Overview

> info **Hint** This chapter covers the Nest Devtools integration with the Nest framework. If you are looking for the Devtools application, please visit the Devtools website.

To start debugging your local application, open up the `main.ts` file and make sure to set the `snapshot` attribute to `true` in the application options object, as follows:

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule, {
    snapshot: true,
  });
  await app.listen(3000);
}
```

This will instruct the framework to collect necessary metadata that will let Nest Devtools visualize your application's graph.

Next up, let's install the required dependency:

```
$ npm i @nestjs/devtools-integration
```

> warning **Warning** If you're using `@nestjs/graphql` package in your application, make sure to install the latest version (`npm i @nestjs/graphql@11`).

With this dependency in place, let's open up the `app.module.ts` file and import the `DevtoolsModule` that we just installed:

```
@Module({
  imports: [
    DevtoolsModule.register({
      http: process.env.NODE_ENV !== 'production',
    }),
  ],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

> warning **Warning** The reason we are checking the `NODE_ENV` environment variable here is that you should never use this module in production!

Once the `DevtoolsModule` is imported and your application is up and running (`npm run start:dev`), you should be able to navigate to Devtools URL and see the instrospected graph.

> info **Hint** As you can see on the screenshot above, every module connects to the `InternalCoreModule`. `InternalCoreModule` is a global module that is always imported into the root module. Since it's registered as a global node, Nest automatically creates edges between all of the modules and the `InternalCoreModule` node. Now, if you want to hide global modules from the graph, you can use the "**Hide global modules**" checkbox (in the sidebar).

So as we can see, `DevtoolsModule` makes your application expose an additional HTTP server (on port 8000) that the Devtools application will use to introspect your app.

Just to double-check that everything works as expected, change the graph view to "Classes". You should see the following screen:

To focus on a specific node, click on the rectangle and the graph will show a popup window with the **"Focus"** button. You can also use the search bar (located in the sidebar) to find a specific node.

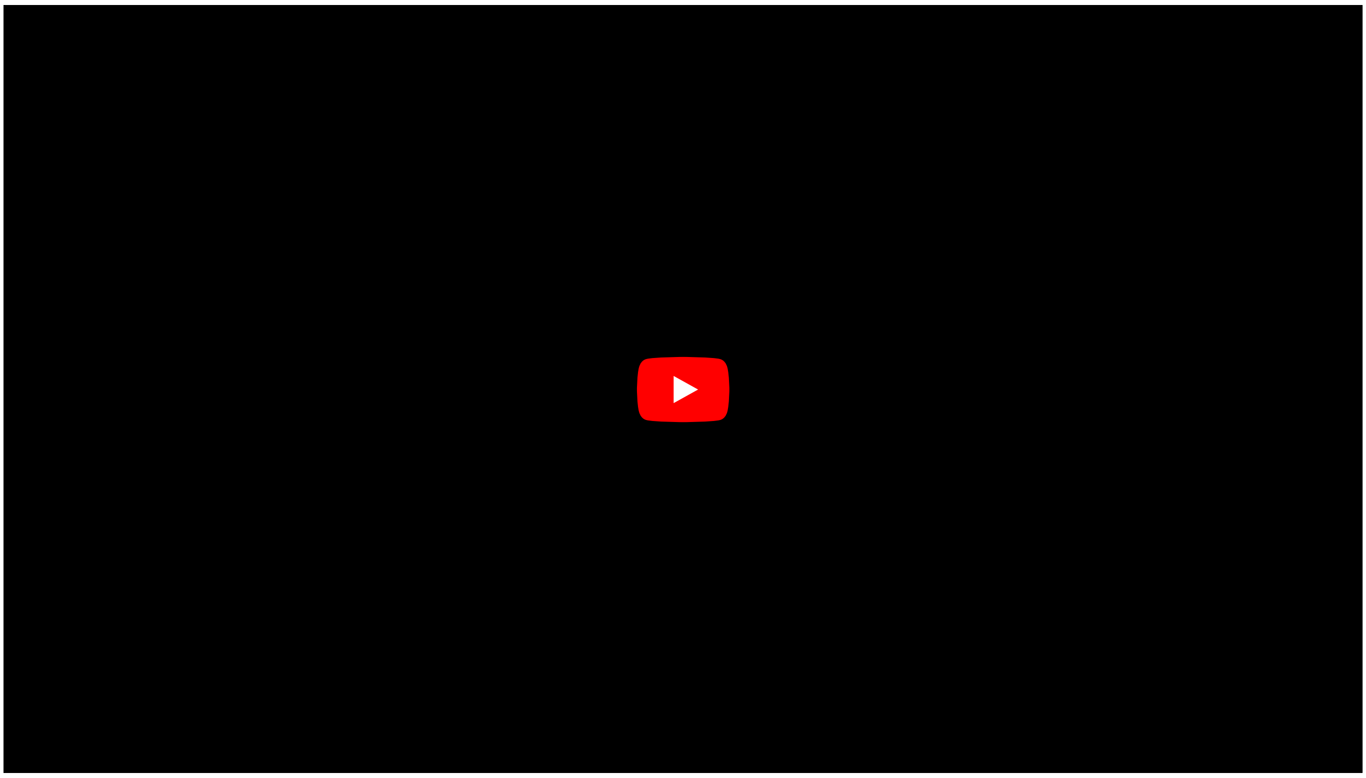
> info **Hint** If you click on the **Inspect** button, application will take you to the `/debug` page with that specific node selected.

> info **Hint** To export a graph as an image, click on the **Export as PNG** button in the right corner of the graph.

Using the form controls located in the sidebar (on the left), you can control edges proximity to, for example, visualize a specific application sub-tree:

This can be particularly useful when you have **new developers** on your team and you want to show them how your application is structured. You can also use this feature to visualize a specific module (e.g. `TasksModule`) and all of its dependencies, which can come in handy when you're breaking down a large application into smaller modules (for example, individual micro-services).

You can watch this video to see the **Graph Explorer** feature in action:



Investigating the "Cannot resolve dependency" error

> info **Note** This feature is supported for `@nestjs/core` >= `v9.3.10`.

Probably the most common error message you might have seen is about Nest not being able to resolve dependencies of a provider. Using Nest Devtools, you can effortlessly identify the issue and learn how to resolve it.

First, open up the `main.ts` file and update the `bootstrap()` call, as follows:

```
bootstrap().catch((err) => {
  fs.writeFileSync('graph.json', PartialGraphHost.toString() ?? '');
  process.exit(1);
});
```

Also, make sure to set the `abortOnError` to `false`:

```
const app = await NestFactory.create(AppModule, {
  snapshot: true,
  abortOnError: false, // <--- THIS
});
```

Now every time your application fails to bootstrap due to the **"Cannot resolve dependency"** error, you'll find the `graph.json` (that represents a partial graph) file in the root directory. You can then drag & drop this file into Devtools (make sure to switch the current mode from "Interactive" to "Preview"):

Upon successful upload, you should see the following graph & dialog window:

As you can see, the highlighted `TasksModule` is the one we should look into. Also, in the dialog window you can already see some instructions on how to use fix this issue.

If we switch to the "Classes" view instead, that's what we'll see:

This graph illustrates that the `DiagnosticsService` which we want to inject into the `TasksService` was not found in the context of the `TasksModule` module, and we should likely just import the `DiagnosticsModule` into the `TasksModule` module to fix this up!

Routes explorer

When you navigate to the **Routes explorer** page, you should see all of the registered entrypoints:

> info **Hint** This page shows not only HTTP routes, but also all of the other entrypoints (e.g. WebSockets, gRPC, GraphQL resolvers etc.).

Entrypoints are grouped by their host controllers. You can also use the search bar to find a specific entrypoint.

If you click on a specific entrypoint, **a flow graph** will be displayed. This graph shows the execution flow of the entrypoint (e.g. guards, interceptors, pipes, etc. bound to this route). This is particularly useful when you want to understand how the request/response cycle looks for a specific route, or when troubleshooting why a specific guard/interceptor/pipe is not being executed.

Sandbox

To execute JavaScript code on the fly & interact with your application in real-time, navigate to the **Sandbox** page:

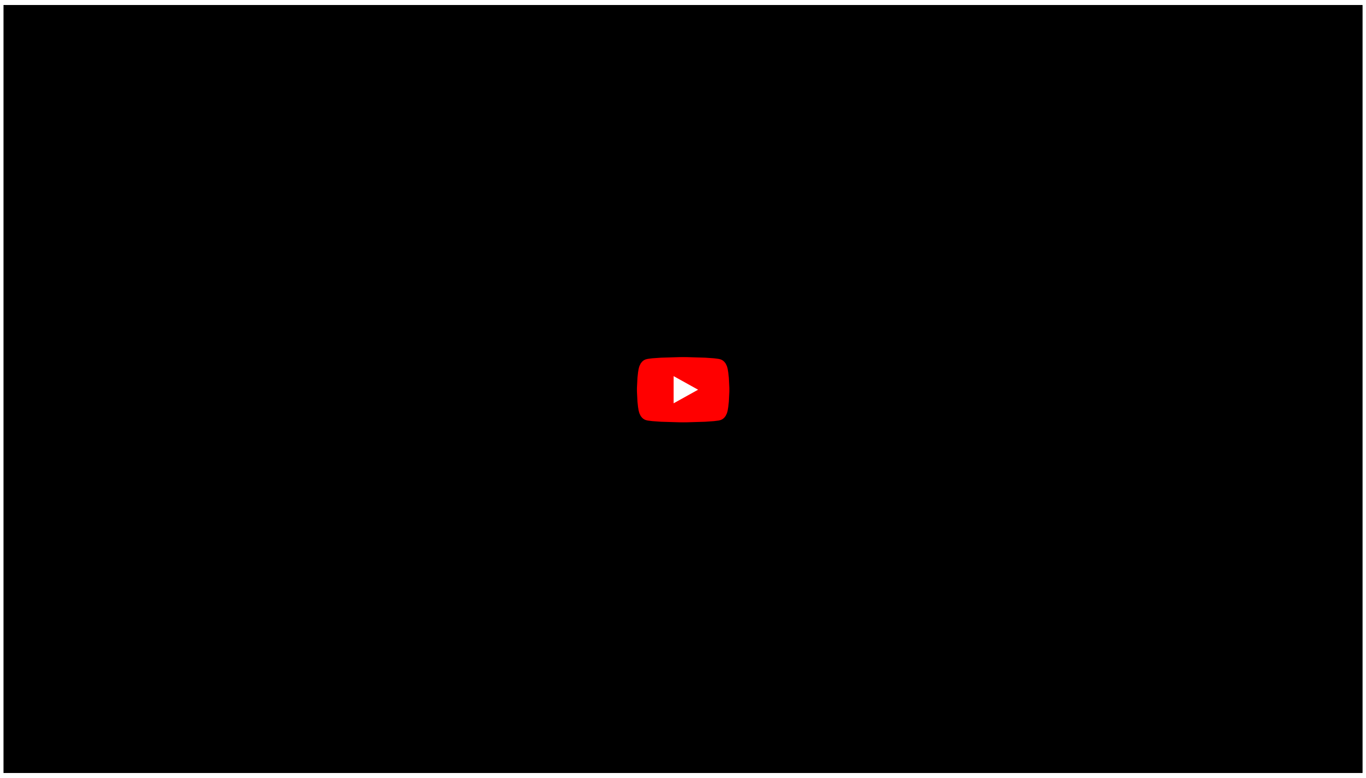
The playground can be used to test and debug API endpoints in **real-time**, allowing developers to quickly identify and fix issues without using, for example, an HTTP client. We can also bypass the authentication layer, and so we no longer need that extra step of logging in, or even a special user account for testing purposes. For event-driven applications, we can also trigger events directly from the playground, and see how the application reacts to them.

Anything that gets logged down is streamlined to the playground's console, so we can easily see what's going on.

Just execute the code **on the fly** and see the results instantly, without having to rebuild the application and restart the server.

> info **Hint** To pretty display an array of objects, use the `console.table()` (or just `table()`) function.

You can watch this video to see the **Interactive Playground** feature in action:



Bootstrap performance analyzer

To see a list of all class nodes (controllers, providers, enhancers, etc.) and their corresponding instantiation times, navigate to the **Bootstrap performance** page:

This page is particularly useful when you want to identify the slowest parts of your application's bootstrap process (e.g. when you want to optimize the application's startup time which is crucial for, for example, serverless environments).

Audit

To see the auto-generated audit - errors/warnings/hints that the application came up with while analyzing your serialized graph, navigate to the **Audit** page:

> info **Hint** The screenshot above doesn't show all of the available audit rules.

This page comes in handy when you want to identify potential issues in your application.

Preview static files

To save a serialized graph to a file, use the following code:

```
await app.listen(3000); // OR await app.init()
fs.writeFileSync('./graph.json', app.get(SerializedGraph).toString());
```

> info **Hint** `SerializedGraph` is exported from the `@nestjs/core` package.

Then you can drag and drop/upload this file:

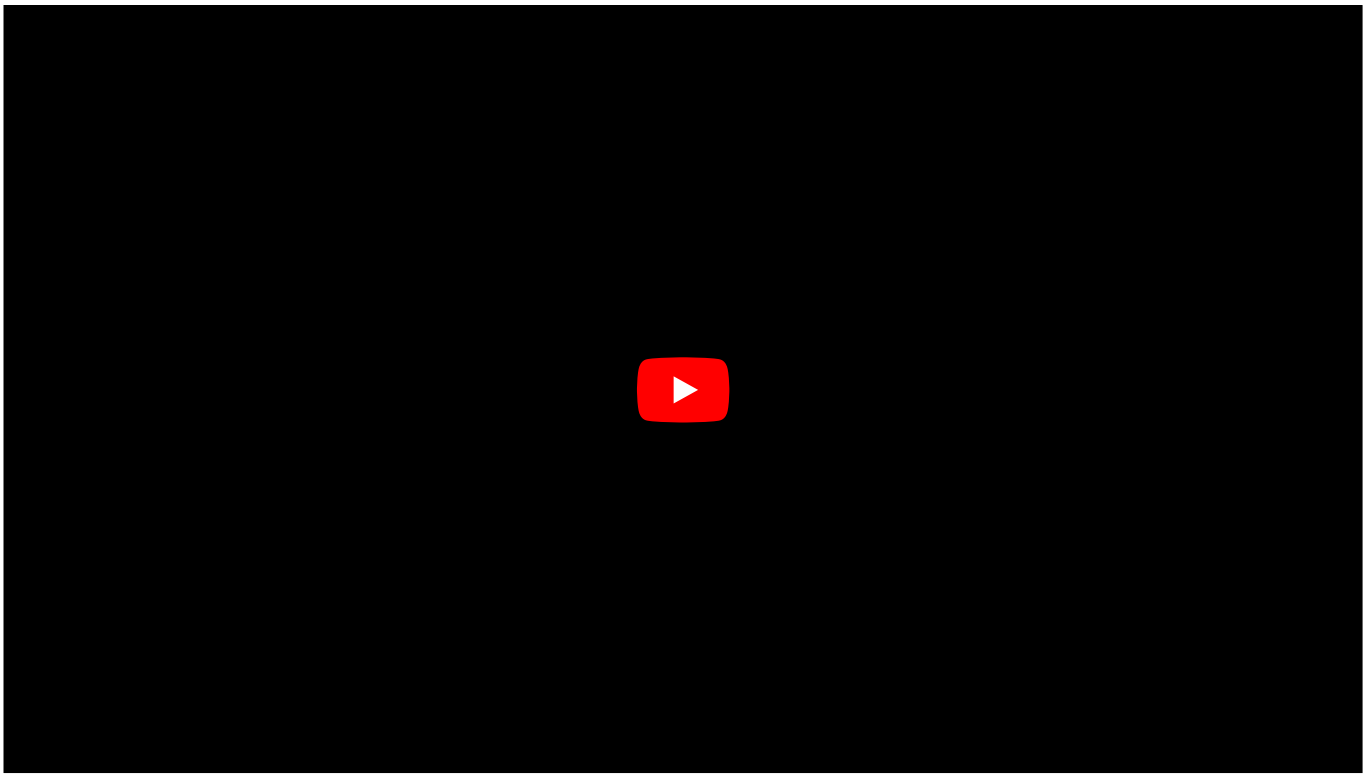
This is helpful when you want to share your graph with someone else (e.g., co-worker), or when you want to analyze it offline.

CI/CD integration

> info **Hint** This chapter covers the Nest Devtools integration with the Nest framework. If you are looking for the Devtools application, please visit the Devtools website.

CI/CD integration is available for users with the **Enterprise** plan.

You can watch this video to learn why & how CI/CD integration can help you:



Publishing graphs

Let's first configure the application bootstrap file (`main.ts`) to use the `GraphPublisher` class (exported from the `@nestjs/devtools-integration` - see previous chapter for more details), as follows:

```
async function bootstrap() {
  const shouldPublishGraph = process.env.PUBLISH_GRAPH === "true";

  const app = await NestFactory.create(AppModule, {
    snapshot: true,
    preview: shouldPublishGraph,
  });

  if (shouldPublishGraph) {
    await app.init();

    const publishOptions = { ... } // NOTE: this options object will vary depending on the CI/CD provider you're
using
    const graphPublisher = new GraphPublisher(app);
    await graphPublisher.publish(publishOptions);

    await app.close();
  } else {
    await app.listen(3000);
  }
}
```

As we can see, we're using the `GraphPublisher` here to publish our serialized graph to the centralized registry. The `PUBLISH_GRAPH` is a custom environment variable that will let us control whether the graph should be published (CI/CD workflow), or not (regular application bootstrap). Also, we set the `preview` attribute here to `true`. With this flag enabled, our application will bootstrap in the preview mode - which basically means that constructors (and lifecycle hooks) of all controllers, enhancers, and providers in our application will not be executed. Note - this isn't **required**, but makes things simpler for us since in this case we won't really have to connect to the database etc. when running our application in the CI/CD pipeline.

The `publishOptions` object will vary depending on the CI/CD provider you're using. We will provide you with instructions for the most popular CI/CD providers below, in later sections.

Once the graph is successfully published, you'll see the following output in your workflow view:

Every time our graph is published, we should see a new entry in the project's corresponding page:

Reports

Devtools generate a report for every build **IF** there's a corresponding snapshot already stored in the centralized registry. So for example, if you create a PR against the `master` branch for which the graph was already published - then the application will be able to detect differences and generate a report. Otherwise, the report will not be generated.

To see reports, navigate to the project's corresponding page (see organizations).

This is particularly helpful in identifying changes that may have gone unnoticed during code reviews. For instance, let's say someone has changed the scope of a **deeply nested provider**. This change might not be immediately obvious to the reviewer, but with Devtools, we can easily spot such changes and make sure that they're intentional. Or if we remove a guard from a specific endpoint, it will show up as affected in the report. Now if we didn't have integration or e2e tests for that route, we might not notice that it's no longer protected, and by the time we do, it could be too late.

Similarly, if we're working on a **large codebase** and we modify a module to be global, we'll see how many edges were added to the graph, and this - in most cases - is a sign that we're doing something wrong.

Build preview

For every published graph we can go back in time and preview how it looked before by clicking at the **Preview** button. Furthermore, if the report was generated, we should see the differences higlighted on our graph:

- green nodes represent added elements
- light white nodes represent updated elements
- red nodes represent deleted elements

See screenshot below:

The ability to go back in time lets you investigate and troubleshoot the issue by comparing the current graph with the previous one. Depending on how you set things up, every pull request (or even every commit) will have a corresponding snapshot in the registry, so you can easily go back in time and see what changed. Think of Devtools as a Git but with an understanding of how Nest constructs your application graph, and with the ability to **visualize** it.

Integrations: Github Actions

First let's start from creating a new Github workflow in the `.github/workflows` directory in our project and call it, for example, `publish-graph.yml`. Inside this file, let's use the following definition:

```
name: Devtools

on:
  push:
    branches:
      - master
  pull_request:
    branches:
      - '*'

jobs:
  publish:
    if: github.actor!= 'dependabot[bot]'
    name: Publish graph
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '16'
          cache: 'npm'
      - name: Install dependencies
        run: npm ci
      - name: Setup Environment (PR)
        if: {{ '${{' }} github.event_name == 'pull_request' {{ '}}' }}
        shell: bash
        run: |
          echo "COMMIT_SHA={{ '${{' }} github.event.pull_request.head.sha {{ '}}' }}" >>\${GITHUB_ENV}
      - name: Setup Environment (Push)
```

```
        if: {{ '${{' }} github.event_name == 'push' {{ '}}' }}
        shell: bash
        run: |
          echo "COMMIT_SHA=\${GITHUB_SHA}" >> \${GITHUB_ENV}
      - name: Publish
        run: PUBLISH_GRAPH=true npm run start
        env:
          DEVTOOLS_API_KEY: CHANGE_THIS_TO_YOUR_API_KEY
          REPOSITORY_NAME: {{ '${{' }} github.event.repository.name {{ '}}' }}
          BRANCH_NAME: {{ '${{' }} github.head_ref || github.ref_name {{ '}}' }}
          TARGET_SHA: {{ '${{' }} github.event.pull_request.base.sha {{ '}}' }}
```

Ideally, `DEVTOOLS_API_KEY` environment variable should be retrieved from Github Secrets, read more here .

This workflow will run per each pull request that's targeting the `master` branch OR in case there's a direct commit to the `master` branch. Feel free to align this configuration to whatever your project needs. What's essential here is that we provide necessary environment varaiables for our `GraphPublisher` class (to run).

However, there's one variable that needs to be updated before we can start using this workflow - `DEVTOOLS_API_KEY`. We can generate an API key dedicated for our project on this **page** .

Lastly, let's navigate to the `main.ts` file again and update the `publishOptions` object we previously left empty.

```
const publishOptions = {
  apiKey: process.env.DEVTOOLS_API_KEY,
  repository: process.env.REPOSITORY_NAME,
  owner: process.env.GITHUB_REPOSITORY_OWNER,
  sha: process.env.COMMIT_SHA,
  target: process.env.TARGET_SHA,
  trigger: process.env.GITHUB_BASE_REF ? 'pull' : 'push',
  branch: process.env.BRANCH_NAME,
};
```

For the best developer experience, make sure to integrate the **Github application** for your project by clicking on the "Integrate Github app" button (see screenshot below). Note - this isn't required.

With this integration, you'll be able to see the status of the preview/report generation process right in your pull request:

Integrations: Gitlab Pipelines

First let's start from creating a new Gitlab CI configuration file in the root directory of our project and call it, for example, `.gitlab-ci.yml`. Inside this file, let's use the following definition:

```
const publishOptions = {
  apiKey: process.env.DEVTOOLS_API_KEY,
  repository: process.env.REPOSITORY_NAME,
  owner: process.env.GITHUB_REPOSITORY_OWNER,
  sha: process.env.COMMIT_SHA,
  target: process.env.TARGET_SHA,
  trigger: process.env.GITHUB_BASE_REF ? 'pull' : 'push',
  branch: process.env.BRANCH_NAME,
};
```

> info **Hint** Ideally, `DEVTOOLS_API_KEY` environment variable should be retrieved from secrets.

This workflow will run per each pull request that's targeting the `master` branch OR in case there's a direct commit to the `master` branch. Feel free to align this configuration to whatever your project needs. What's essential here is that we provide necessary environment variables for our `GraphPublisher` class (to run).

However, there's one variable (in this workflow definition) that needs to be updated before we can start using this workflow - `DEVTOOLS_API_KEY`. We can generate an API key dedicated for our project on this **page** .

Lastly, let's navigate to the `main.ts` file again and update the `publishOptions` object we previously left empty.

```
image: node:16

stages:
  - build

cache:
  key:
    files:
      - package-lock.json
  paths:
    - node_modules/

workflow:
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
      when: always
    - if: $CI_COMMIT_BRANCH == "master" && $CI_PIPELINE_SOURCE == "push"
      when: always
    - when: never

install_dependencies:
  stage: build
  script:
    - npm ci

publish_graph:
  stage: build
  needs:
    - install_dependencies
  script: npm run start
  variables:
    PUBLISH_GRAPH: 'true'
    DEVTOOLS_API_KEY: 'CHANGE_THIS_TO_YOUR_API_KEY'
```

Other CI/CD tools

Nest Devtools CI/CD integration can be used with any CI/CD tool of your choice (e.g., Bitbucket Pipelines , CircleCI, etc) so don't feel limited to providers we described here.

Look at the following `publishOptions` object configuration to understand what information is required to publish the graph for a given commit/build/PR.

```
const publishOptions = {
  apiKey: process.env.DEVTOOLS_API_KEY,
  repository: process.env.CI_PROJECT_NAME,
  owner: process.env.CI_PROJECT_ROOT_NAMESPACE,
  sha: process.env.CI_COMMIT_SHA,
  target: process.env.CI_MERGE_REQUEST_DIFF_BASE_SHA,
  trigger: process.env.CI_MERGE_REQUEST_DIFF_BASE_SHA ? 'pull' : 'push',
  branch:
    process.env.CI_COMMIT_BRANCH ??
    process.env.CI_MERGE_REQUEST_SOURCE_BRANCH_NAME,
};
```

Most of this information is provided through CI/CD built-in environment variables (see CircleCI built-in environment list and Bitbucket variables).

When it comes to the pipeline configuration for publishing graphs, we recommend using the following triggers:

- `push` event - only if the current branch represents a deployment environment, for example `master`, `main`, `staging`, `production`, etc.
- `pull request` event - always, or when the **target branch** represents a deployment environment (see above)