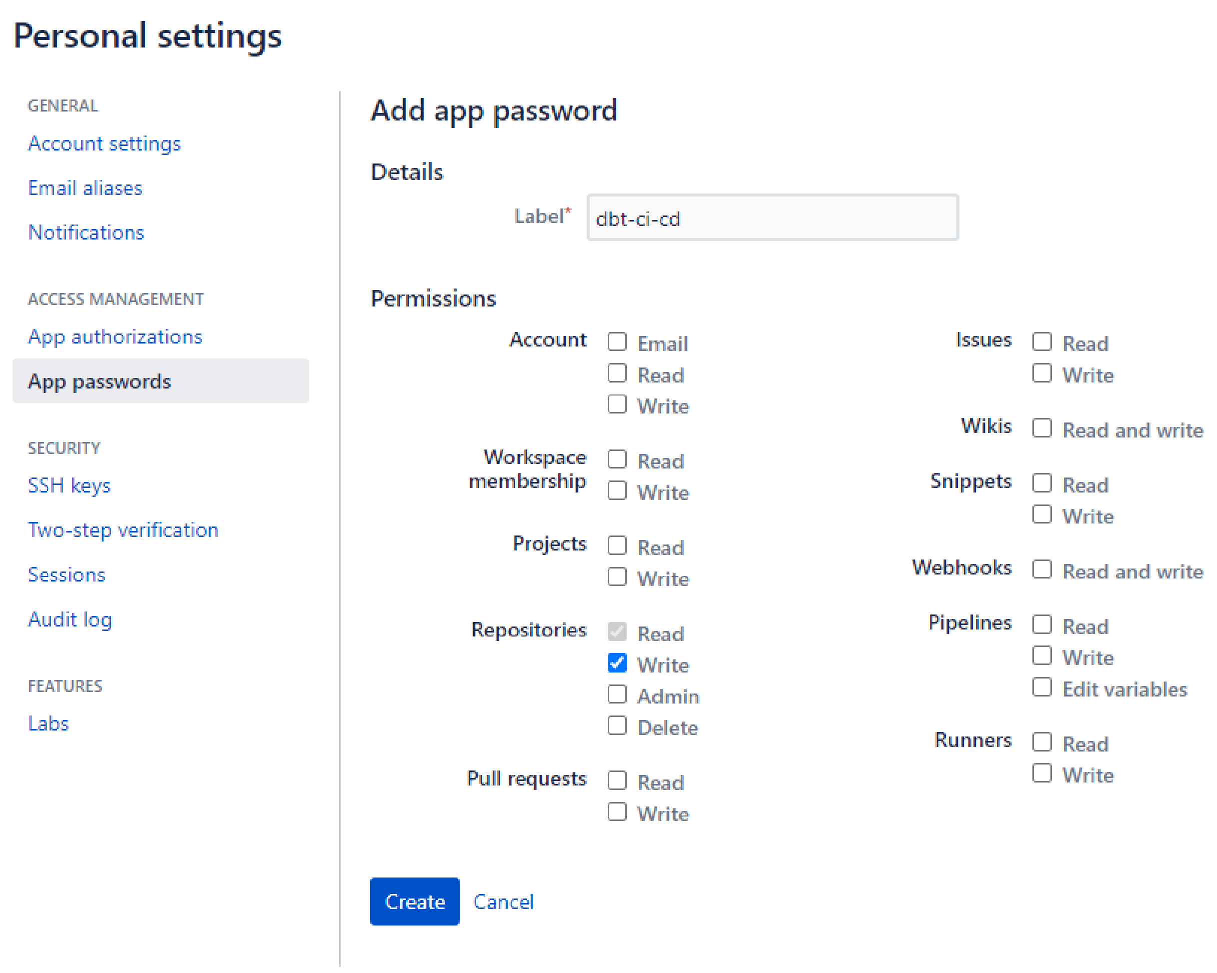
If you want the variable to be stored unencrypted and shown in plain text in the logs, unsecure it by unchecking the checkbox.

DB_CI_PWD •••••• ☑ Secured Add

Bitbucket Downloads authentication

1. Go to Personal Settings > App Passwords and create a Bitbucket App Password with scope `repository:write`.

2. Go to Repository > Repository Settings > Repository Variables and define the following:
    a. `BITBUCKET_USERNAME`, which is <u>not</u> your sign-up e-mail, but rather the username found by clicking your avatar in the top left > Personal settings > Account settings page, under Bitbucket Profile Settings.
    b. `BITBUCKET_APP_PASSWORD`, making sure to store it as “Secured”



Personal settings

GENERAL
Account settings
Email aliases
Notifications

ACCESS MANAGEMENT
App authorizations
App passwords

SECURITY
SSH keys
Two-step verification
Sessions
Audit log

FEATURES
Labs

Add app password

Details

Label*: dbt-ci-cd

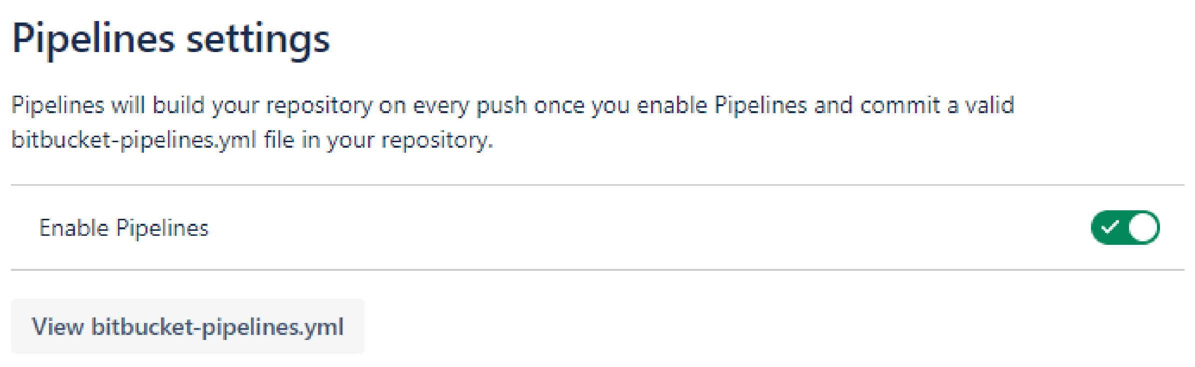
Permissions

Account	☐ Email ☐ Read ☐ Write	Issues	☐ Read ☐ Write
		Wikis	☐ Read and write
Workspace membership	☐ Read ☐ Write	Snippets	☐ Read ☐ Write
Projects	☐ Read ☐ Write	Webhooks	☐ Read and write
Repositories	✓ Read ✓ Write ☐ Admin ☐ Delete	Pipelines	☐ Read ☐ Write ☐ Edit variables
		Runners	☐ Read ☐ Write
Pull requests	☐ Read ☐ Write		

Create Cancel

Enable Bitbucket Pipelines

Lastly, under Repository > Repository Settings > Pipelines Settings, check “Enable Pipelines”.


Pipelines settings

Pipelines will build your repository on every push once you enable Pipelines and commit a valid bitbucket-pipelines.yml file in your repository.

Enable Pipelines ✓

Step 4: Test

You're all done! Now it's time to test that things work:

1. Push a change to your main branch. This should trigger a Pipeline. Check that it's successful.
2. File a Pull Request with a change to a single model / addition of a single test. Check that only that model/test ran.

Conclusion

It's important to remember that CI/CD is a convenience, not a panacea. You must still devise the model logic and determine the appropriate tests. Some things it can do, though: catch some mistakes early, make sure that the database always reflects the code, and decrease the friction in collaboration. By automating the steps that should *always* be taken, it frees you up to think about the unusual steps required (e.g., do your changes to incremental models require an additional deployment with `--full-refresh`?) and reduces the amount of review that others' actions necessitate.

Plus, it's a good time, and it's fun to watch the test lights turn green. Ding!