Deploy a Java application to Tomcat using Octopus and GitHub Actions

In this tutorial, we show you how to build a fully-functional continuous delivery pipeline for a simple Java web application and deploy it to Tomcat. We use GitHub Actions to build the code and run tests, and we use Octopus Deploy to deploy and promote releases.

To get up and running quickly, the **TestDrive VMs** provide preconfigured environments demonstrating various continuous delivery pipelines documented in these guides.

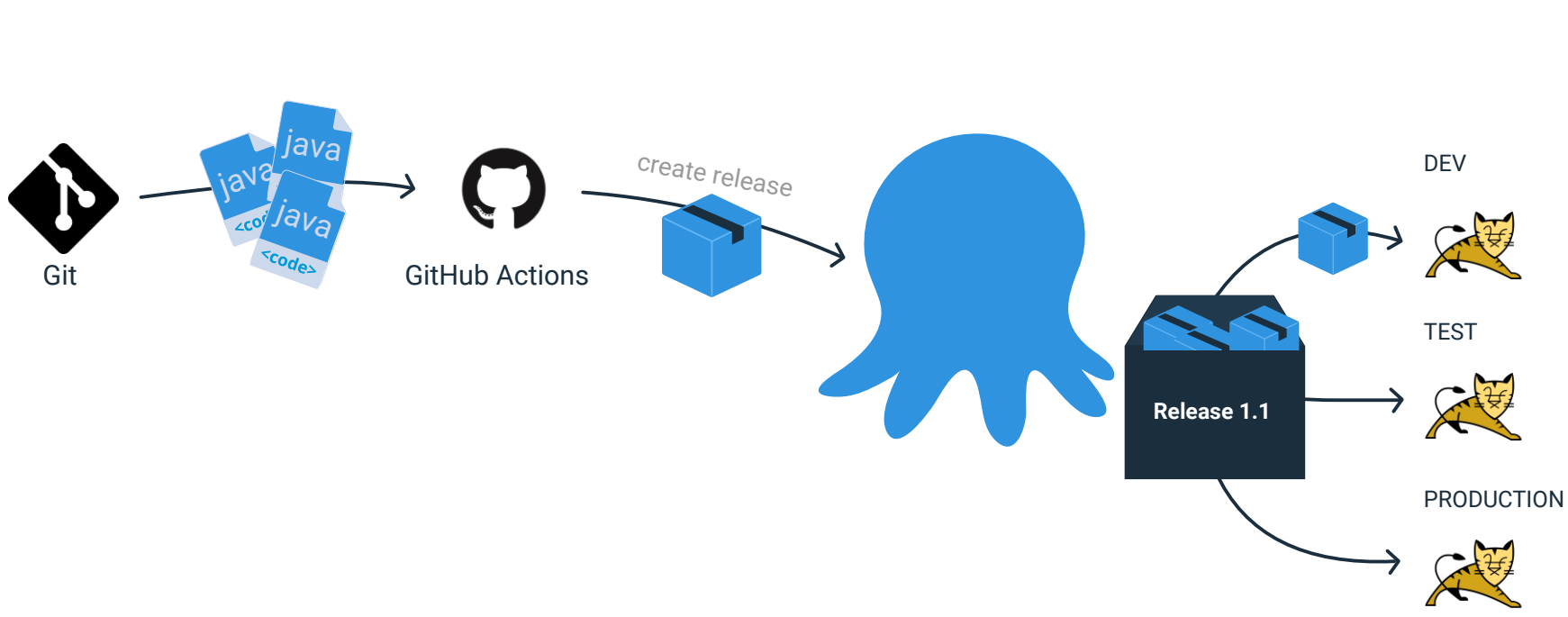
TestDrive VMs →

Introduction

The application we'll deploy is called **Random Quotes**, which is a simple web application that randomly displays a famous quote each time the page loads. It consists of a web front end and a database that contains the quotes. We'll build a complete Continuous Integration/Continuous Delivery (CI/CD) pipeline with automated builds, deployments to a dev environment, and sign offs for production deployments.

Deployment pipeline

For this tutorial, we assume you use **Git** for version controlling changes to your source code and **GitHub Actions** to compile code and run unit tests. **Octopus Deploy** will take care of the deployment. Here is what the full continuous integration and delivery pipeline will look like when we are finished:



The development team's workflow is:

running unit tests.

3 When the GitHub Actions build completes, the change will be deployed to the **Dev** environment.

4 When one of your team members (perhaps a tester) wants to see what's in a particular release, they can use Octopus to manually deploy a release to the **Test** environment.

5 When the team is satisfied with the quality of the release and they are ready for it to go to production, they use Octopus to promote the release from the **Test** environment to the **Production** environment.

Since Octopus is designed to be used by teams, in this tutorial we also set up some simple rules:

- Anyone can deploy to the dev or test environments.
- Only specific people can deploy to production.
- Production deployments require sign off from someone in our project stakeholders group.
- We'll send an email to the team after any test or production deployment has succeeded or failed.

This tutorial makes use of the following tools:

Git

Source control

Version control of source code

GitHub Actions

Build server

Compile code, run tests

Octopus Deploy

Deployment server

Release management & deployments

Tomcat

Java application server

Hosts the running web application

Octopus is an extremely powerful deployment automation tool, and there are numerous ways to model a development team's workflow in Octopus Deploy, but this tends to be the most common for small teams. If you're not sure how to configure Octopus, we recommend following this guide to learn the basics. You'll then know how to adjust Octopus to suit your team's workflow.

This tutorial takes about an hour to complete. That sounds like a long time, but keep in mind, at the end of the tutorial, you'll have a fully-functional CI/CD environment for your entire team, and you'll be ready to deploy to production at the click of a button. It's worth the effort!

Build vs. deployment

For any non-trivial application, you're going to deploy the software to multiple environments. For this tutorial, we're using the environments **Dev**, **Test**, and **Prod**. This means you need to choose between building your application once or building it before each deployment? To reduce the risk of a failed production deployment, Octopus strongly encourages the practice of building once, and deploying multiple times.

The following activities are a **build time** concern, so they will happen in GitHub Actions after any change to code is committed to Git:

1. Check out the latest changes from Git.
2. Resolve and install any dependencies from Maven.
3. Run unit tests.
4. Package the application by bundling all the files it needs to run into a WAR file.

This results in a green CI build and a file that contains the application and everything it needs to run. Any configuration files will have their default values, but they won't know anything about dev vs. production settings just yet.

An example of a package that is ready to be deployed is:

```
randomquotes.1.0.0.war
```

At this point, we have a single *artifact* that contains all the files our application needs to run, ready to be deployed. We can deploy it over and over, using the same artifact in each environment. If we deploy a bad release, we can go and find the older version of the artifact and re-deploy it.

The following activities happen at **deployment time** by Octopus Deploy:

1. Changing any configuration files to include settings appropriate for the environment, e.g., database connection strings, API keys, etc.
2. Running tasks that need to be performed during the deployment such as database migrations or taking the application temporarily offline.

Prerequisites

There are a number of tools you need to install to implement a complete CI/CD workflow. These include the GitHub Actions and Octopus servers, some command-line tools, and Tomcat to host the final deployment.

Git

The source code for the sample application is hosted on GitHub. To access the code, you need the Git client. The Git documentation has instructions to download and install the Git client.

GitHub Actions

GitHub Actions are available for free on public GitHub repsoitories and with paid options for private repositories.

Java

The Java Developer Kit (JDK) is required to be installed to build and test Java applications. OpenJDK builds are freely available for all platforms.

Maven

The Java application is built using Maven. Maven is available via Chocolatey for Windows, and it can be installed via the appropriate package manager for Linux.

Octopus Deploy

The installation file used to install Octopus Server on-premises can be downloaded from the Octopus website.

Octopus requires a Microsoft SQL Server to host the database. Microsoft SQL Server Express can be downloaded for free from the Microsoft website.

Complete the following steps to install Octopus:

1. Start the Octopus Installer, click **Next**, accept the Terms in the License Agreement and click **Next**.
2. Accept the default Destination Folder or choose a different location and click **Next**.
3. Click **Install**, and give the app permission to make changes to your device.
4. Click **Finish** to exit the installation wizard and launch the Getting started wizard to configure your Octopus Server.
5. Click **Get started...** and either enter your details to start a free trial of Octopus Deploy or enter your license key and click **Next**.
6. Accept the default Home Directory or enter a location of your choice and click **Next**.
7. Decide whether to use a **Local System Account** or a **Custom Domain Account**.

Learn more about the permissions required for the Octopus Windows Service or using a Managed Service Account.

8. On the Database page, click the dropdown arrow in the Server Name field to detect the SQL Server database. Octopus will create the database for you which is the recommended process; however, you can also create your own database.
9. Enter a name for the database, and click **Next** and **OK** to create the database.

Be careful not to use the name of an existing database as the setup process will install Octopus into that pre-existing database.

10. Accept the default port and directory or enter your own. This guide assumes Octopus is listening on port 80. Click **Next**.

13. When the installation has completed, click **Finish** to launch the Octopus Manager.

Tomcat

Tomcat is an open source Java application server. It can be downloaded from the product homepage, installed via Chocolatey, or installed via a Linux package manager.

Deployments to Tomcat are done via the Manager. This application is included in some Tomcat distributions, but may need to be installed separately. For example, Ubuntu distributes Tomcat and the Manager application as the **tomcat9** and **tomcat9-admin** packages.

A user account must be added to the Manager application to allow deployments to be performed. The following is an example of a **tomcat-users.xml** file that defines a user called **tomcat** that belongs to the roles **manager-script**, which allows access to the Manager REST API, and **manager-gui**, which allows access to the Manager web interface:

```
<?xml version="1.0" encoding="UTF-8"?>
 <tomcat-users xmlns="http://tomcat.apache.org/xml"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="http://tomcat.apache.org/xml tomcat-users.xsd"
     <role rolename="manager-script"/>
     <role rolename="manager-gui"/>
     <user username="tomcat" password="Password01!" roles="manager-script,manager-gui"/>
 </tomcat-users>
```

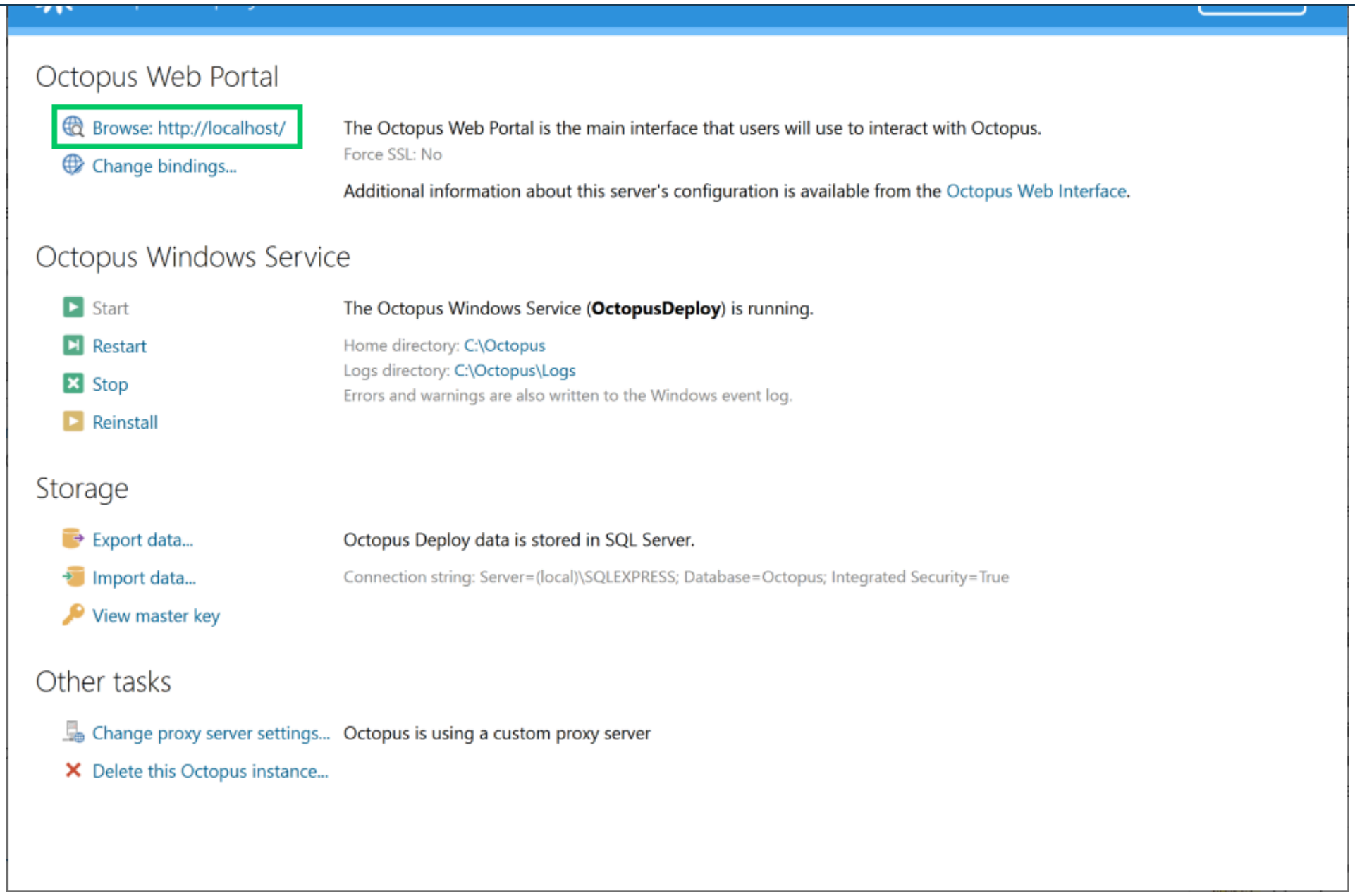
Clone source code

The source code for the random quotes application is hosted in GitHub. The code can be cloned from https://github.com/OctopusSamples/RandomQuotes-Java.git with the command:

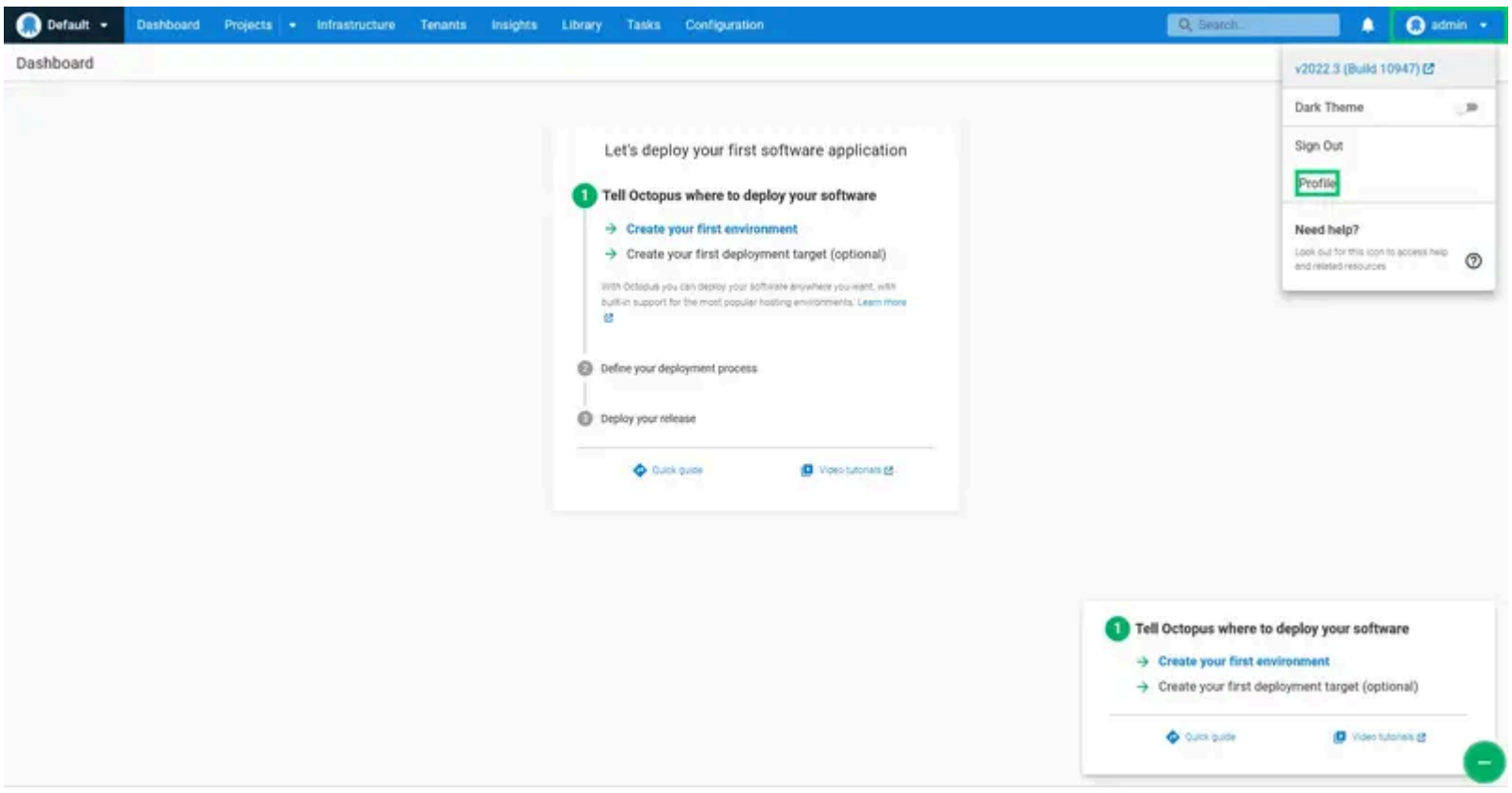
```
git clone https://github.com/OctopusSamples/RandomQuotes-Java.git
```

Octopus API key

In order to allow GitHub Actions to communicate with Octopus, we need to generate an API key. This is done in the Octopus web portal. The web portal can be opened from the **Browse** link in the Octopus Manager:

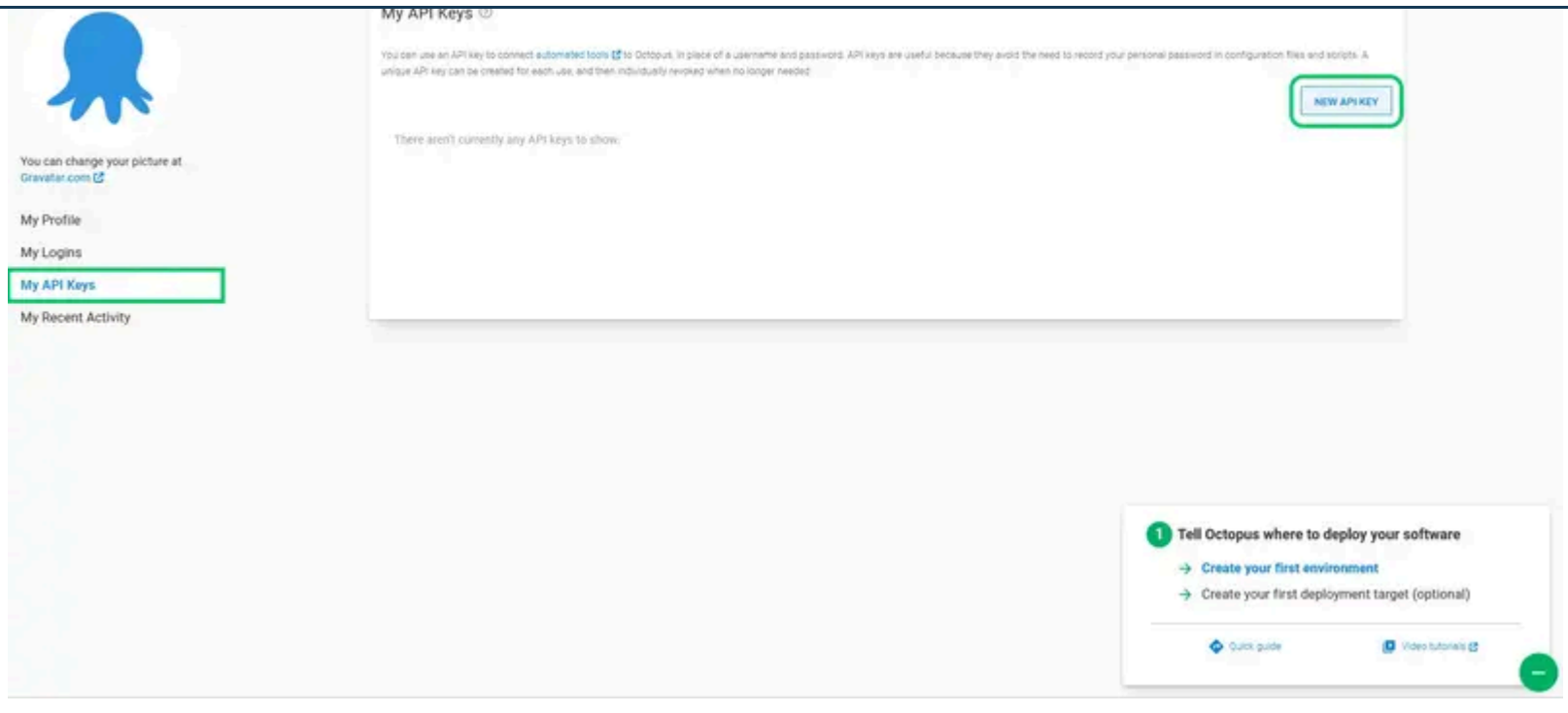


From the Octopus Deploy web portal, sign in, and view your profile:

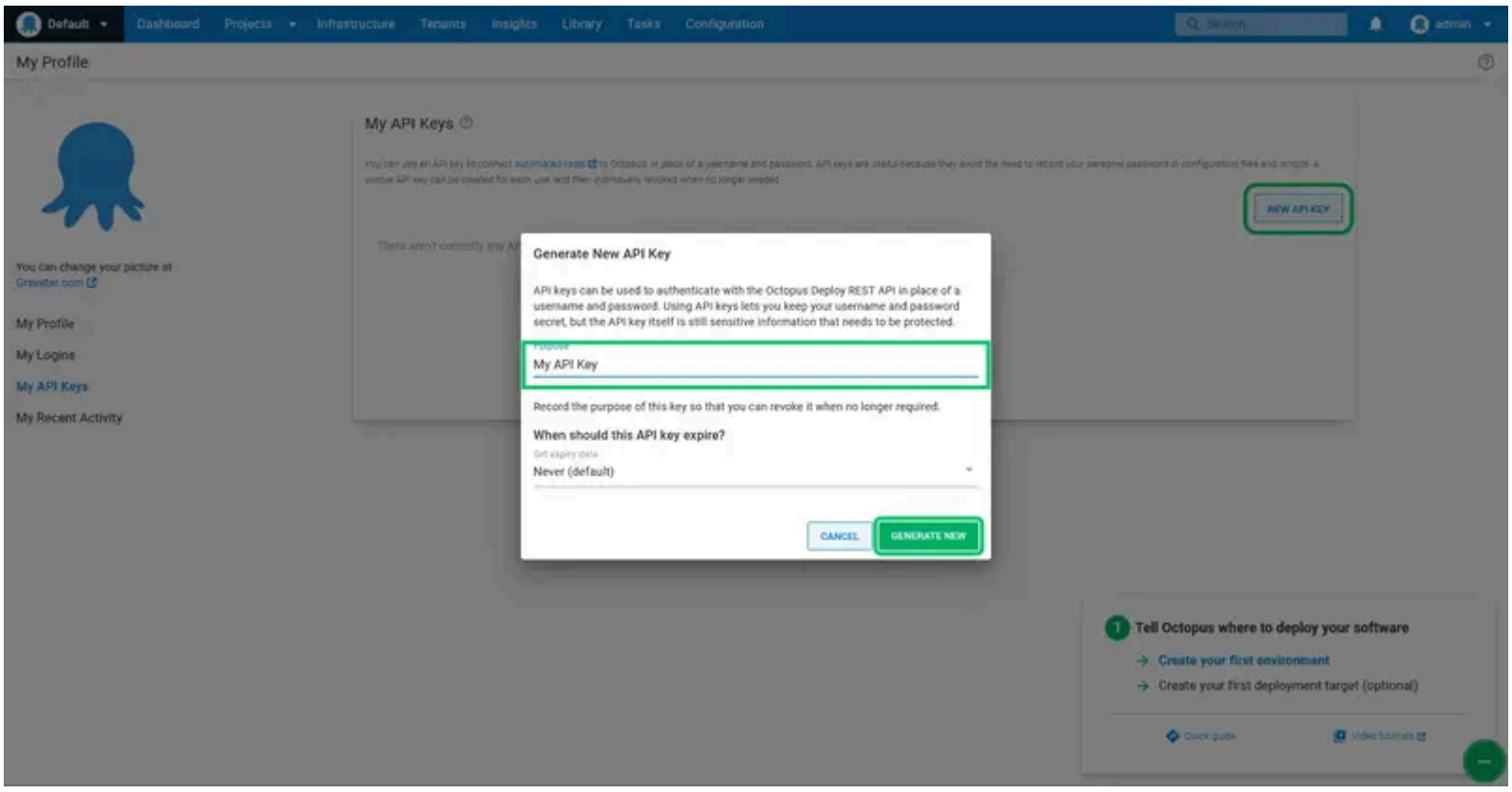


Go to the **API keys tab**. This lists any previous API keys that you have created. Click on **New API key**:

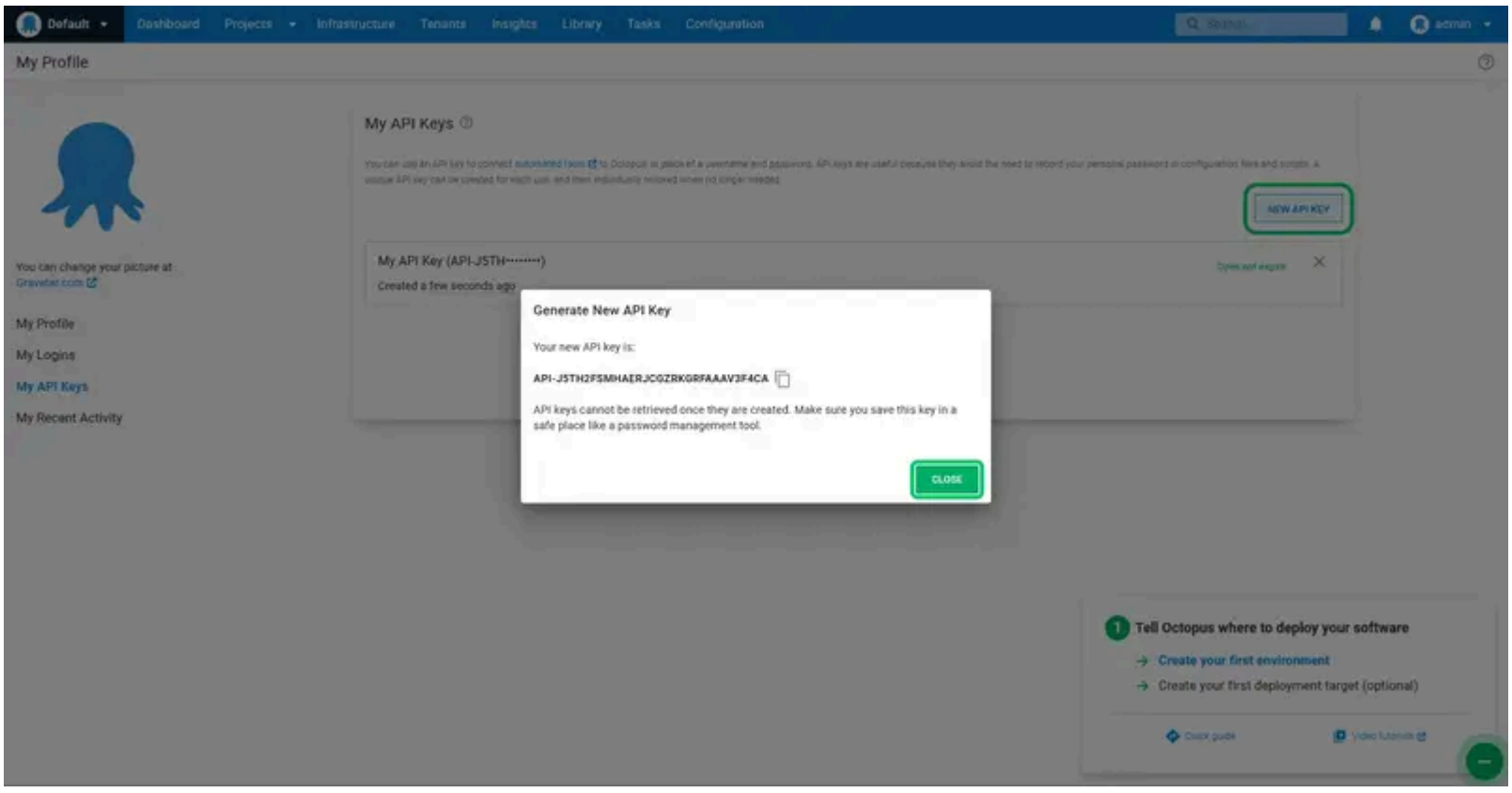
Octopus Deploy



Give the API key a name so that you remember what the key is for, and click **Generate New**:



Copy the new API key to your clipboard:



Octopus Deploy

Continuous integration

GitHub Actions supports Node.js, Python, Java, Ruby, PHP, Go, Rust, .NET, and more. Build, test, and deploy applications in your language of choice. In this tutorial, we rely on GitHub Actions to do the following:

- Clone the code from Git.
- Resolve and install any dependencies from Maven.
- Run unit tests.
- Package the application by bundling all the files it needs to run into a WAR file.
- Push the package to the Octopus (built-in) package repository.

At a high level, GitHub Actions workflows are YAML documents committed to a GitHub repsository that define the **runner** that the steps are executed on, the **triggers** that execute the workflow, and the steps to be run.

GitHub Actions workflows have access to **encrypted secrets**. Two secrets are defined called:

- **OCTOPUS_SERVER_URL**, which defines the URL of the Octopus server (for example, https://myserver.octopus.app for hosted Octopus instances)
- **OCTOPUS_API_TOKEN**, which defines the API key used to access the Octopus instance.

The YAML document below is saved to a file called **.github/workflows/build.yaml**. It is configured to be triggered on a git push and defines steps to build, test, package, and publish the sample application:

Octopus Deploy

```
    steps:
    - uses: actions/checkout@v3
      with:
        fetch-depth: '0'
    - name: Install GitVersion
      uses: gittools/actions/gitversion/setup@v0.9.14
      with:
        versionSpec: 5.x
    - id: determine_version
      name: Determine Version
      uses: gittools/actions/gitversion/execute@v0.9.14
      with:
        additionalArguments: /overrideconfig mode=Mainline
    - name: Install Octopus Deploy CLI
      uses: OctopusDeploy/install-octopus-cli-action@v1
      with:
        version: latest
    - name: Set up JDK 1.17
      uses: actions/setup-java@v2
      with:
        java-version: '17'
        distribution: adopt
    - name: Set Version
      run: ./mvnw --batch-mode versions:set -DnewVersion=${{ steps.determine_version.outputs.semVer }}
      shell: bash
    - name: Test
      run: ./mvnw --batch-mode test
      shell: bash
    - if: always()
      name: Report
      uses: dorny/test-reporter@v1
      with:
        name: Maven Tests
        path: target/surefire-reports/*.xml
        reporter: java-junit
        fail-on-error: 'false'
    - name: Package
      run: ./mvnw --batch-mode -DskipTests=true package
      shell: bash
    - id: get_artifact
      name: Get Artifact Path
      run: |-
        # Find the largest WAR or JAR, and assume that was what we intended to build.
        echo "::set-output name=artifact::$(find target -type f \( -iname \*.jar -o -iname \*.war \) -printf "%p\n" |
sort -n | head -1)"
      shell: bash
    - id: get_artifact_name
      name: Get Artifact Name
      run: |-
        # Get the filename without a path
        path="${{ steps.get_artifact.outputs.artifact }}"
        echo "::set-output name=artifact::${path##*/}"
      shell: bash
    - id: get_octopus_artifact
      name: Create Octopus Artifact
      run: |-
        # Octopus expects artifacts to have a specific file format
        file="${{ steps.get_artifact.outputs.artifact }}"
        extension="${file##*.}"
        octofile="randomquotes.${{ steps.determine_version.outputs.semVer }}.${extension}"
        cp ${file} ${octofile}
        echo "::set-output name=artifact::${octofile}"
        # The version used when creating a release is the package id, colon, and version
```

Octopus Deploy

```
      shell: bash
    - name: Push packages to Octopus Deploy 🐙
      uses: OctopusDeploy/push-package-action@v2
      env:
        OCTOPUS_API_KEY: ${{ secrets.OCTOPUS_API_TOKEN }}
        OCTOPUS_HOST: ${{ secrets.OCTOPUS_SERVER_URL }}
      with:
        overwrite_mode: OverwriteExisting
        packages: ${{ steps.get_octopus_artifact.outputs.artifact }}
name: Java Maven Build
'on':
  workflow_dispatch: {}
  push: {}
```

There is a lot of work being performed by this workflow, so let's break it down.

The job called **build** is run on an Ubuntu runner:

```
jobs:
  build:
    runs-on: ubuntu-latest
```

The first step checks out the source code from the git repository. Setting the **fetch-depth** option to **0** means all the history for all branches and tags is fetched. This is required by the GitVersion action used in later steps:

```
    steps:
    - uses: actions/checkout@v3
      with:
        fetch-depth: '0'
```

GitVersion builds meaningful version strings based on the commits to git.

This step installs GitVersion:

```
    - name: Install GitVersion
      uses: gittools/actions/gitversion/setup@v0.9.14
      with:
        versionSpec: 5.x
```

This step calls GitVersion which outputs many different variables representing different parts of a SemVer version string generated from the commits to the git repository. Later steps use these output values as the version of the package generated by this workflow:

```
    - id: determine_version
      name: Determine Version
      uses: gittools/actions/gitversion/execute@v0.9.14
      with:
        additionalArguments: /overrideconfig mode=Mainline
```

This step installs the Octopus CLI, which is used by the subsequent Octopus GitHub Actions:

Octopus Deploy

```
        version: latest
```

This step sets up Java 17:

```
    - name: Set up JDK 1.17
      uses: actions/setup-java@v2
      with:
        java-version: '17'
        distribution: adopt
```

This step sets the project version to the SemVer version determined by GitVersion:

```
    - name: Set Version
      run: ./mvnw --batch-mode versions:set -DnewVersion=${{ steps.determine_version.outputs.semVer }}
      shell: bash
```

This step runs the unit tests:

```
    - name: Test
      run: ./mvnw --batch-mode test
      shell: bash
```

This step publishes the results of the test against the workflow run. The **if** parameter is set to **always()** to ensure that the results are published even if the tests fail:

```
    - if: always()
      name: Report
      uses: dorny/test-reporter@v1
      with:
        name: Maven Tests
        path: target/surefire-reports/*.xml
        reporter: java-junit
        fail-on-error: 'false'
```

This step packages the application, skipping tests, as those have already been run:

```
    - name: Package
      run: ./mvnw --batch-mode -DskipTests=true package
      shell: bash
```

This step finds the largest war or jar file in the **target** directory. We assume the largest file is the one we wish to deploy:

```
    - id: get_artifact
      name: Get Artifact Path
      run: |-
        # Find the largest WAR or JAR, and assume that was what we intended to build.
        echo "::set-output name=artifact::$(find target -type f \( -iname \*.jar -o -iname \*.war \) -printf "%p\n" |
sort -n | head -1)"
      shell: bash
```

```
    name: Get Artifact Name
    run: |-
      # Get the filename without a path
      path="${{ steps.get_artifact.outputs.artifact }}"
      echo "::set-output name=artifact::${path##*/}"
    shell: bash
```

Java packages are saved with filenames like **randomquotes-1.0.0.war**, with a dash between the package name and the version. Octopus requires file names like **randomquotes.1.0.0.war**, with a period between the package name and version.

This step builds two output variables. The first, **artifact**, is the Java package filename reformatted to conform to the format expected by Octopus. The second, **octoversion**, is the package ID and version in a format expected by the create release step (which will be defined later):

```
- id: get_octopus_artifact
  name: Create Octopus Artifact
  run: |-
    # Octopus expects artifacts to have a specific file format
    file="${{ steps.get_artifact.outputs.artifact }}"
    extension="${file##*.}"
    octofile="randomquotes.${{ steps.determine_version.outputs.semVer }}.${extension}"
    cp ${file} ${octofile}
    echo "::set-output name=artifact::${octofile}"
    # The version used when creating a release is the package id, colon, and version
    octoversion="randomquotes:${{ steps.determine_version.outputs.semVer }}"
    echo "::set-output name=octoversion::${octoversion}"
  shell: bash
```

The package files are then pushed to Octopus:

```
- name: Push packages to Octopus Deploy 🐙
  uses: OctopusDeploy/push-package-action@v2
  env:
    OCTOPUS_API_KEY: ${{ secrets.OCTOPUS_API_TOKEN }}
    OCTOPUS_HOST: ${{ secrets.OCTOPUS_SERVER_URL }}
  with:
    overwrite_mode: OverwriteExisting
    packages: ${{ steps.get_octopus_artifact.outputs.artifact }}
```

The workflow is triggered on each push to the git repository. The workflow can also be triggered manually with the **workflow_dispatch** event:

```
name: Java Maven Build
'on':
  workflow_dispatch: {}
  push: {}
```

Commit the file and push it to the GitHub repository with the command:

```
git add .; git commit -m "Added workflow file"; git push
```

GitHub Actions then runs the workflow.

Octopus Deploy

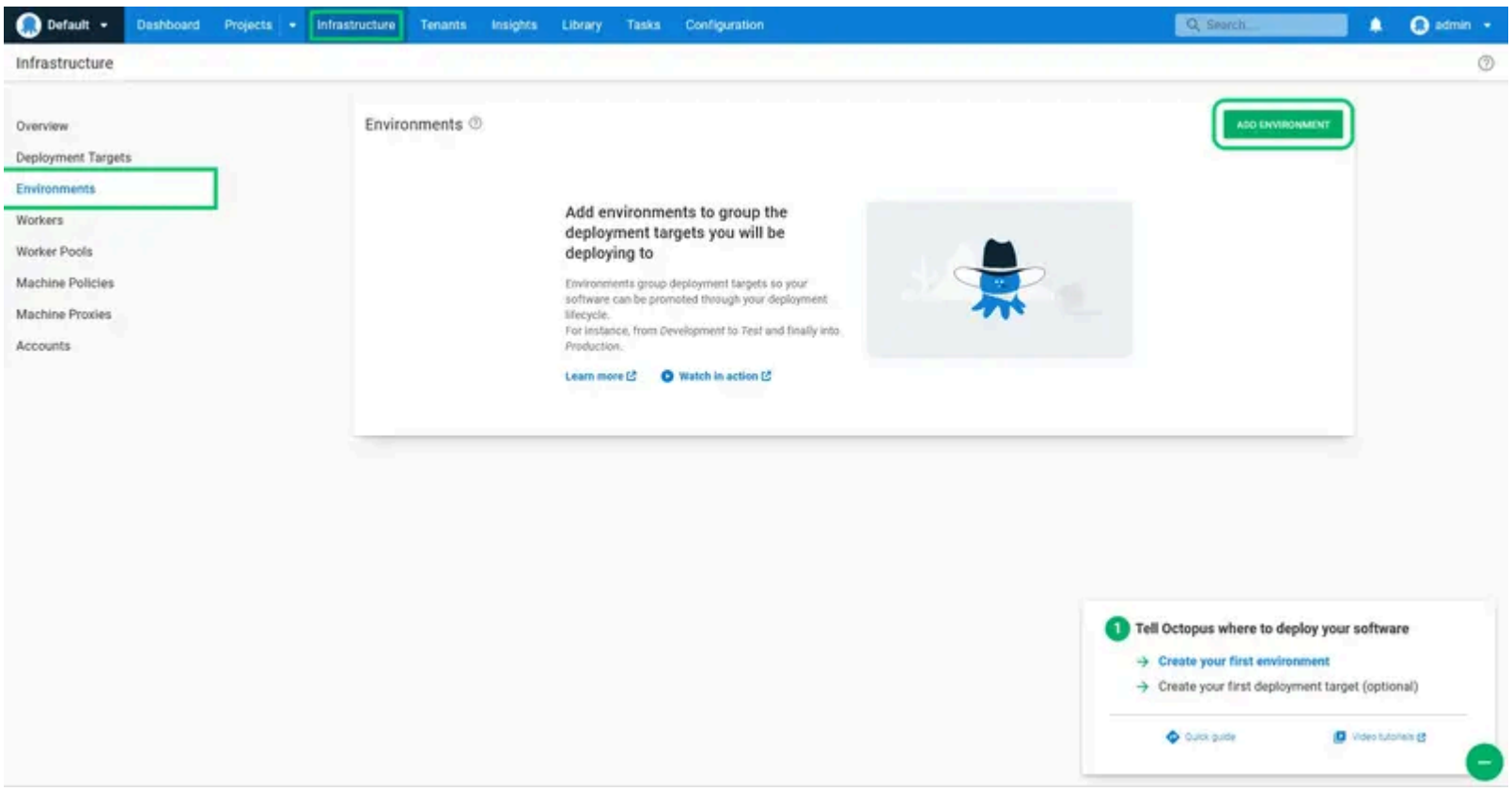
Deploying with Octopus Deploy

Now that GitHub Actions has successfully built the application, we need to configure Octopus to deploy it into our environments.

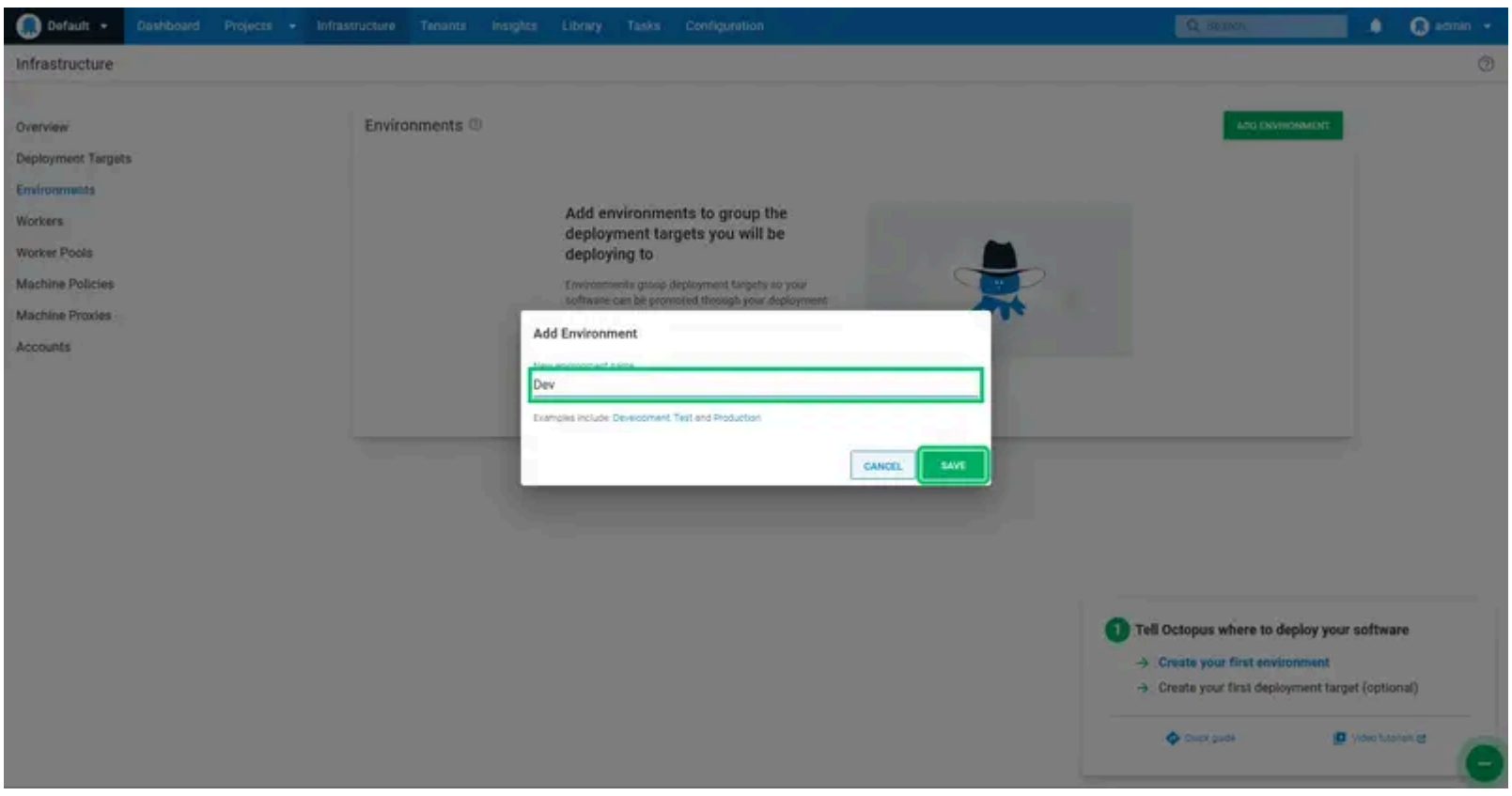
Create the environments

Environments represent the stages that a deployment must move through as part of the deployment pipeline. We'll create three environments: **Dev**, **Test**, and **Prod**.

Log into Octopus, and click the **Infrastructure** link, then the **Environments** link, and click **ADD ENVIRONMENT**:

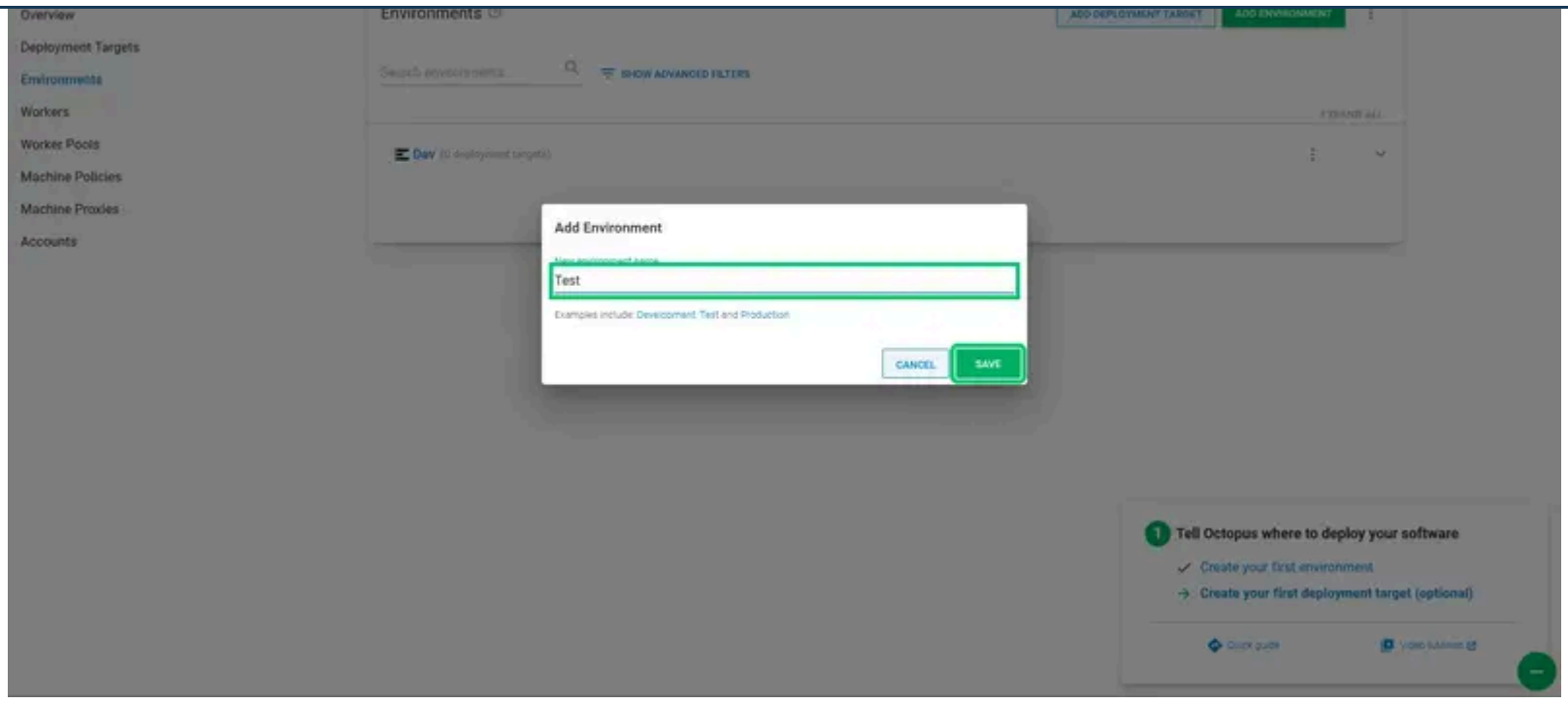


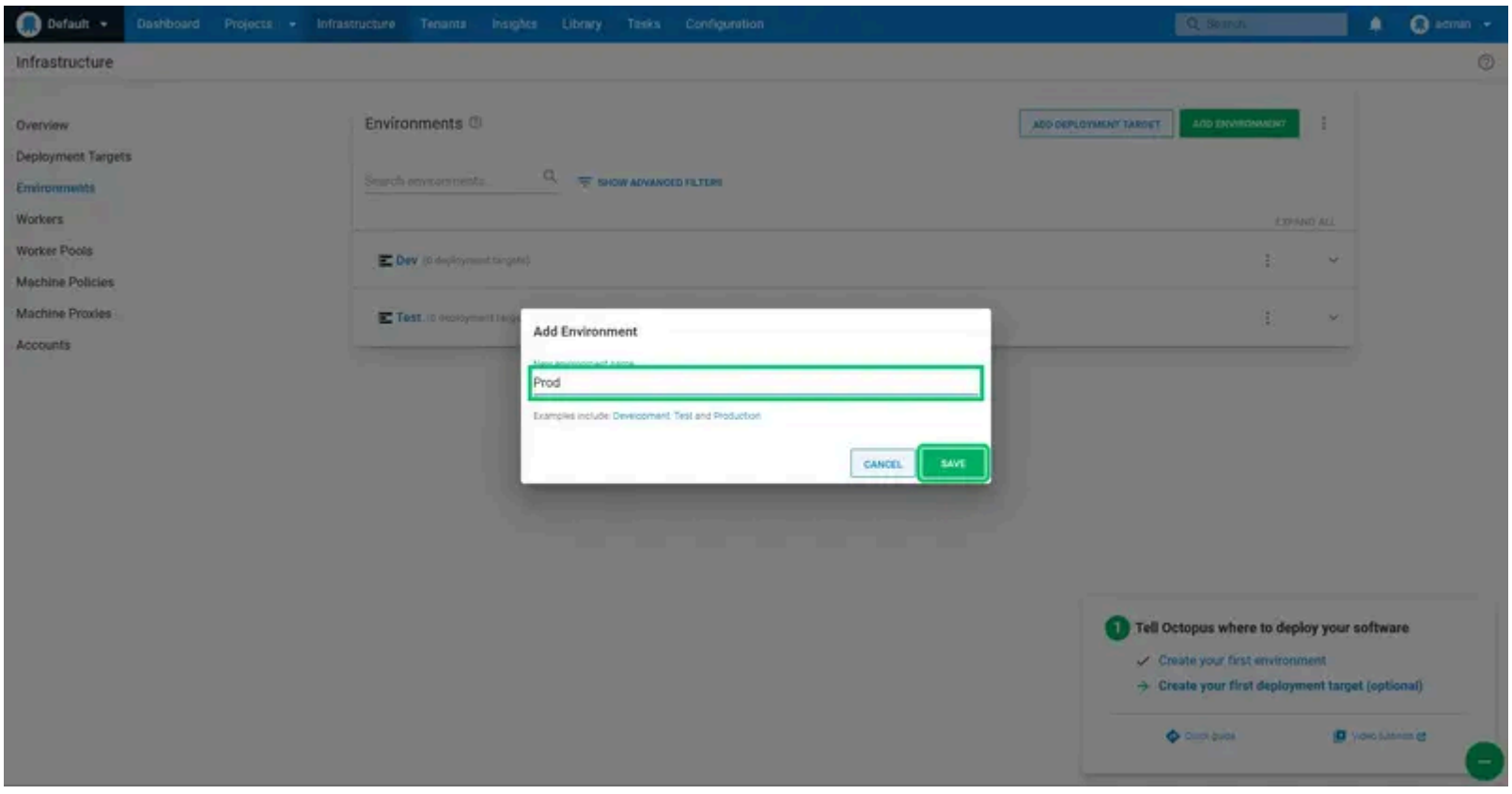
Enter **Dev** as the New environment name, and click **SAVE**:



Repeat the process for the **Test** and **Prod** environments:

Octopus Deploy



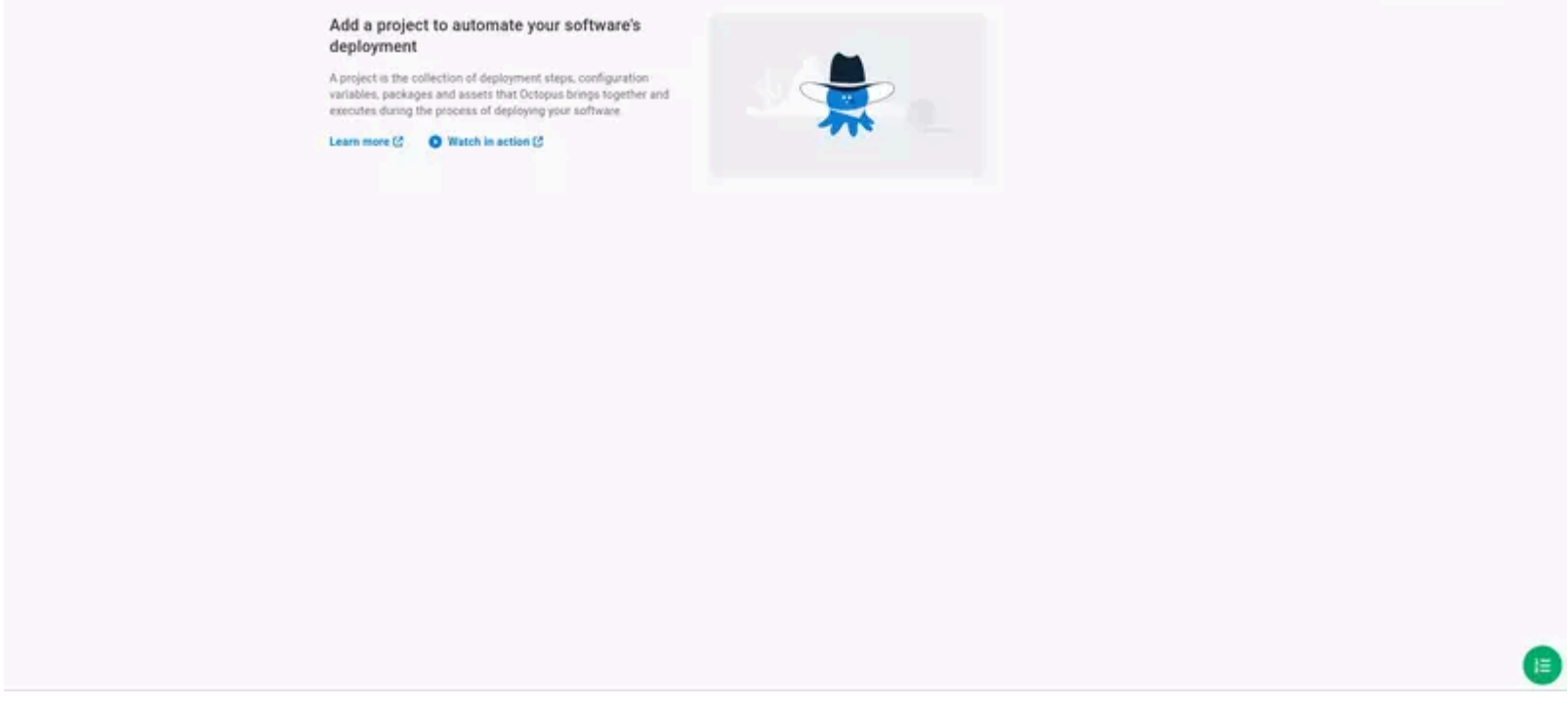


Create the Octopus deployment project

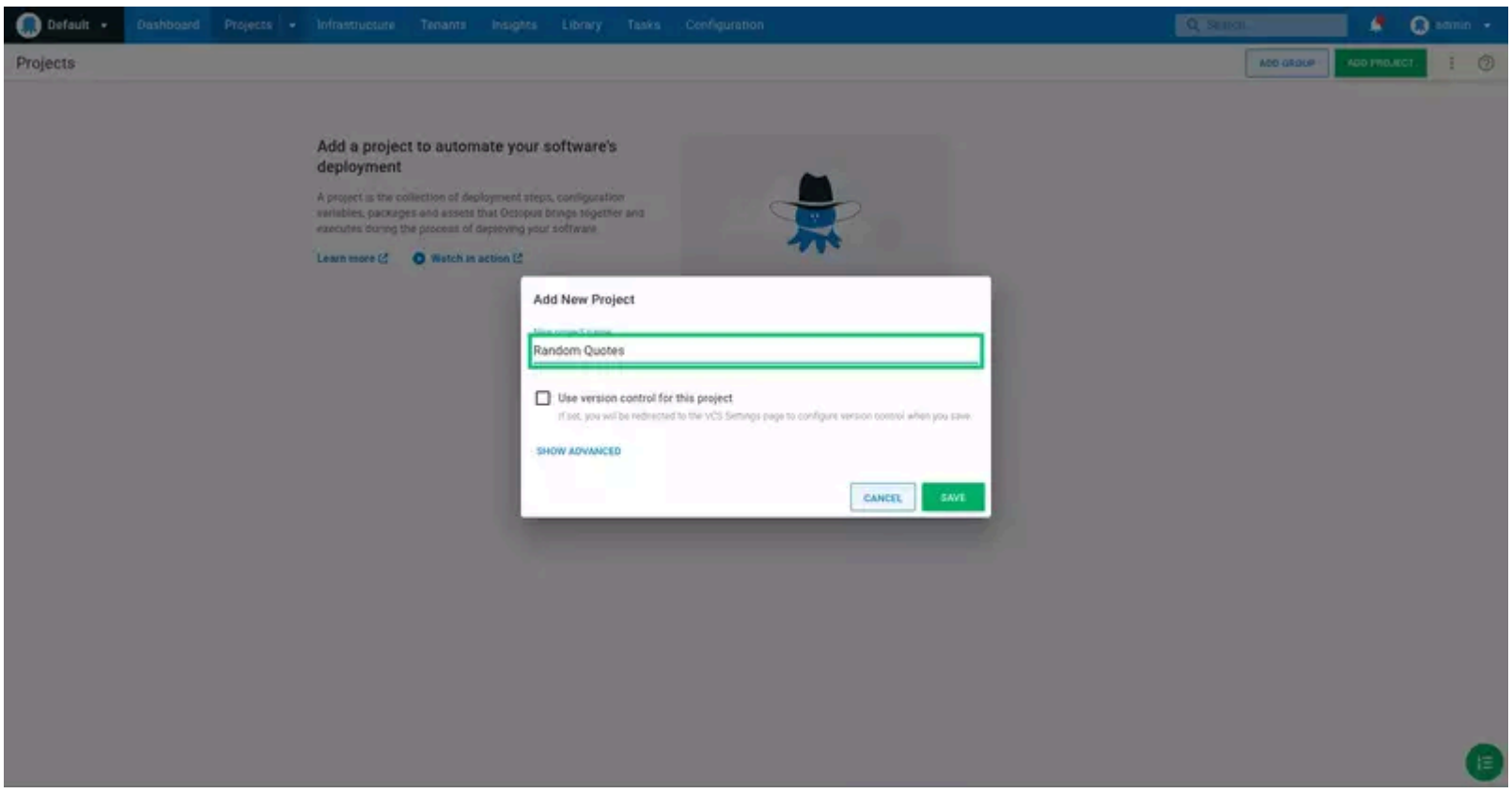
With the environments defined and a target created, we now need to create a deployment project in Octopus.

Log into Octopus, click the **Projects** link, and click **ADD PROJECT**:

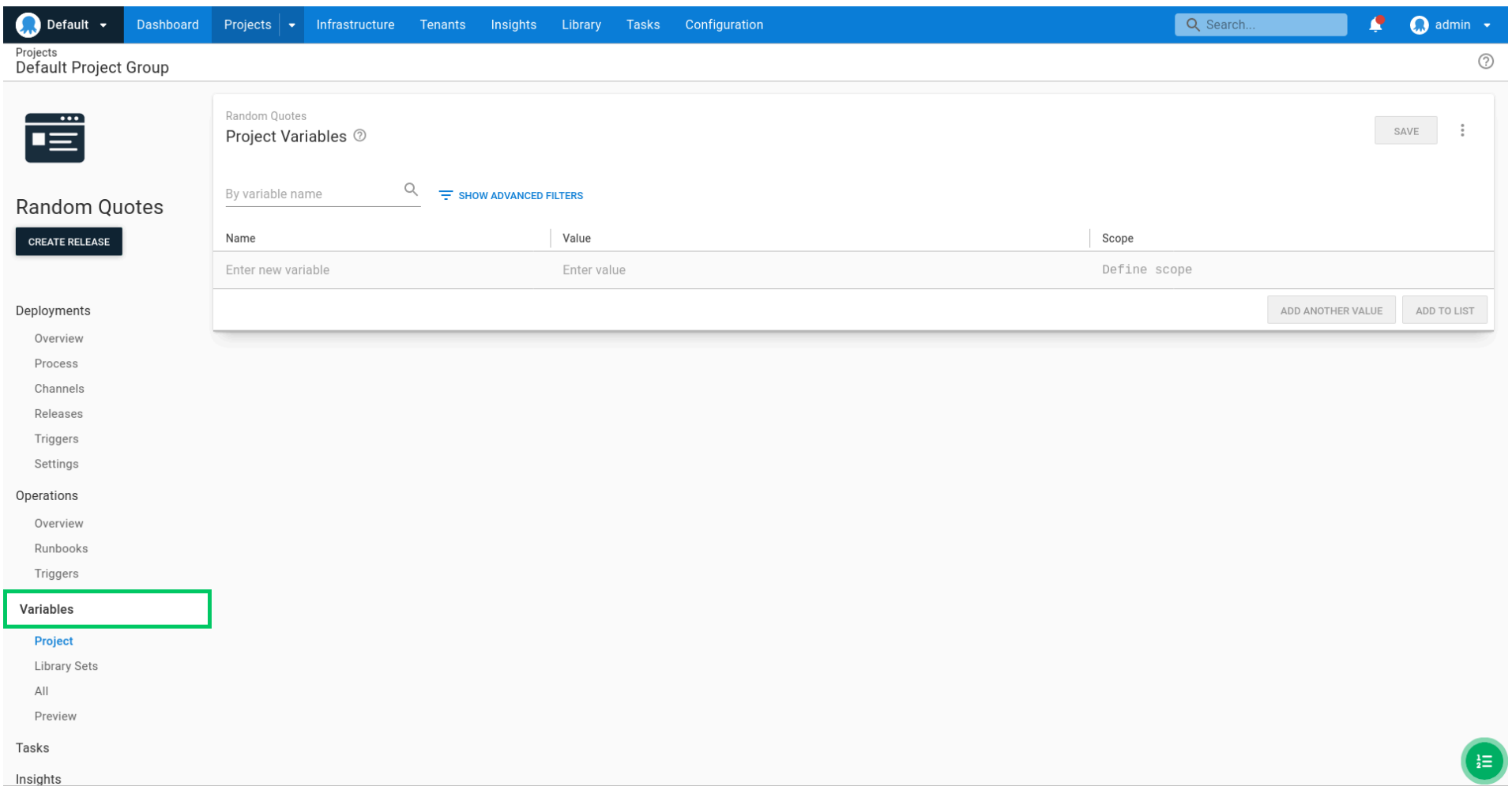
Octopus Deploy



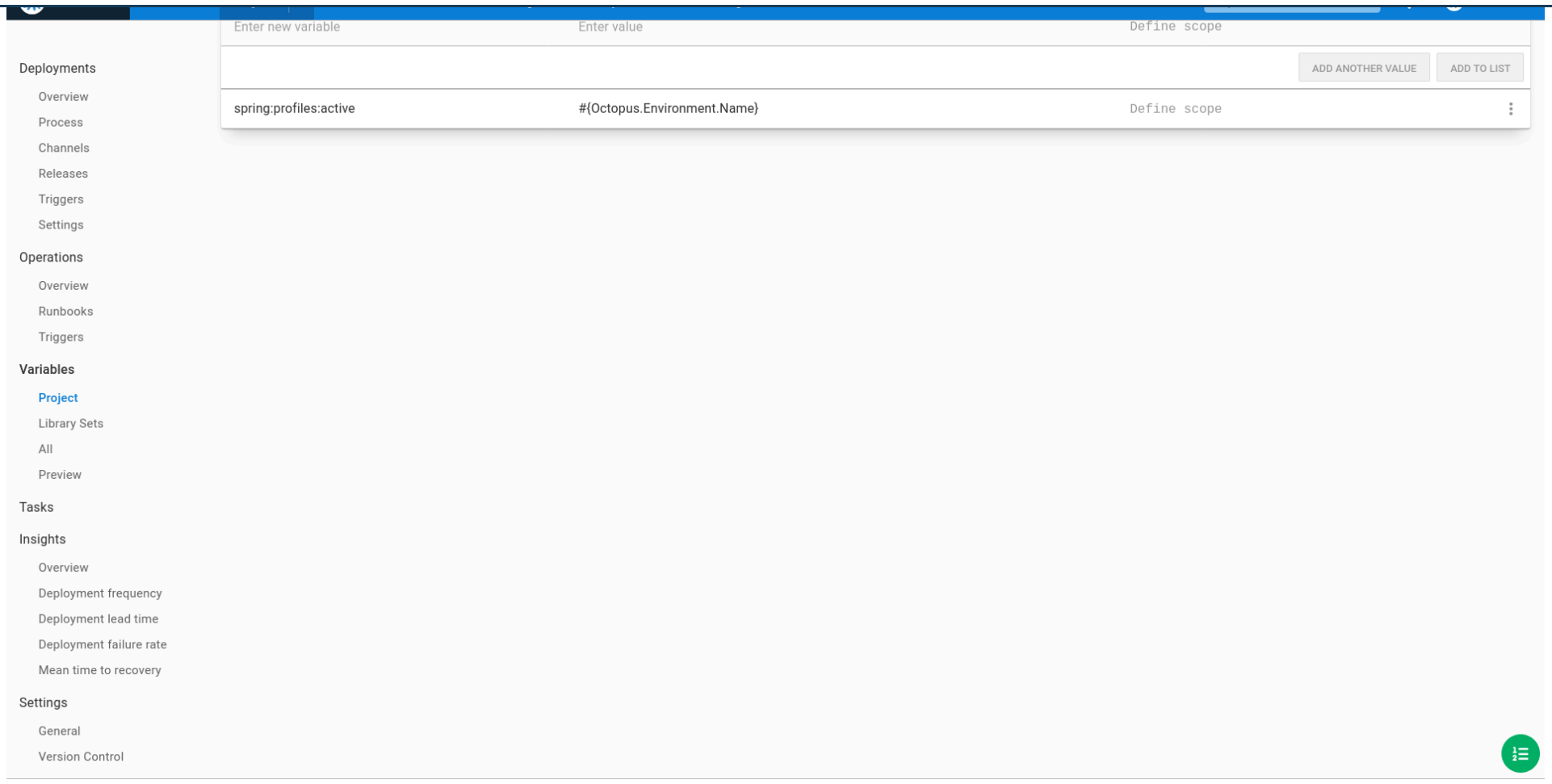
Enter **Random Quotes** for the project name, and click **SAVE**:



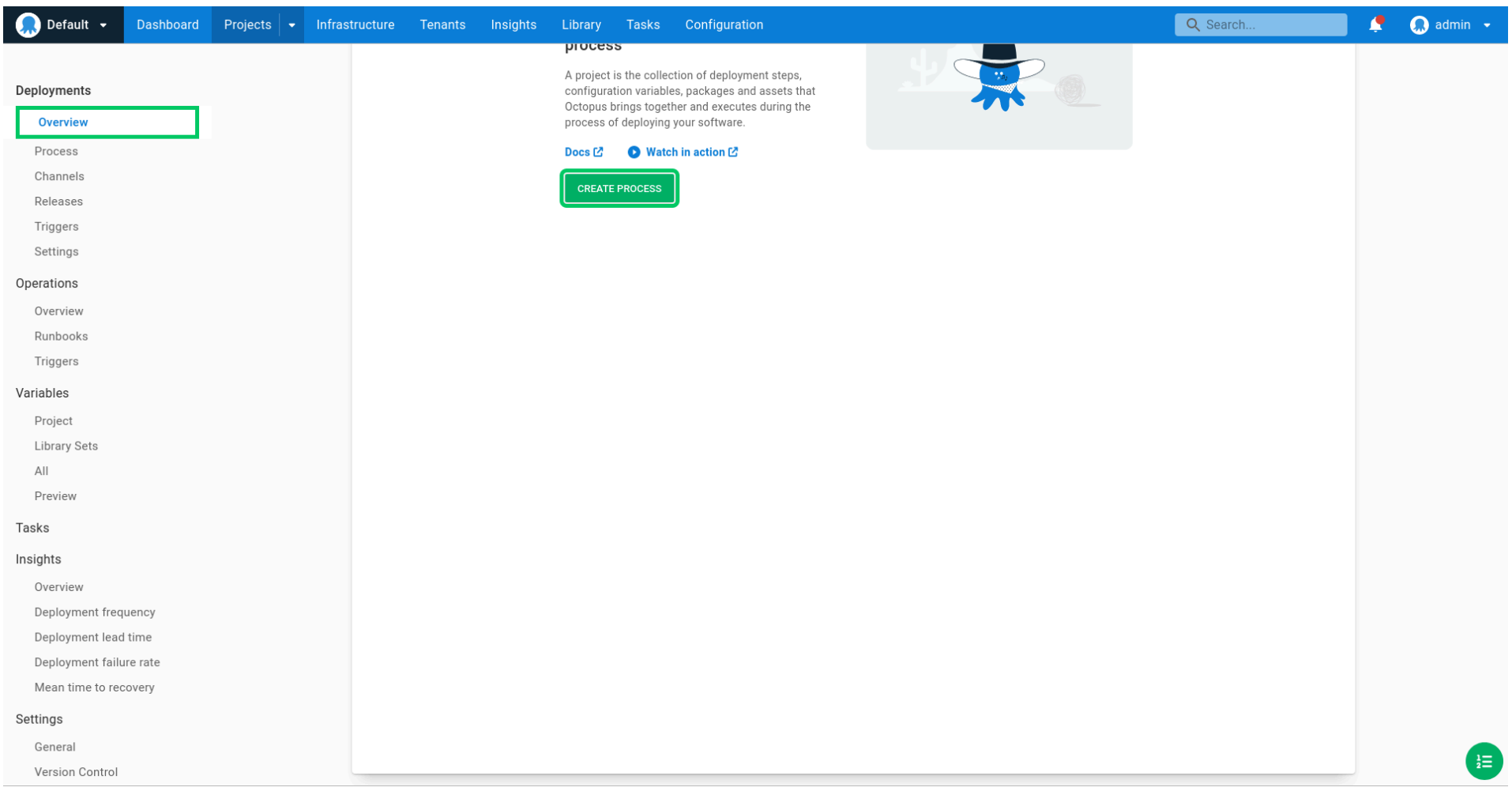
We start by defining a variable that will be injected into the **application.yml**. Click the **Variables** link:



Octopus Deploy

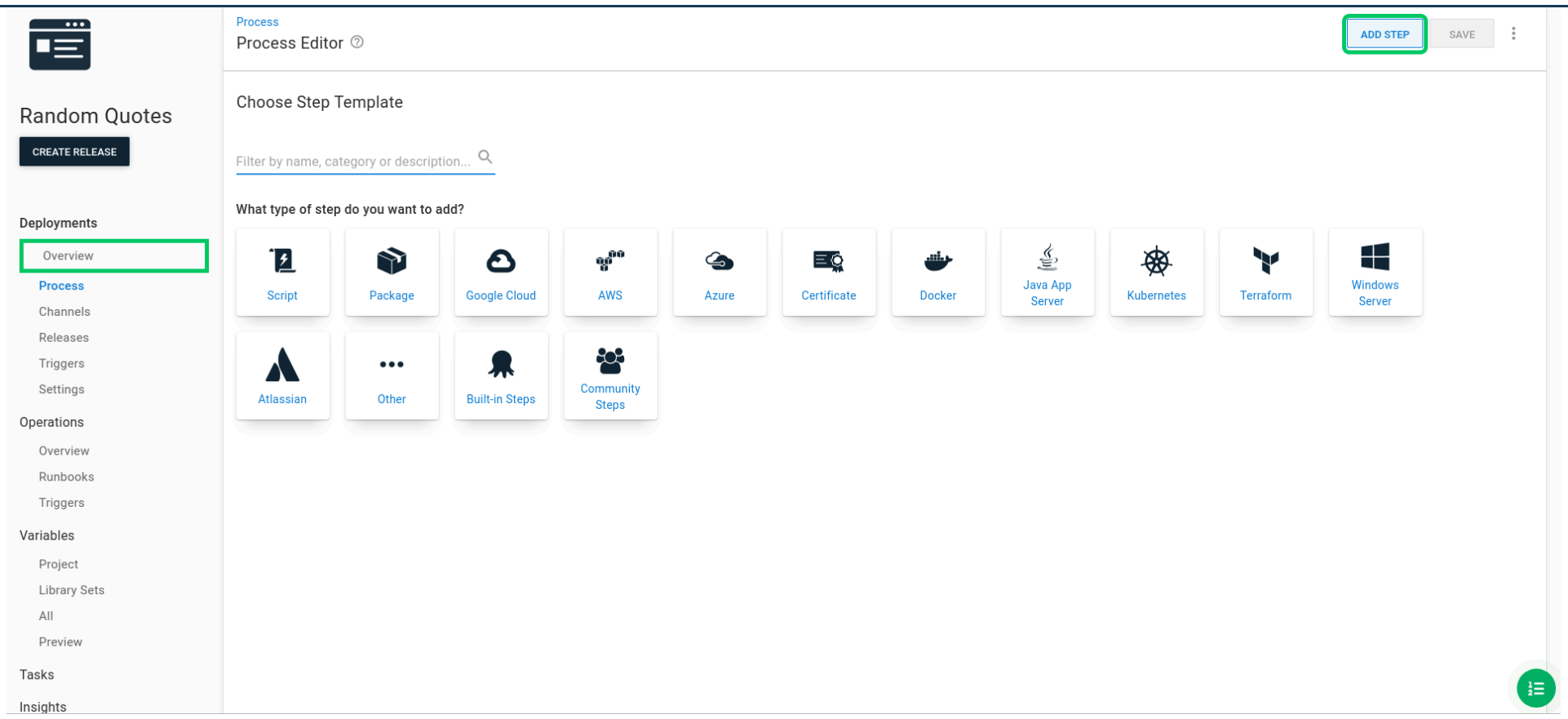


We will now define the deployment process. Click the **Deployments** link, click **Overview**, and then click **DEFINE YOUR DEPLOYMENT PROCESS**:

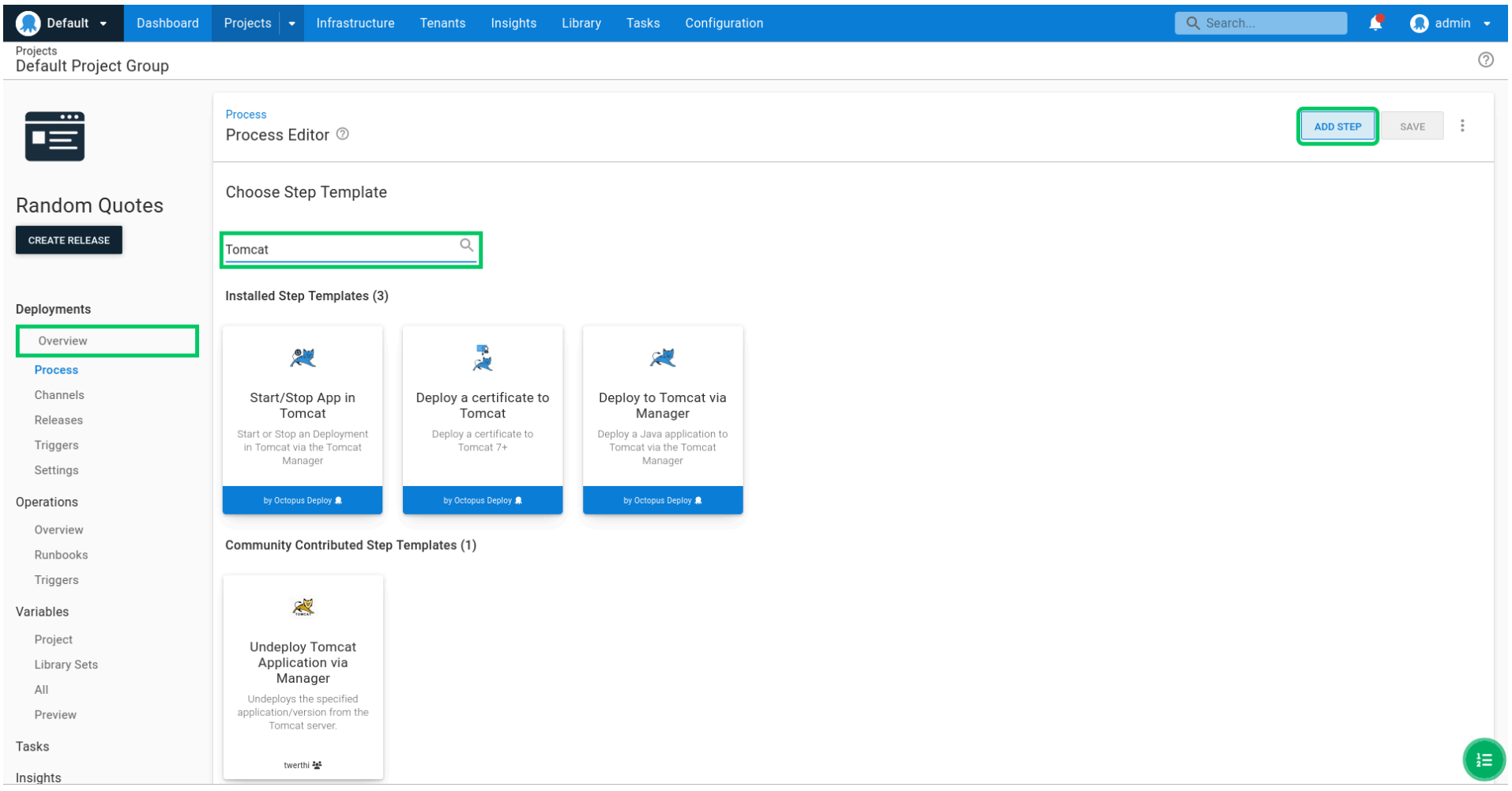


Click **ADD STEP**:

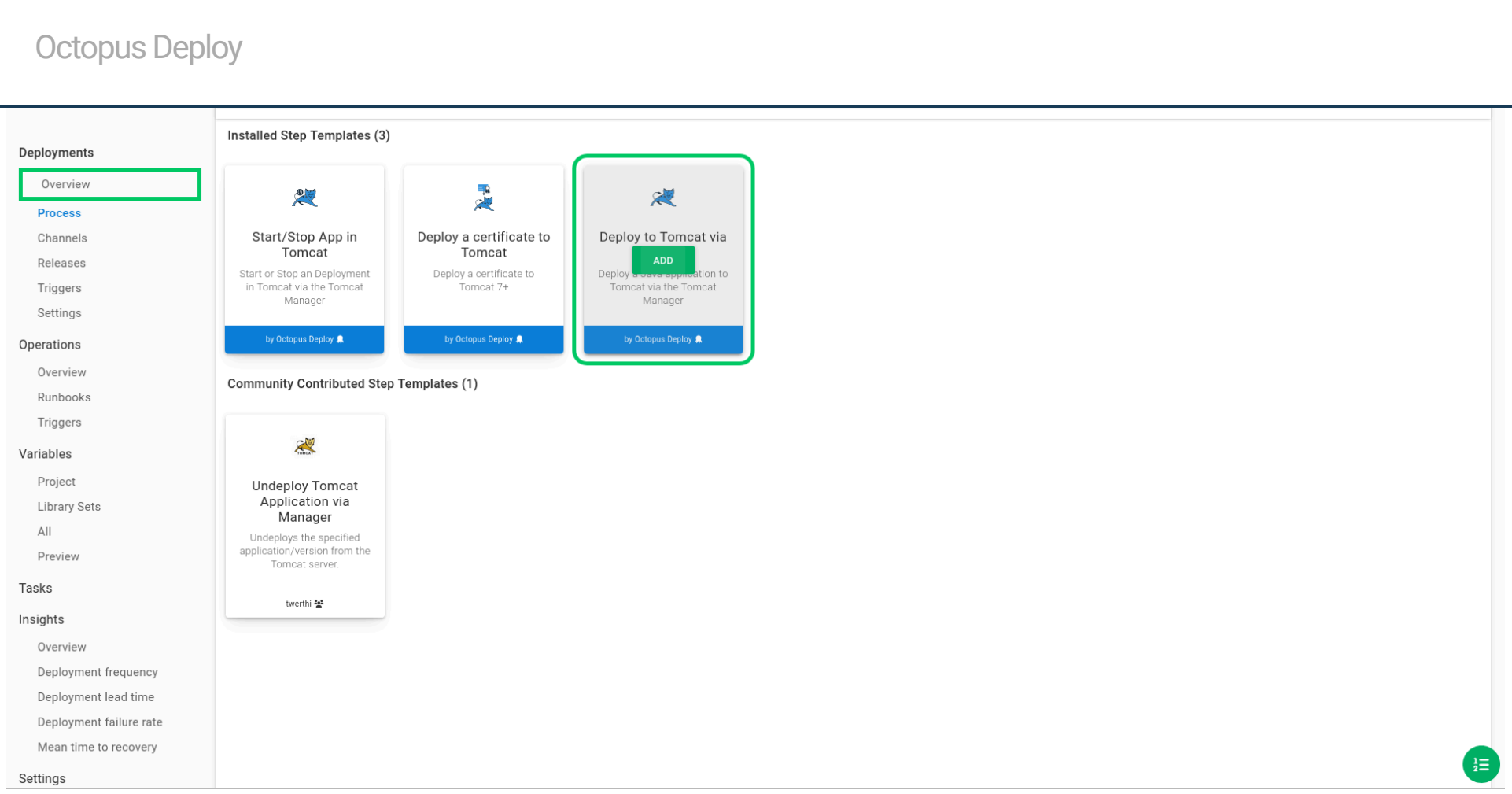
Octopus Deploy



Enter **Tomcat** into the search box:

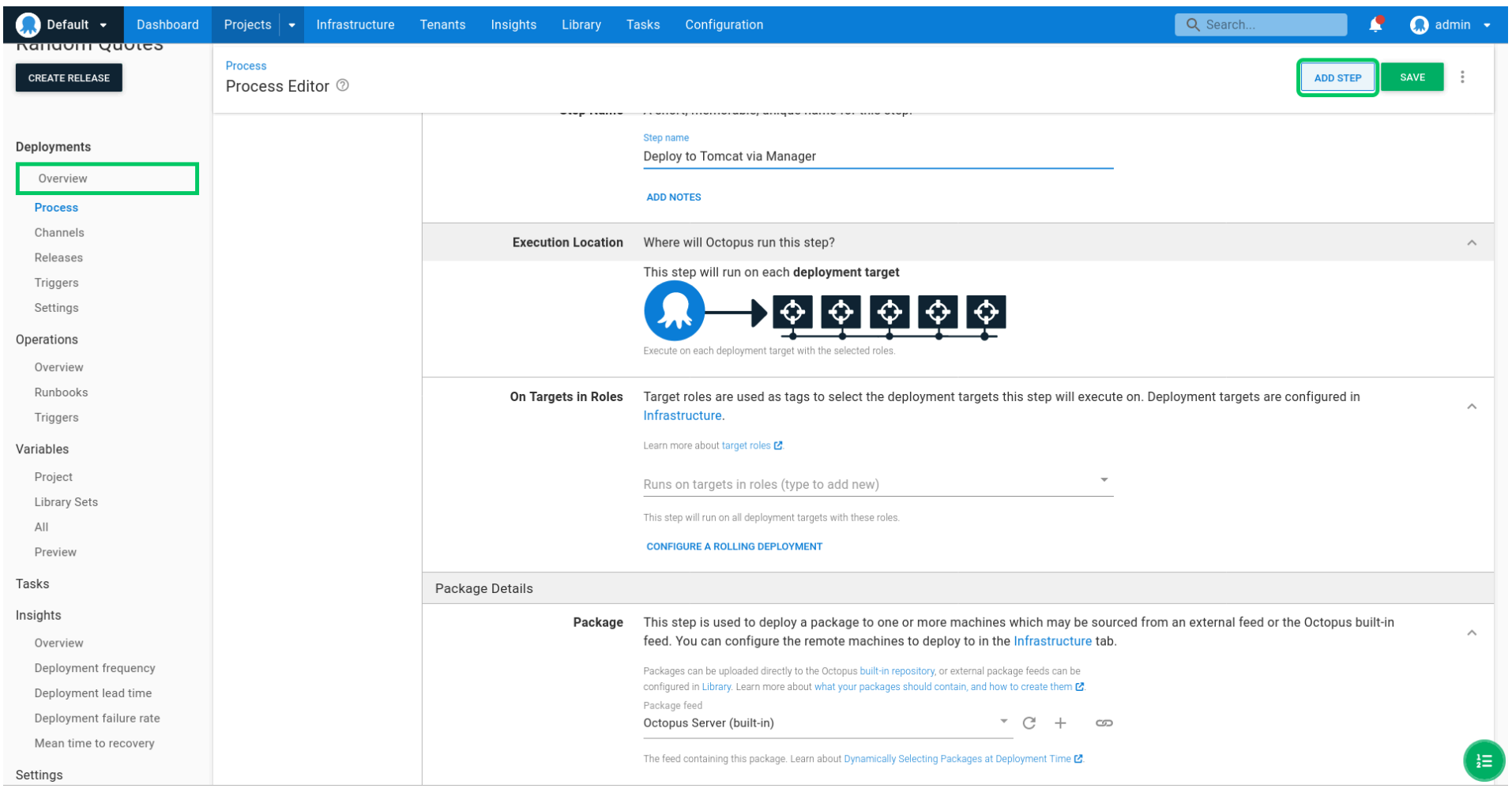


Click **ADD** on the **Deploy to Tomcat via Manager** tile:



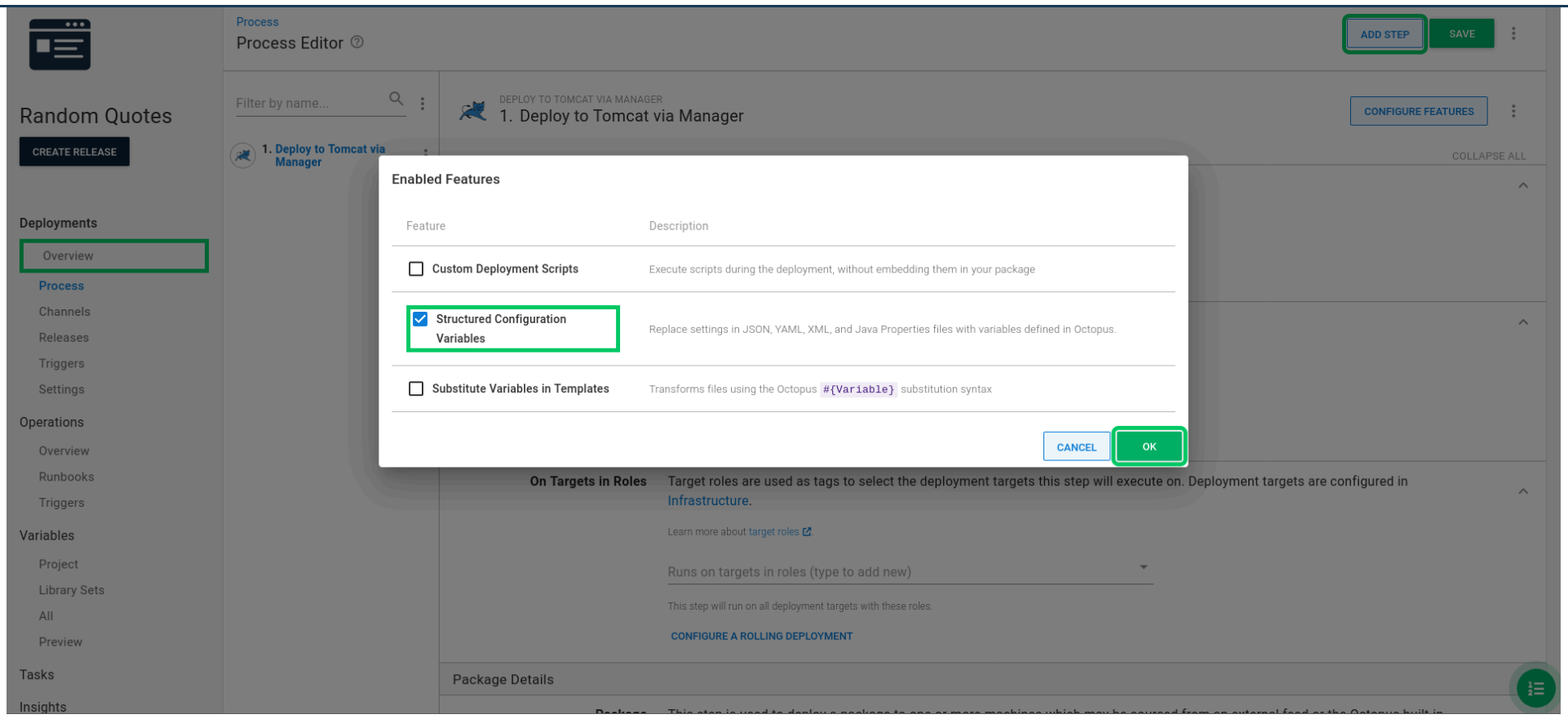
We want to take advantage of a feature in Octopus that will replace values in the application.yml file, which will set the Spring profile to the name of the environment that is being deployed to. To do this, we enable a feature on the deployment.

Click **CONFIGURATION FEATURES**:

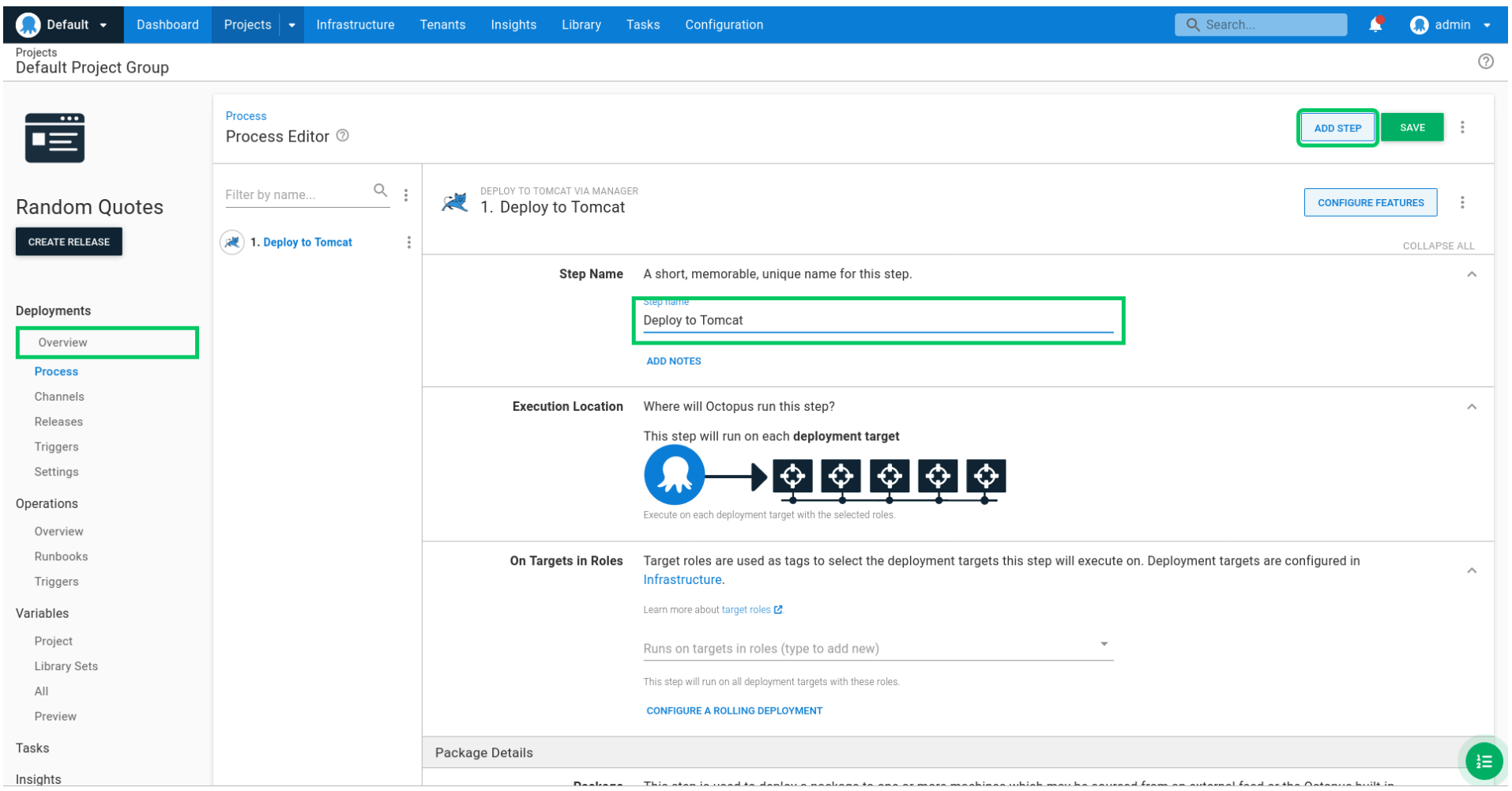


Enable the **Structured Configuration Variables** option, and click **OK**:

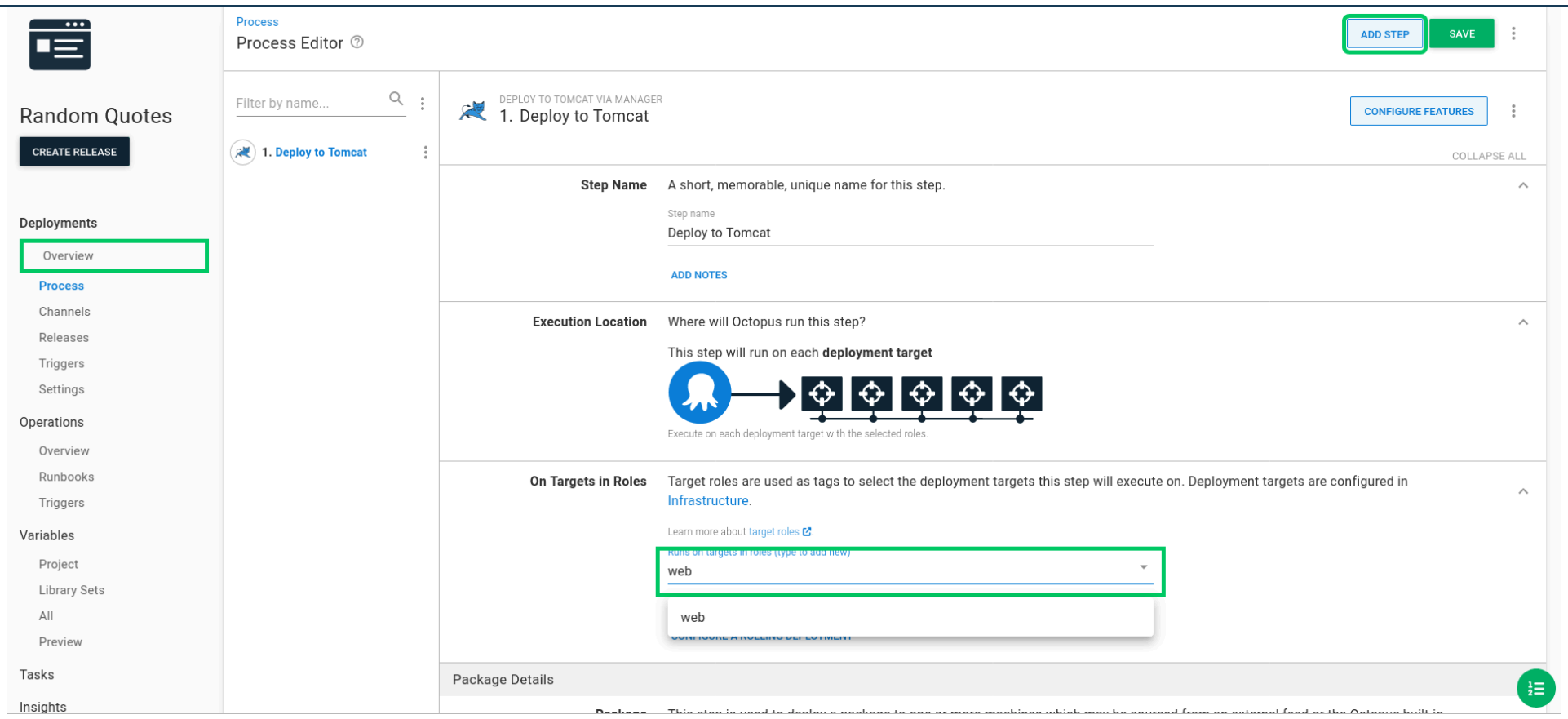
Octopus Deploy



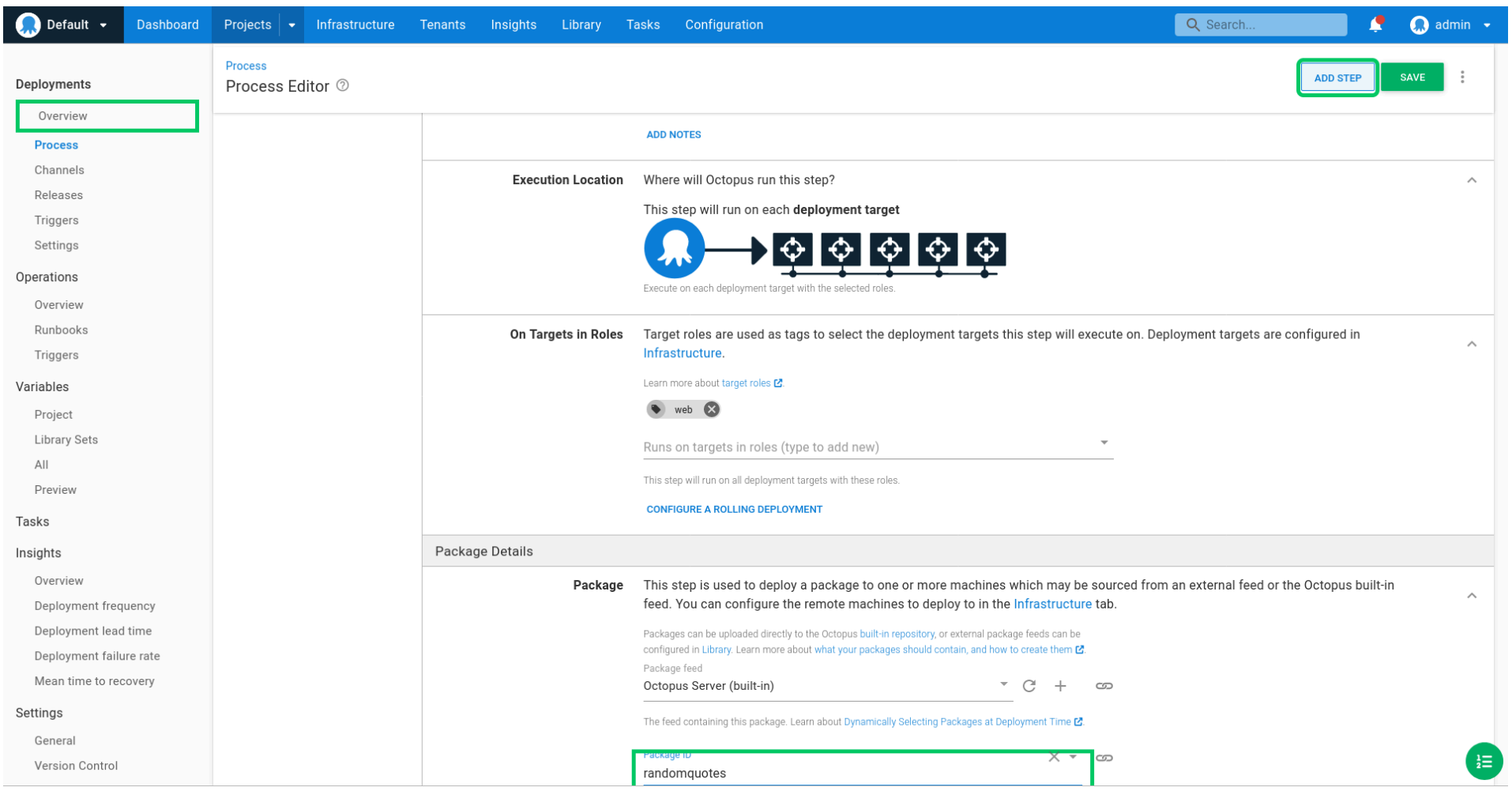
Enter **Deploy to Tomcat** as the **Step Name**:



Enter **web** as the **Role**:

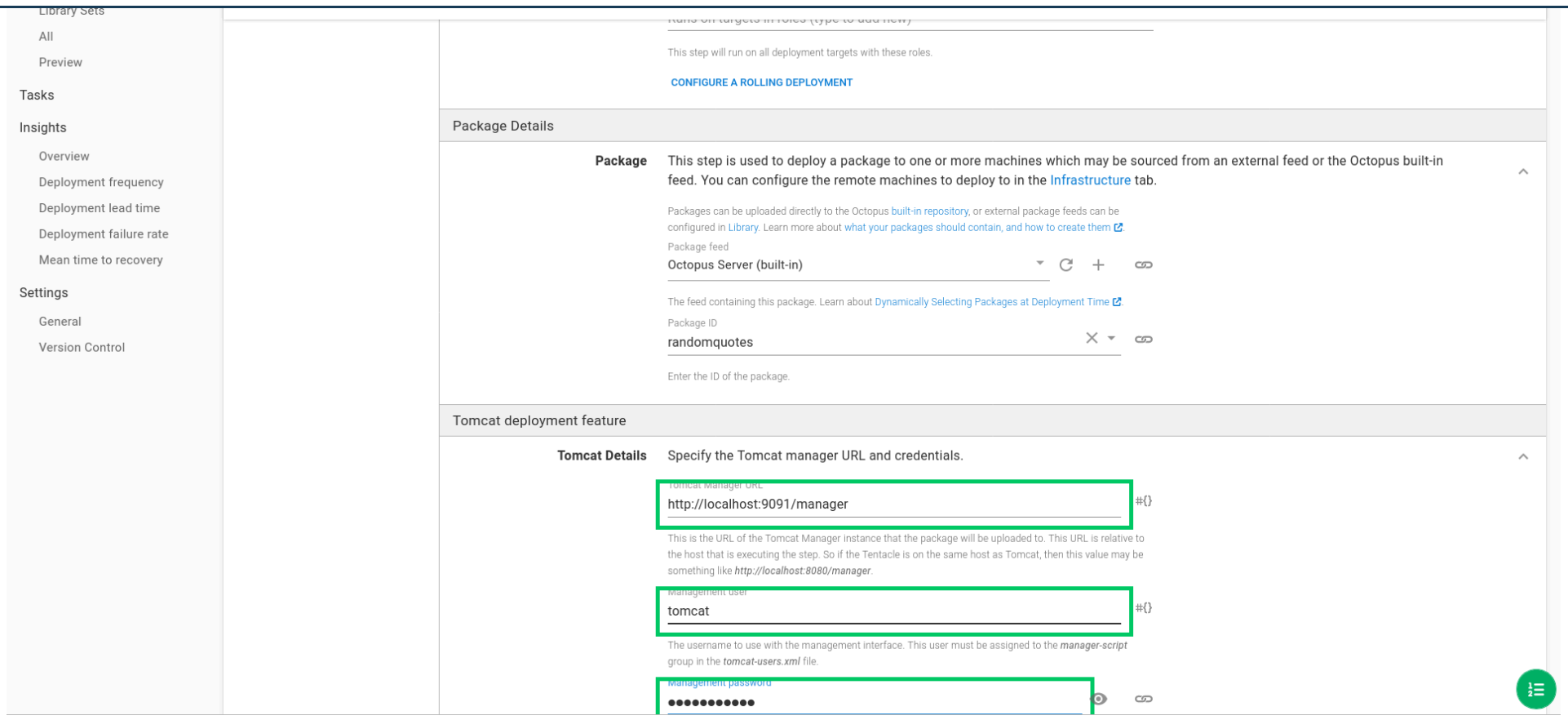


Select the **randomquotes** package:

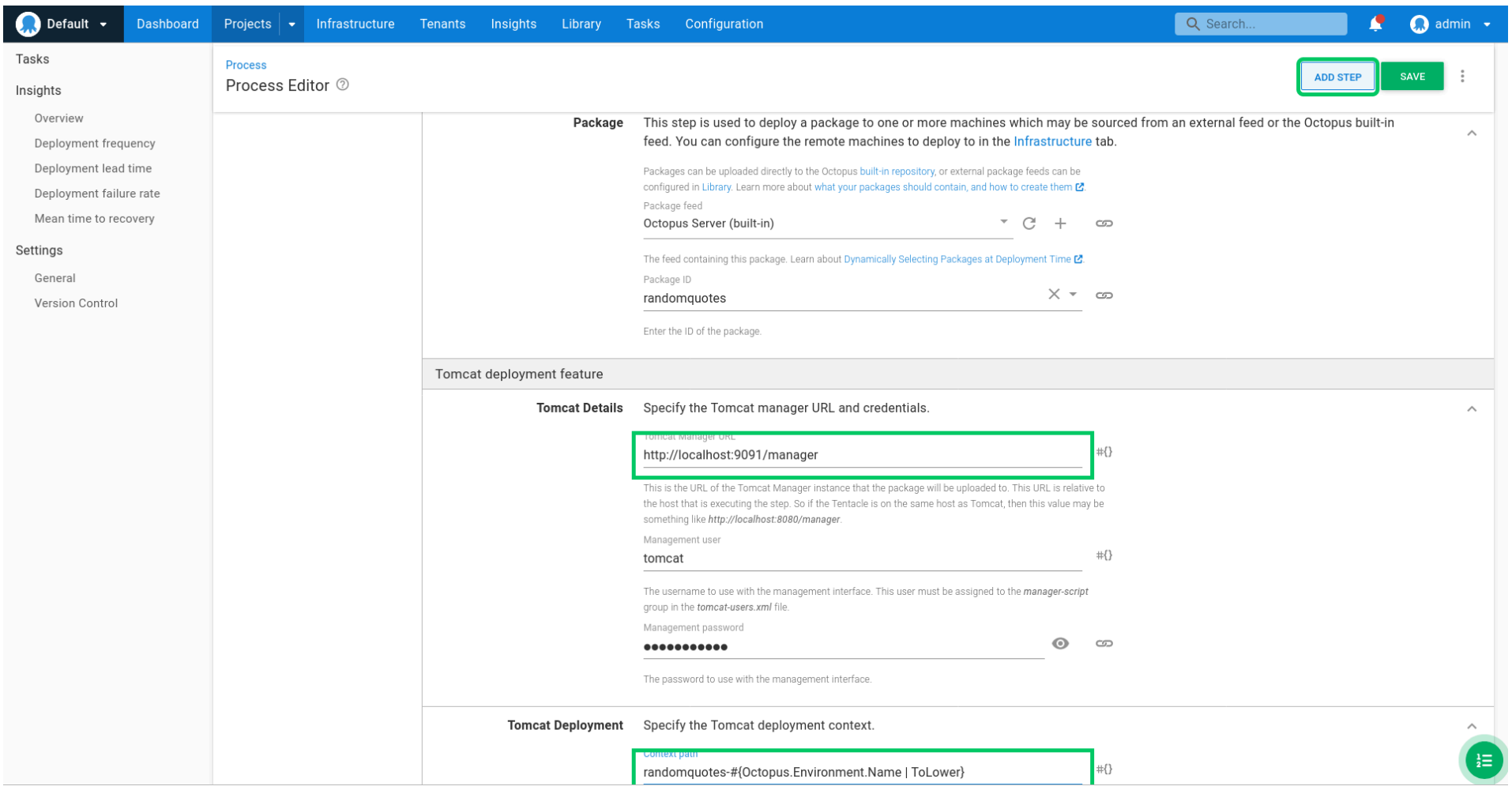


Enter the path to the Tomcat Manager application. The URL will be something like **http://localhost:9091/manager**.

Provide the credentials for the user account that has been assigned the **manager-script** role in the tomcat-users.xml file. See the Prerequisites for more details.



Enter **randomquotes-#{Octopus.Environment.Name | ToLower}** as the **Context path**. This ensures that the application is deployed to a unique path for every environment. This is how we can use a single Tomcat server to host multiple environments:



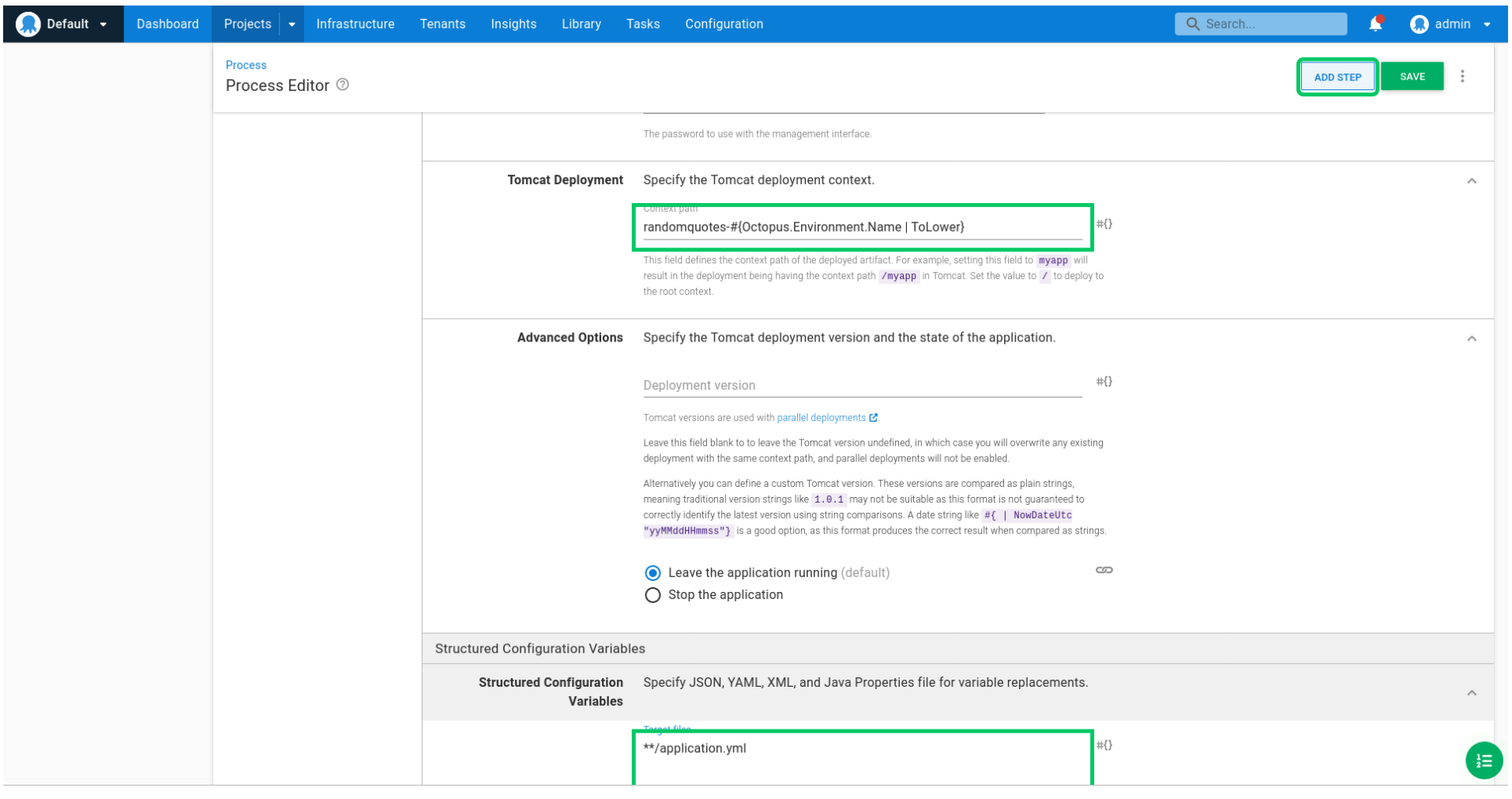
In the **Structured Configuration Variables** section, enter ****/application.yml** as the **Target files** value. Click **Save**.

> The slash in the filename is operating system specific. The pattern ****/application.yml** (with a forward slash) is for Linux targets, while ****\application.yml** (with a backslash) is for Windows targets.

The contents of the application.yml file is shown below. The **spring:profiles:active** variable we defined earlier will overwrite the Spring active profile in this YAML file:

Octopus Deploy

```
spring:
  profiles:
    active: Dev
  h2:
    console:
      enabled: true
  jpa:
    database-platform: org.hibernate.dialect.H2Dialect
  datasource:
    url: jdbc:h2:mem:testdb
    dbcp2:
      driver-class-name: org.h2.Driver
  flyway:
    locations: classpath:db/migration/{vendor}
```



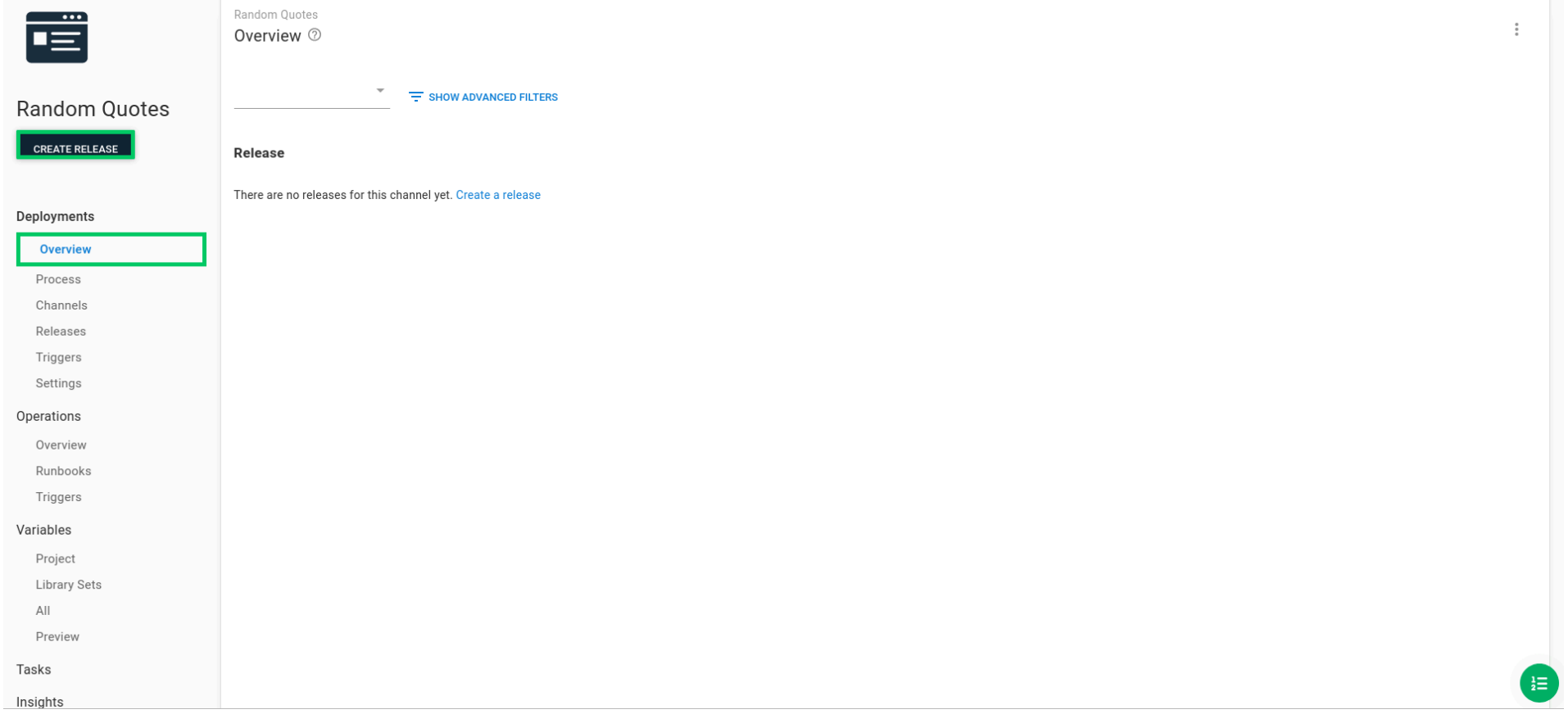
Deploy!

Now we have a deployment project in Octopus ready to deploy our Java application to our **Dev**, **Test**, and **Prod** environments. The next step is to create and deploy a release.

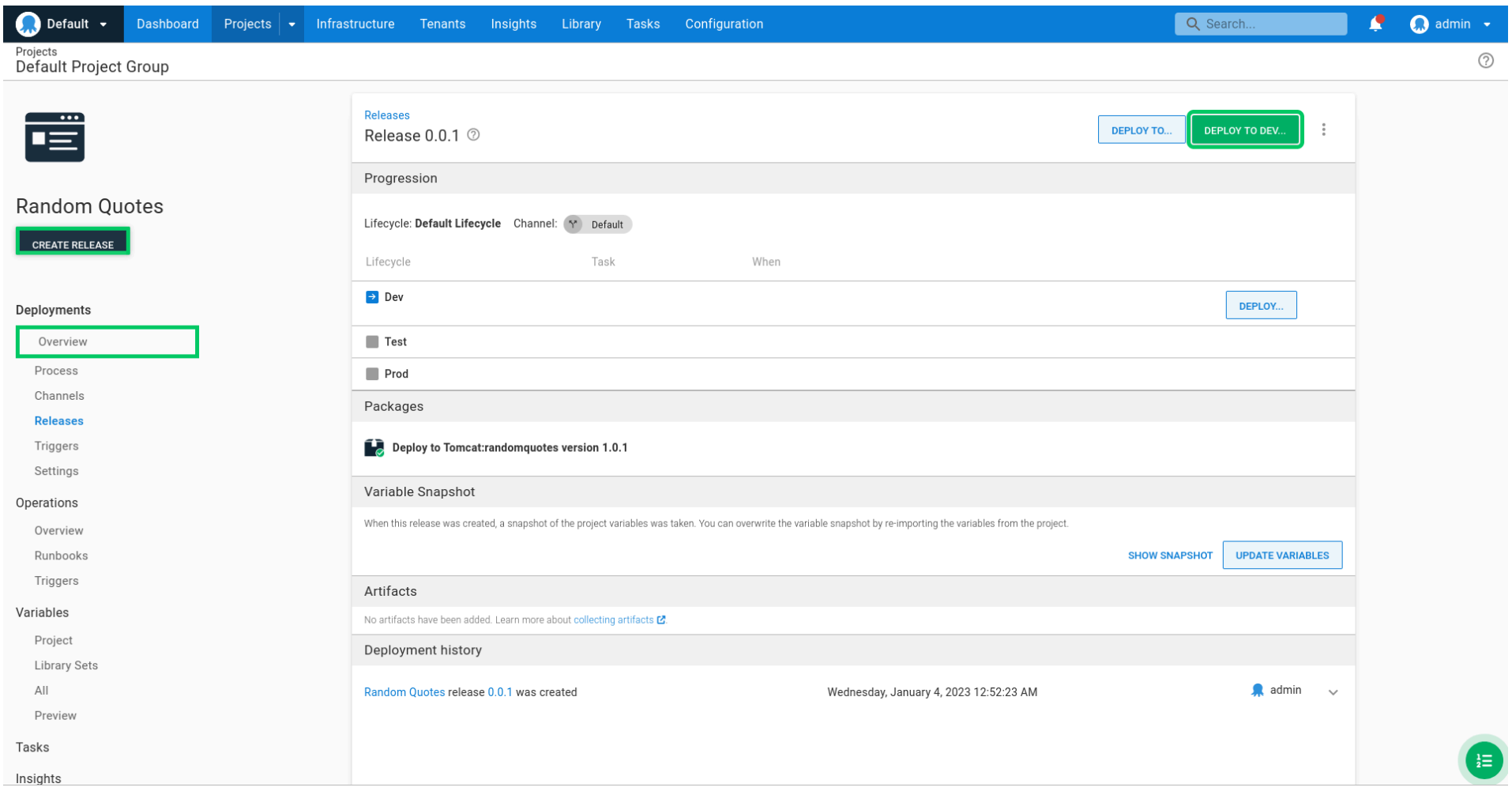
Click **CREATE RELEASE**.

The release creation screen provides an opportunity to review the packages that will be included and to add any release notes. By default, the latest package is selected automatically. Click **SAVE**:

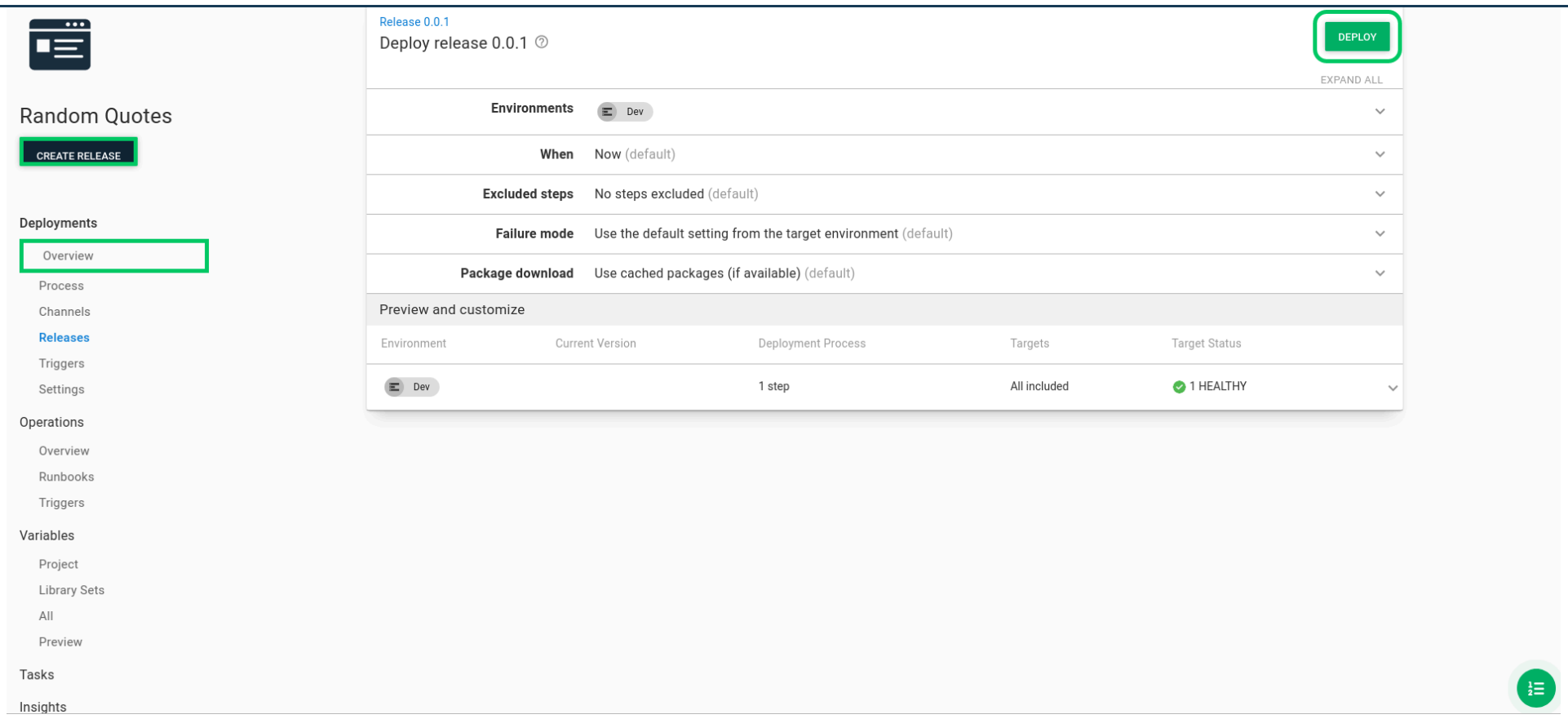
Octopus Deploy



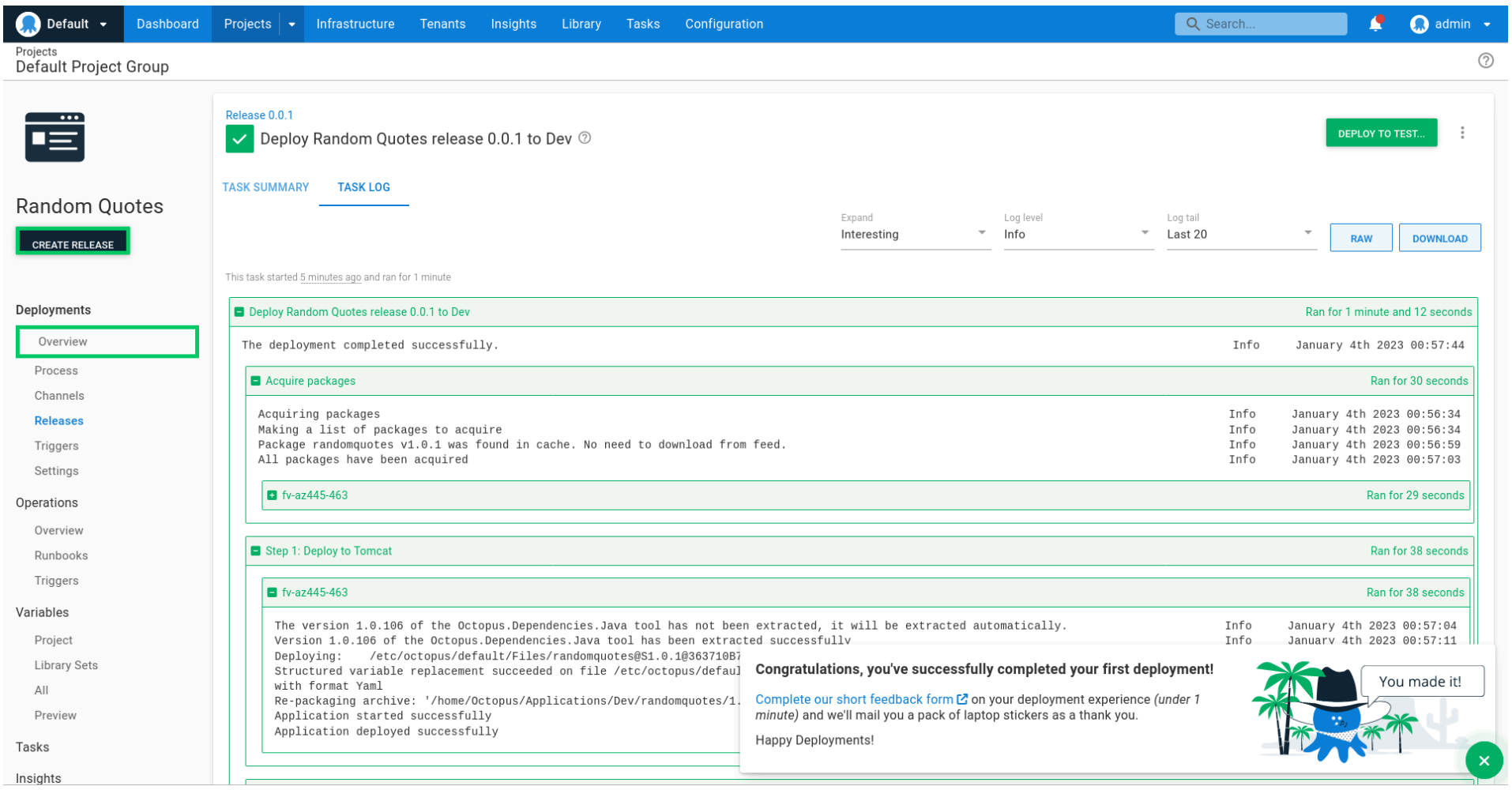
This screen allows you to select the environment that will be deployed into. Lifecycles can be used to customize the progression of deployments through environments (this is demonstrated later in the guide), however, we will accept the default option to deploy to the **Dev** environment by clicking **DEPLOY TO DEV...**:



Click **DEPLOY** to deploy the application into the **Dev** environment:



The application is then deployed:



Congratulations! You have now built and deployed your first application. Visit http://localhost:9091/randomquotes-dev in a browser (replacing **localhost:9091** with the IP address and port of your Tomcat instance) to view a random quote. Note the context path matches the templated value of the context path that we deployed the application to. Ensuring each environment deploys to a unique context path is how we can use a single Tomcat instance to host multiple environments:

Octopus Deploy

> Simplicity is not the goal. It is the by-product of a good idea and modest expectations.
>
> — Paul Rand

Refresh

© 2018 - Random Quotes | Version 1.0.1 running in Dev

Quote count: 2

Continuous deployments

The process of deploying a successful build to the **Dev** environment is currently a manual one; GitHub Actions pushes the file to Octopus, and we must trigger the initial deployment to the **Dev** environment from within Octopus. Typically though, deployments to the **Dev** environment will be performed automatically if a build and all of its tests succeed.

To trigger the initial deployment to the **Dev** environment after a successful build, we will go back to the project we created in GitHub Actions and add an additional step to create an Octopus release and then deploy it to the **Dev** environment.

When added to the workflow file, the YAML below creates a new release in Octopus with each build and deploys it to the development environment.

Note that the package versions are explicitly defined via the output variable called **octoversion** generated in previous steps:

```
    - name: Create Octopus Release
      uses: OctopusDeploy/create-release-action@v1.1.1
      with:
        api_key: ${{ secrets.OCTOPUS_API_TOKEN }}
        project: Random Quotes
        server: ${{ secrets.OCTOPUS_SERVER_URL }}
        deploy_to: Dev
        packages: ${{ steps.get_octopus_artifact.outputs.octoversion }}
```

Octopus Deploy

Additional configuration

Now we will explore some of the more advanced features of Octopus that allow us to customize the deployment progression through environments, secure deployments to production environments, add deployment sign offs, view the audit logs, and add notifications.

Lifecycles

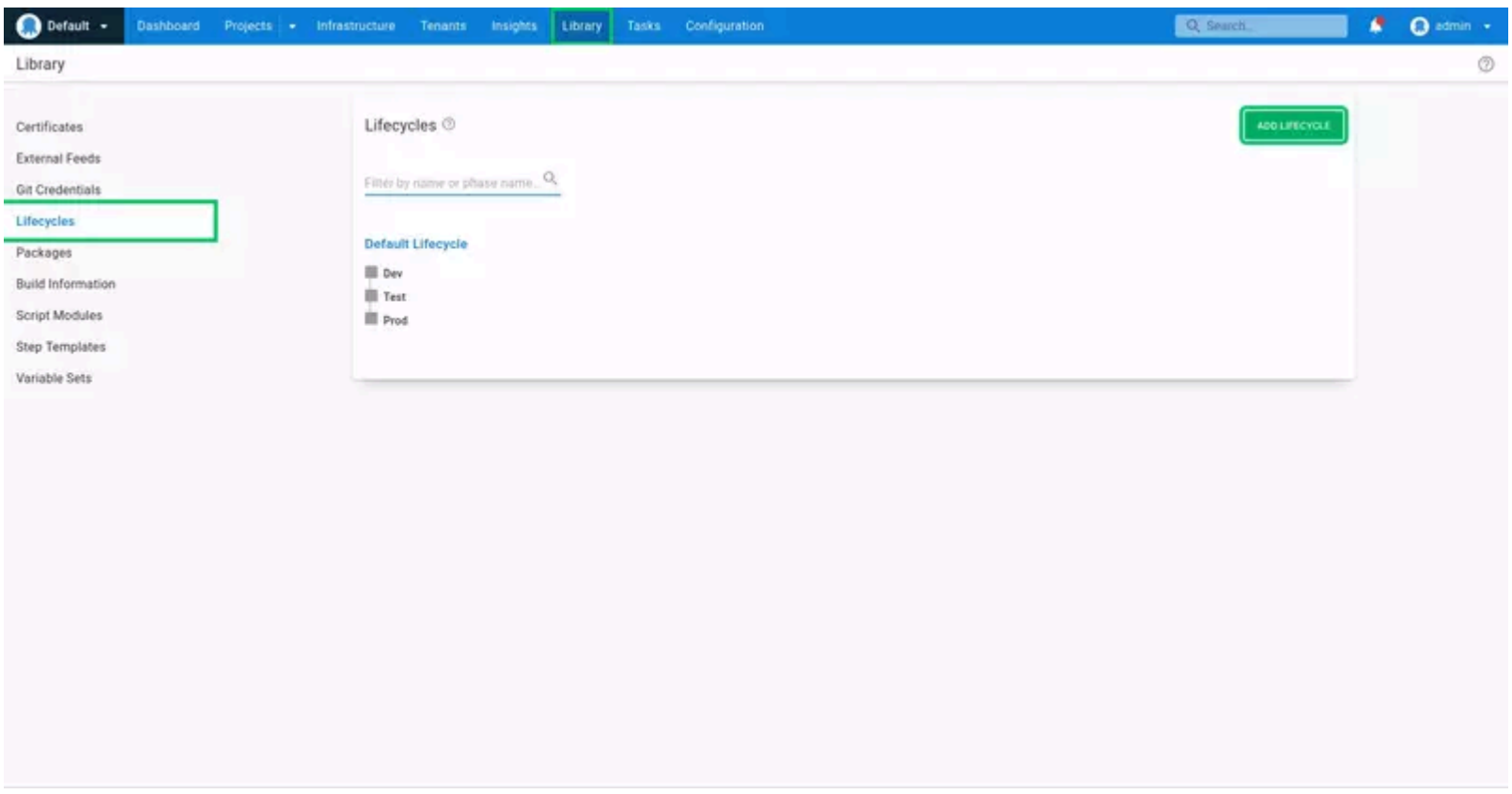
Our project currently uses the default lifecycle, which defines a progression through all the environments in the order they were created.

A custom lifecycle allows the progression of a deployment to be further refined, by defining only a subset of environments that can be deployed to, allowing some environments to be skipped entirely, or requiring that a minimum number of environments are successfully deployed to before moving onto the next environment.

Here we will create a custom lifecycle that makes deployments to the **Dev** environment optional. This means that initial deployments can be made to the **Dev** *or* **Test** environments, but a successful deployment *must* be made to the **Test** environment before it can be progressed to the **Prod** environment.

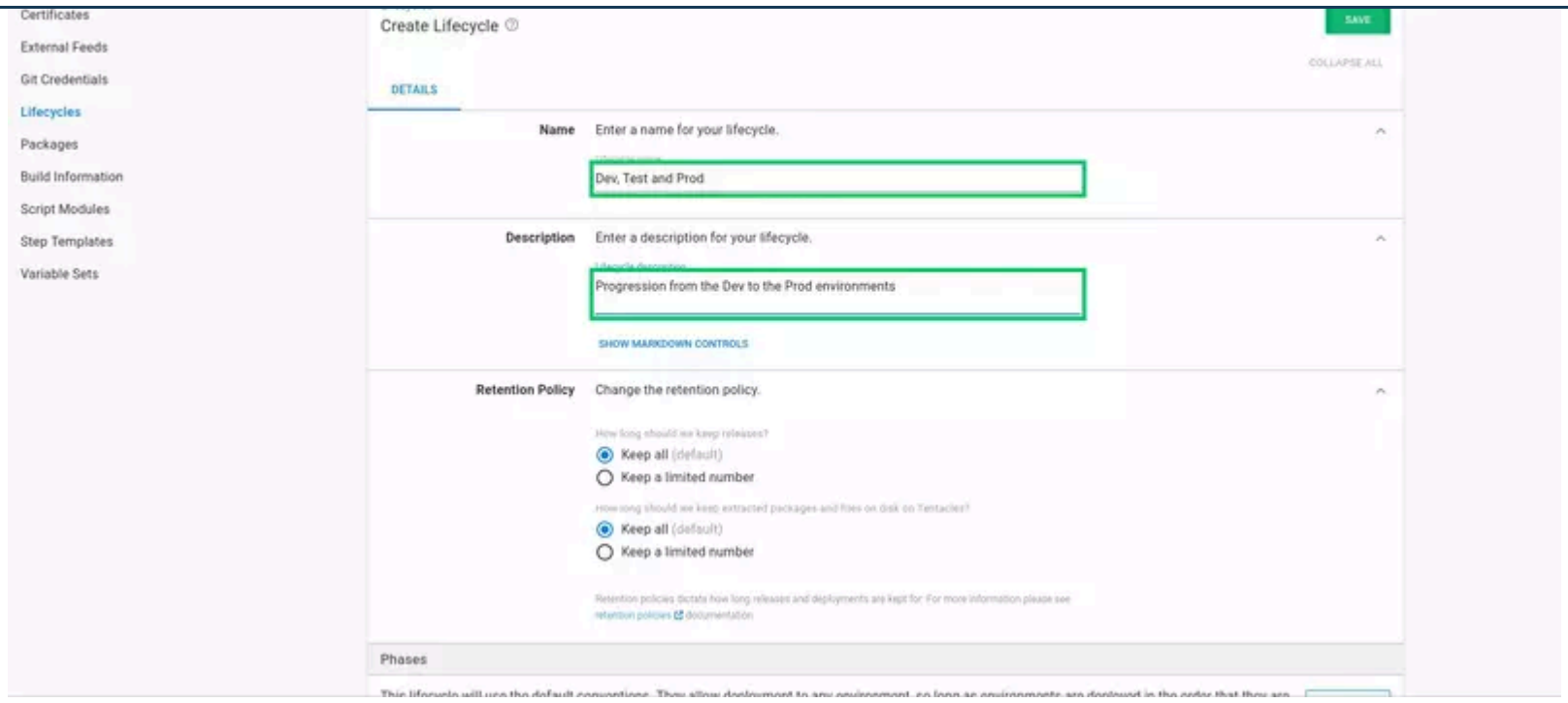
Skipping the **Dev** environment like this may be useful for promoting a release candidate build directly to the **Test** environment for product owners or testers to review.

Click the **Library** link, click the **Lifecycles** link, and click **ADD LIFECYCLE**:



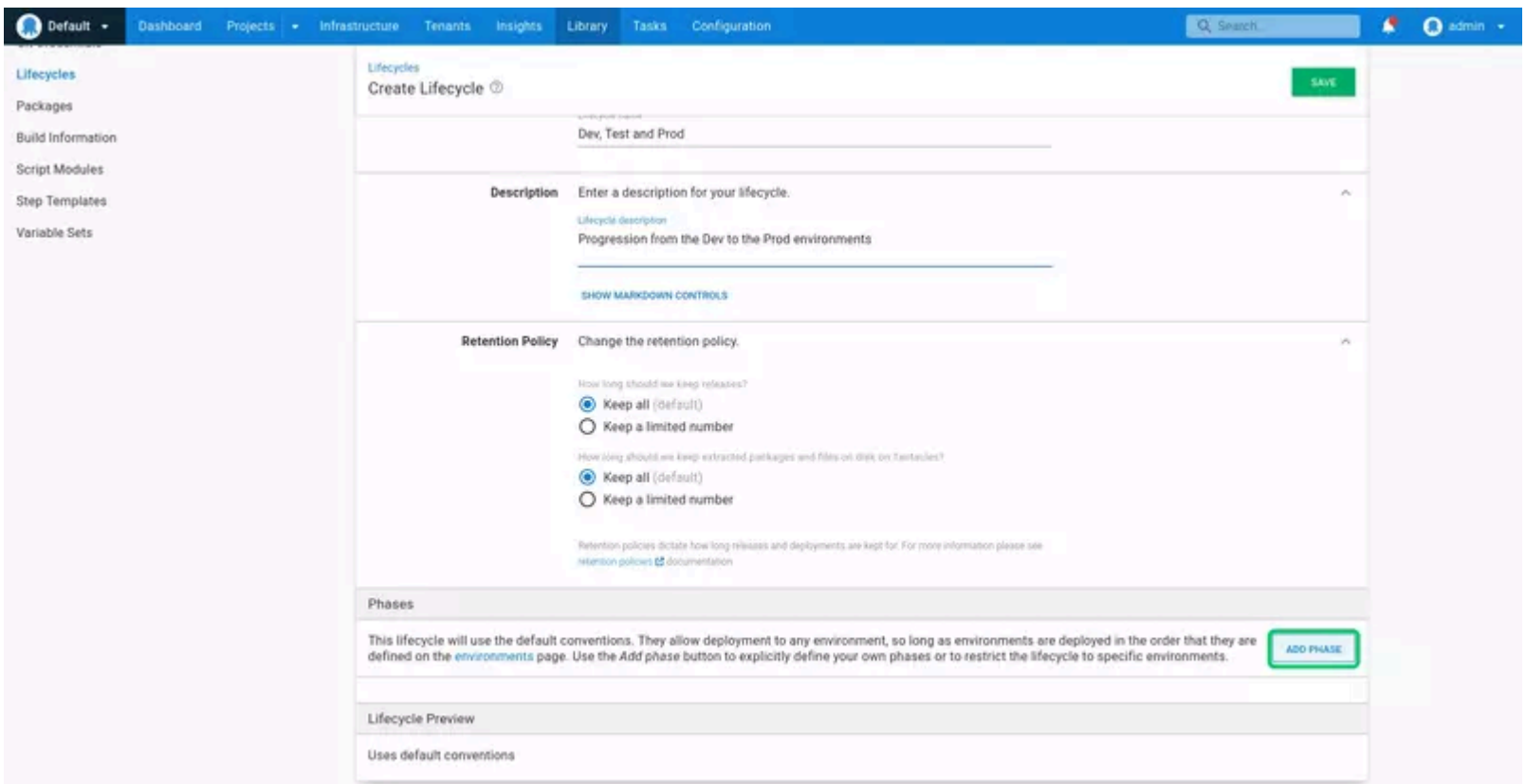
Set the lifecycle name to **Dev, Test, and Prod**, and the description to **Progression from the Dev to the Prod environments**:

Octopus Deploy



Phases are used to group environments that can accept a deployment. Simple lifecycles, such as the lifecycle we are creating, have a one-to-one relationship between phases and environments.

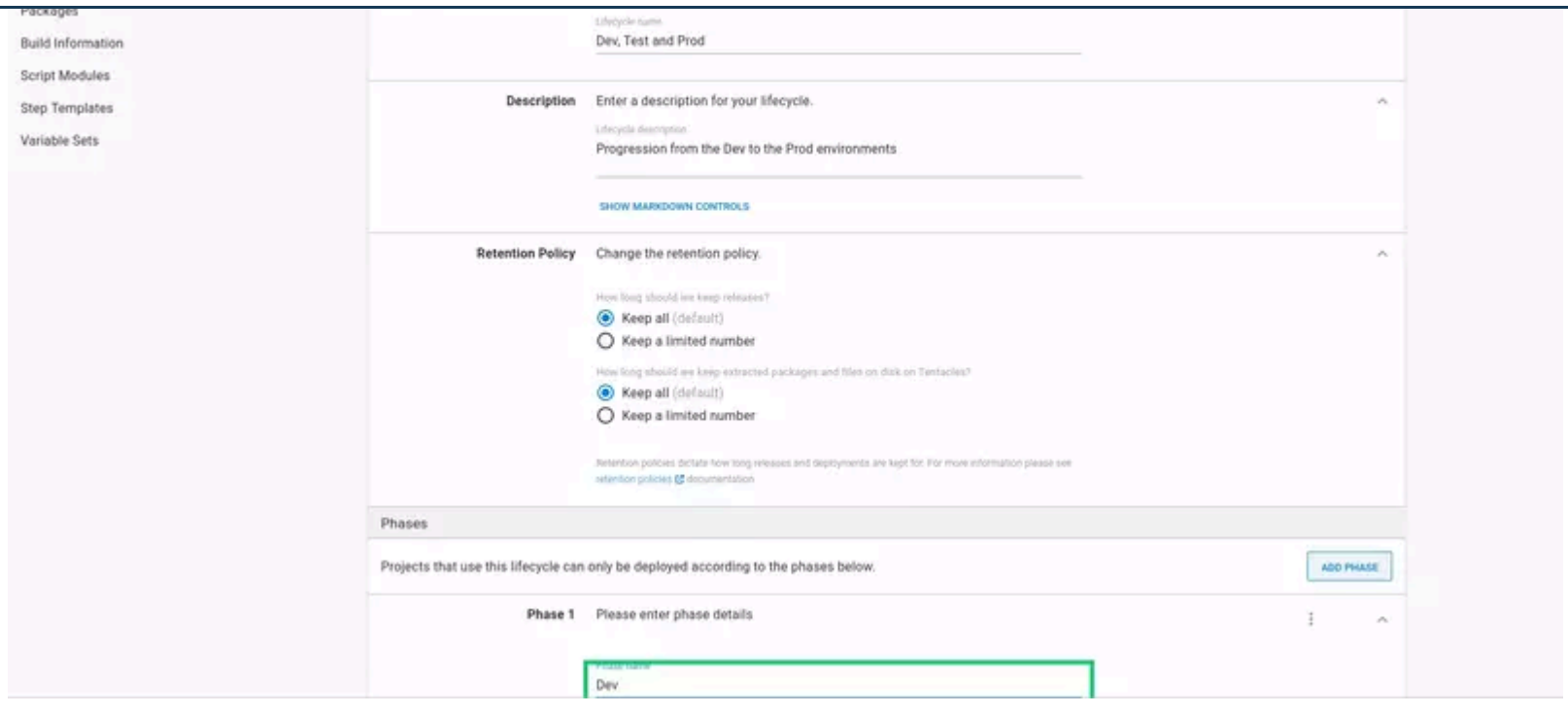
Click **ADD PHASE**:



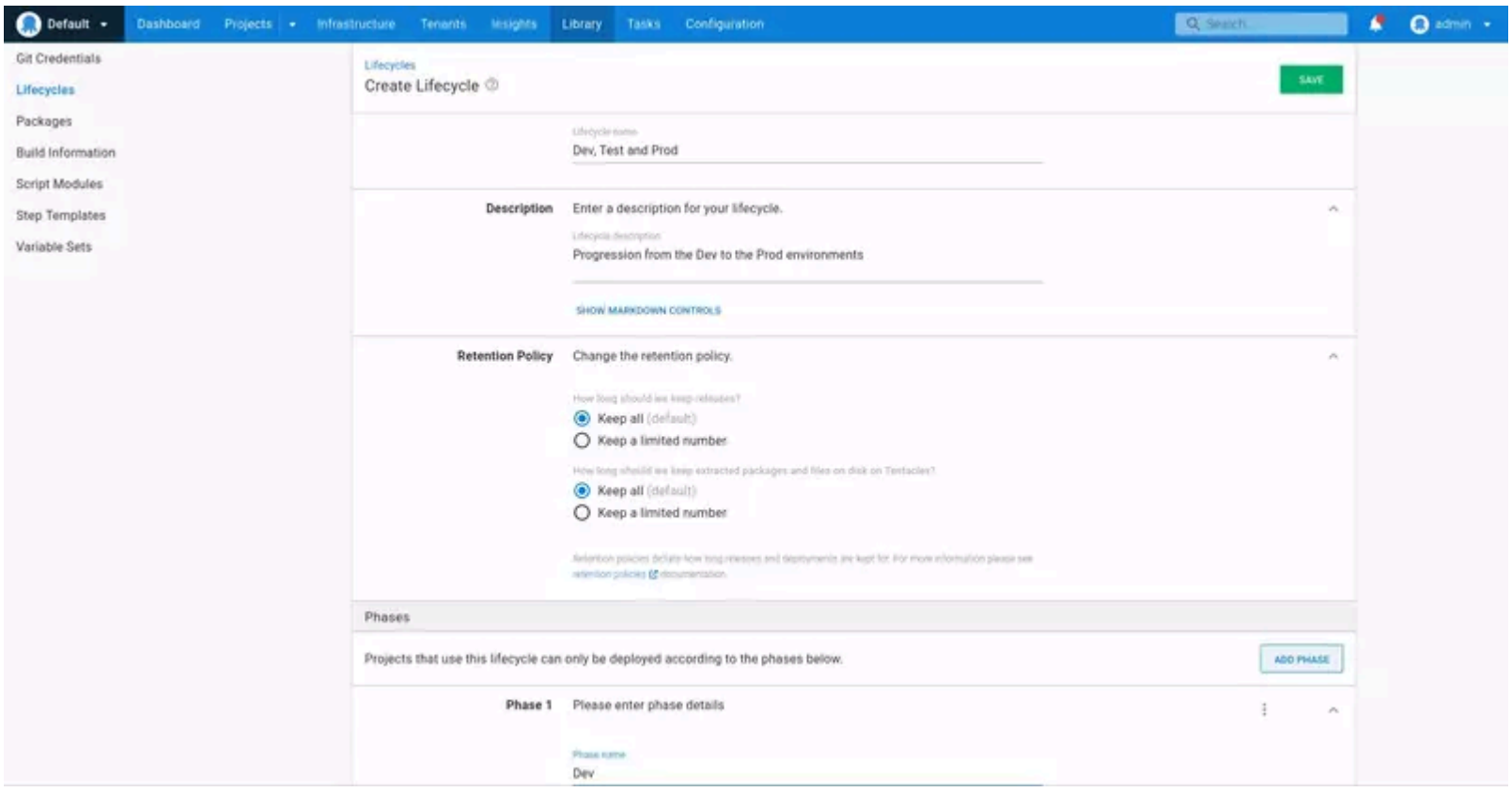
Enter **Dev** as the phase name, and select the **Optional phase** option. This means that deployments can skip this phase and any environments defined in it, and deploy directly to the next phase.

Because we are mapping each environment to its own phase, the name of the phase matches the name of the environment:

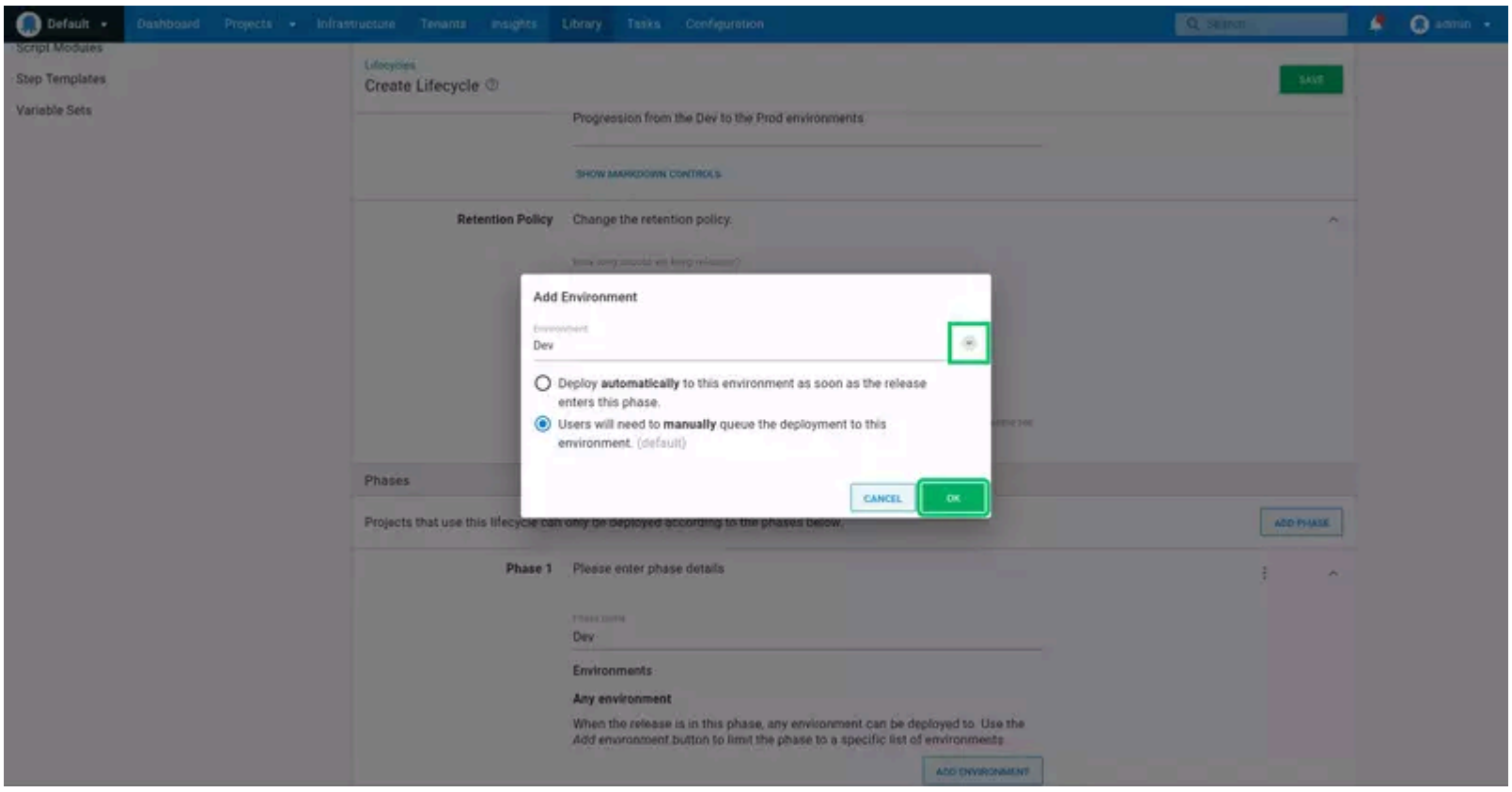
Octopus Deploy



Click **ADD ENVIRONMENT**:

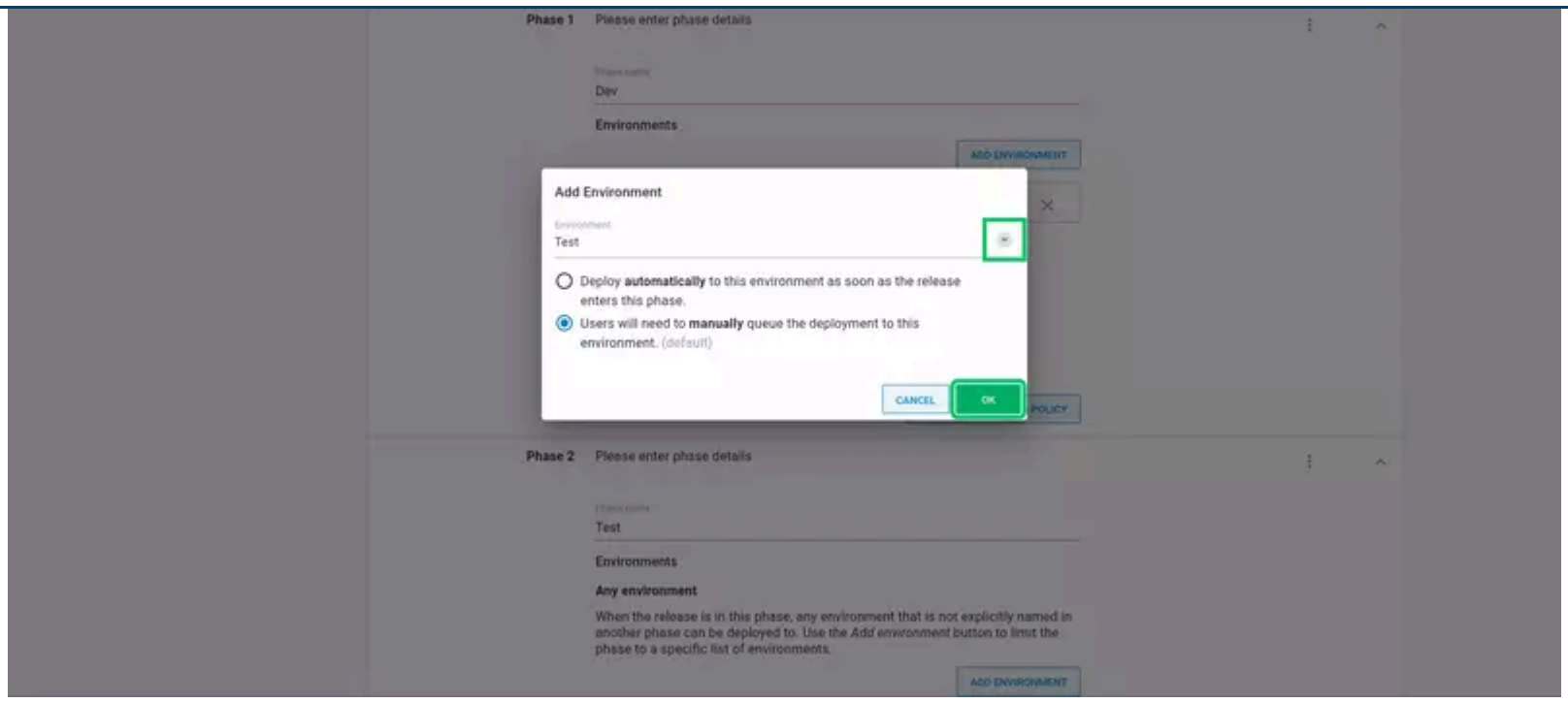


Click the dropdown arrow, select the **Dev** environment, and click **OK**:

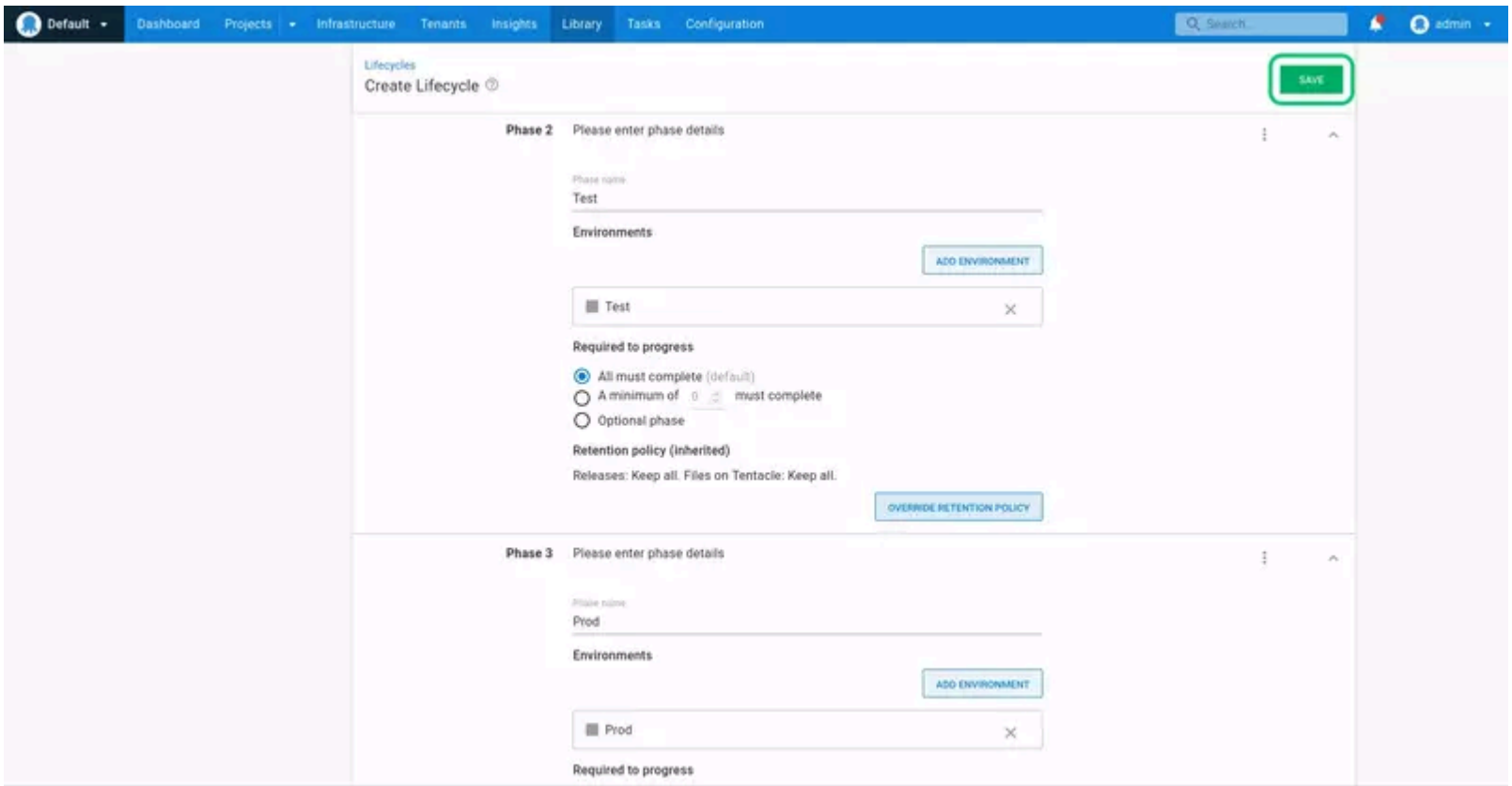


Repeat the process to add a new phase for the **Test** and **Prod** environments, leaving the default **All must complete** option selected:

Octopus Deploy



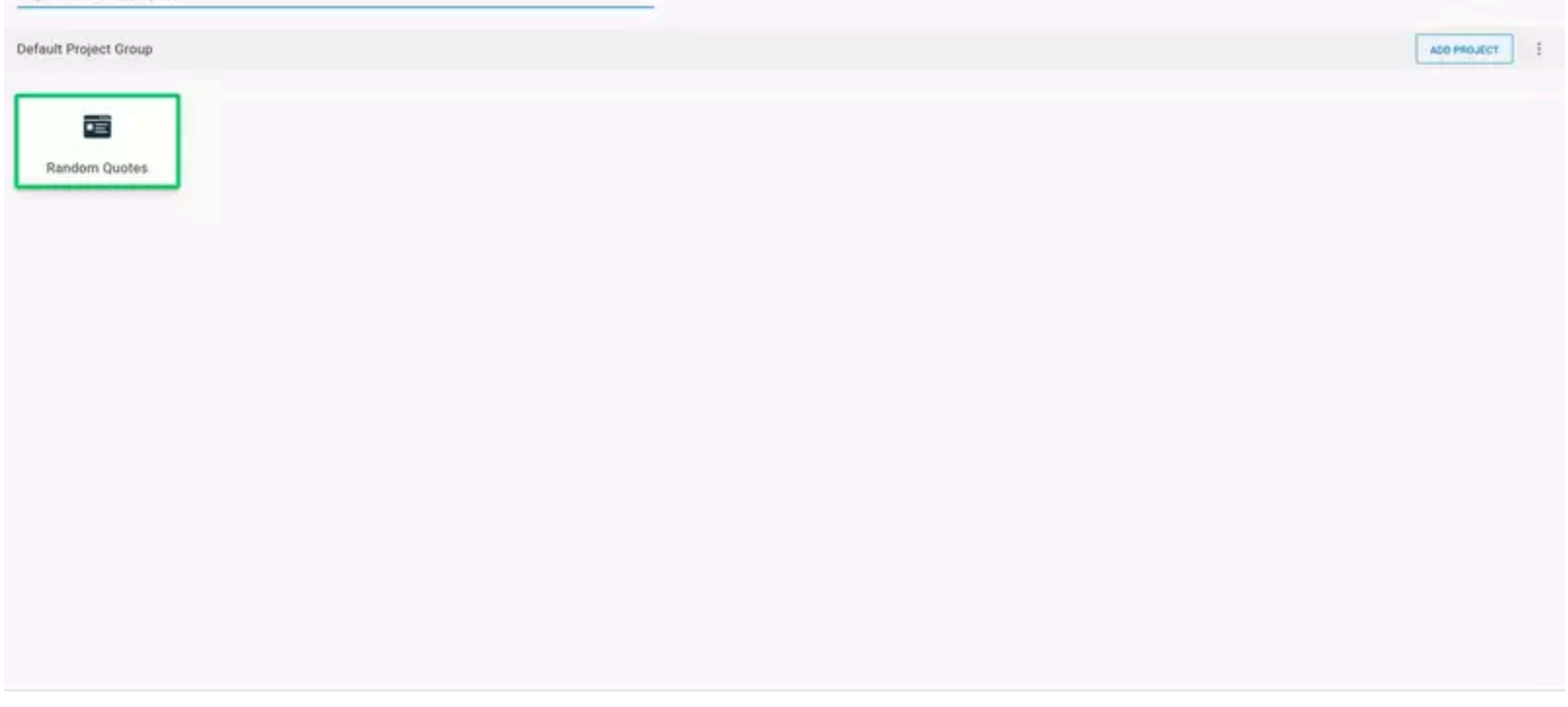
Click **SAVE**:



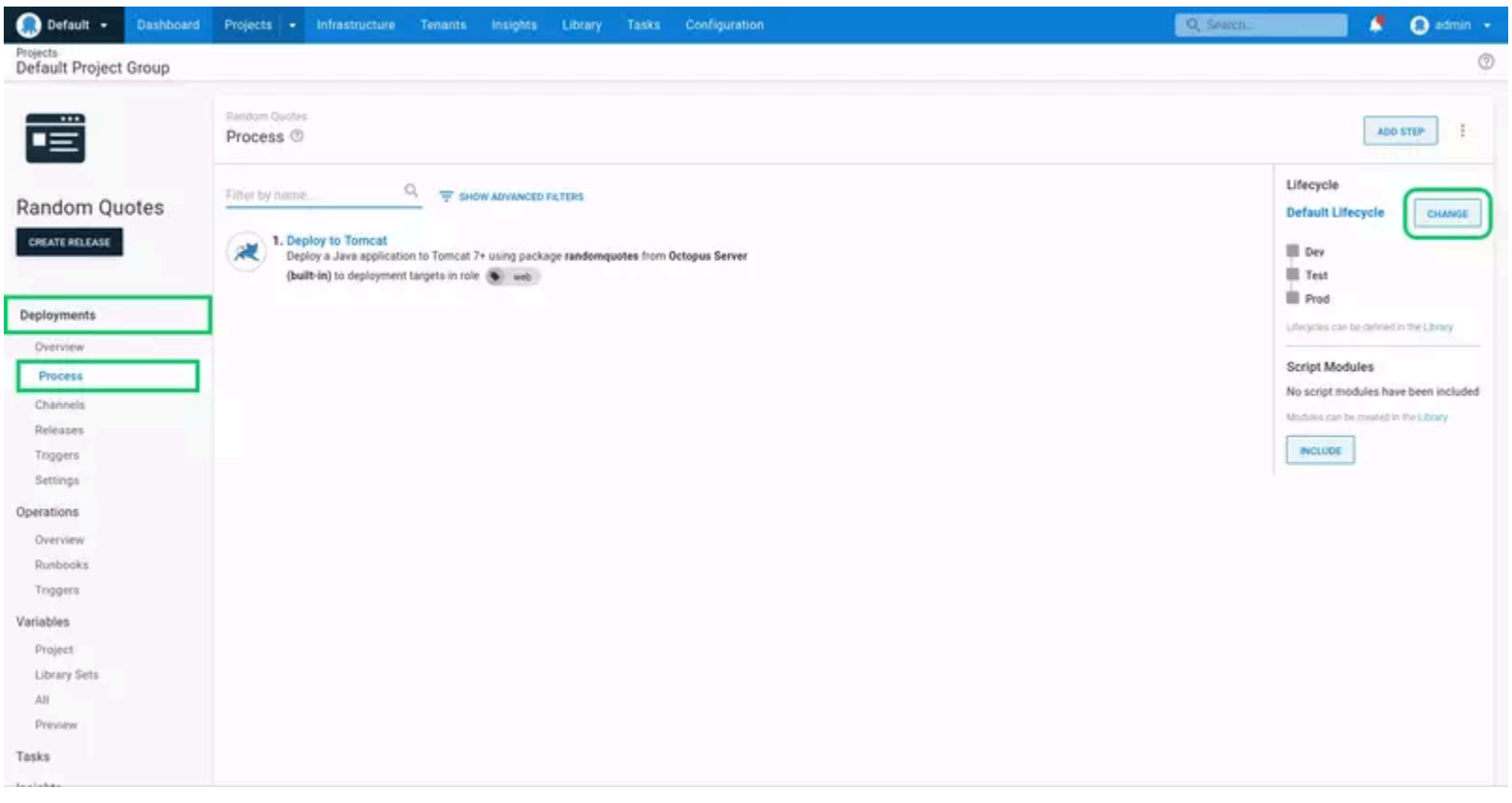
Now, we need to switch the deployment project from the **Default Lifecycle** to the newly created lifecycle.

Click the **Projects** link, and click the **Random Quotes** project tile:

Octopus Deploy

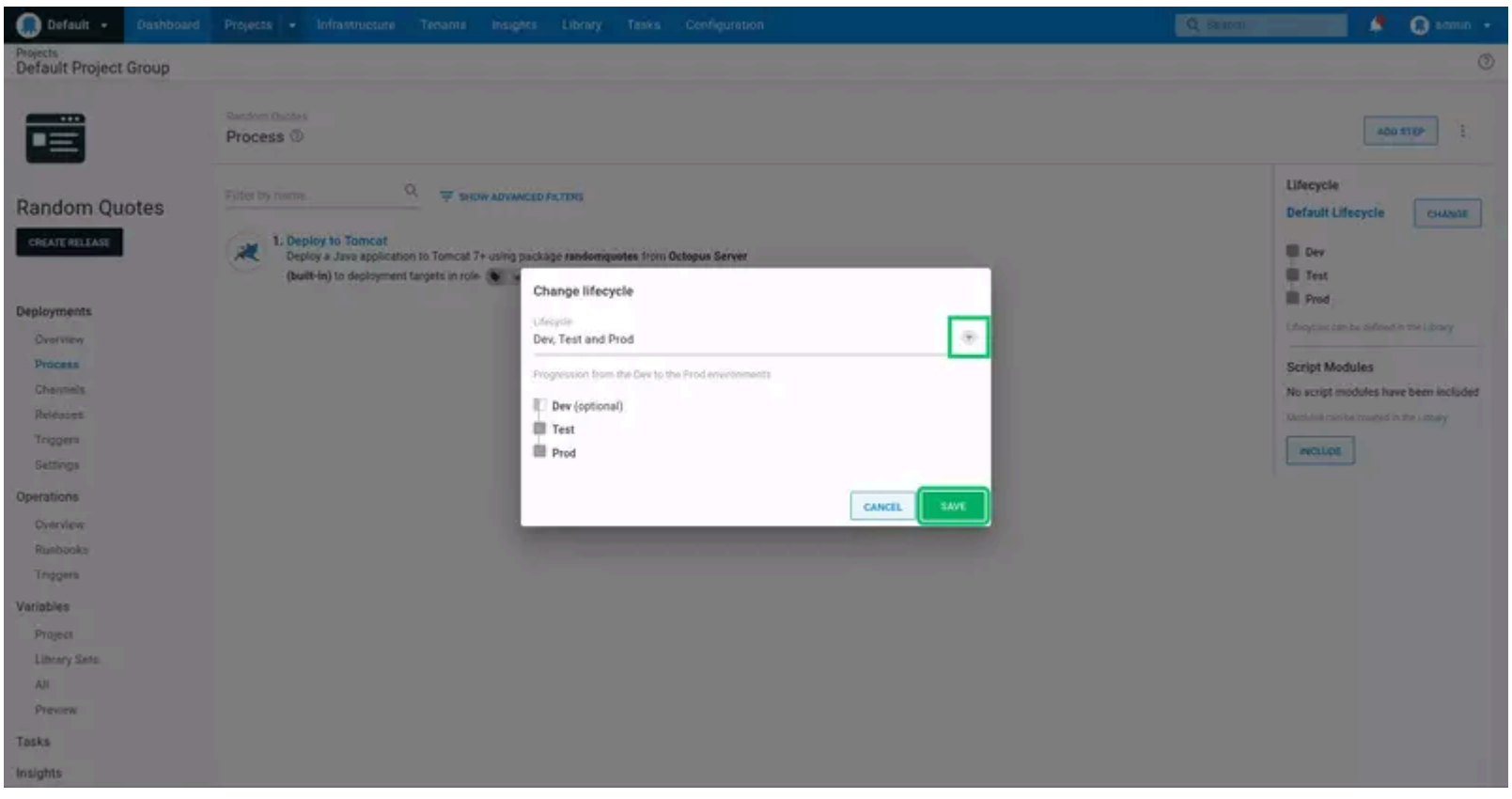


Click the **Process** link, and click **CHANGE**:



Select the **Dev, Test, and Prod** lifecycle. Notice the **Dev** environment is shown as optional.

Click **SAVE**:



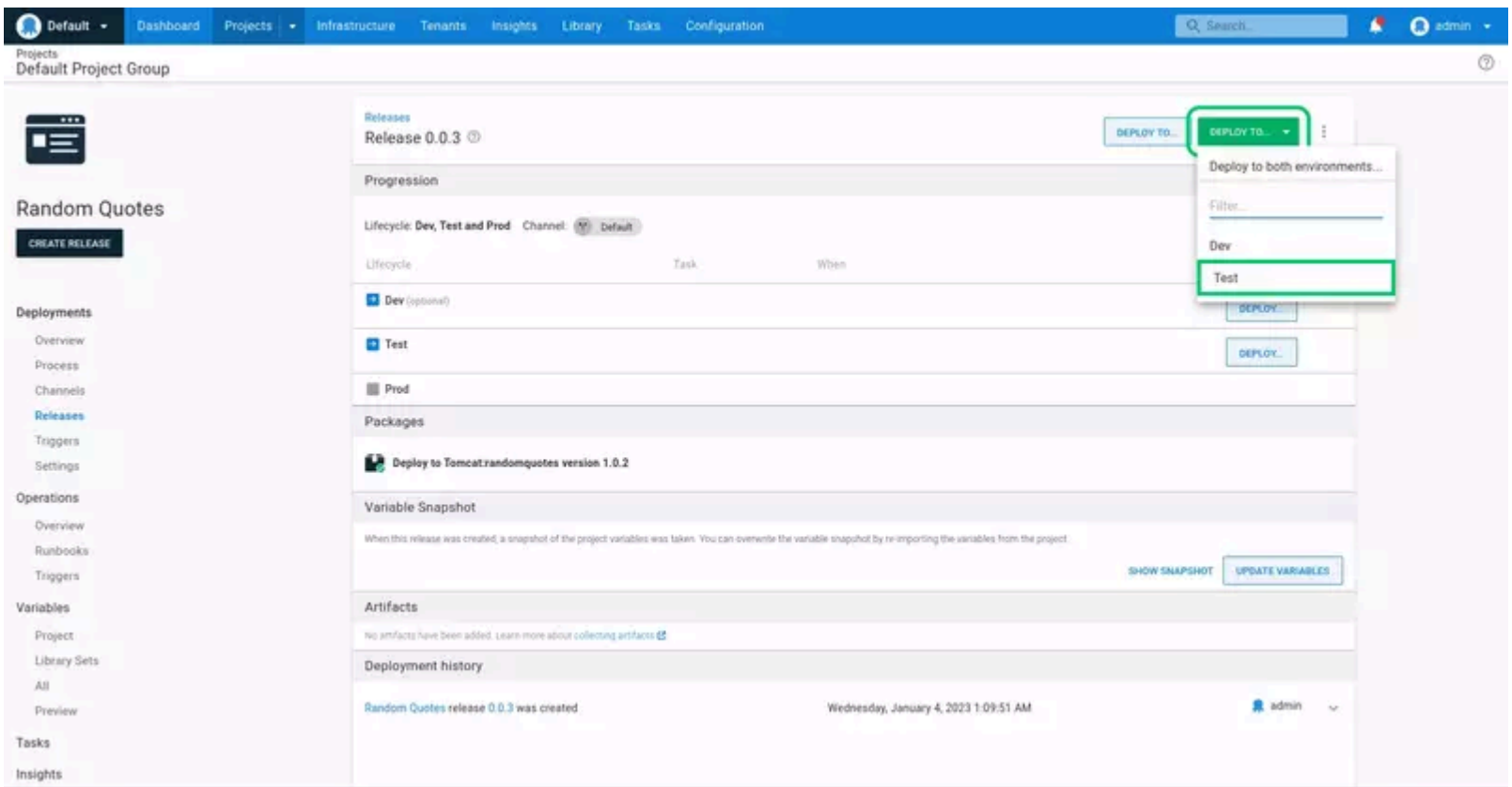
Click **CREATE RELEASE**, and click **SAVE** to save the new release:

Octopus Deploy

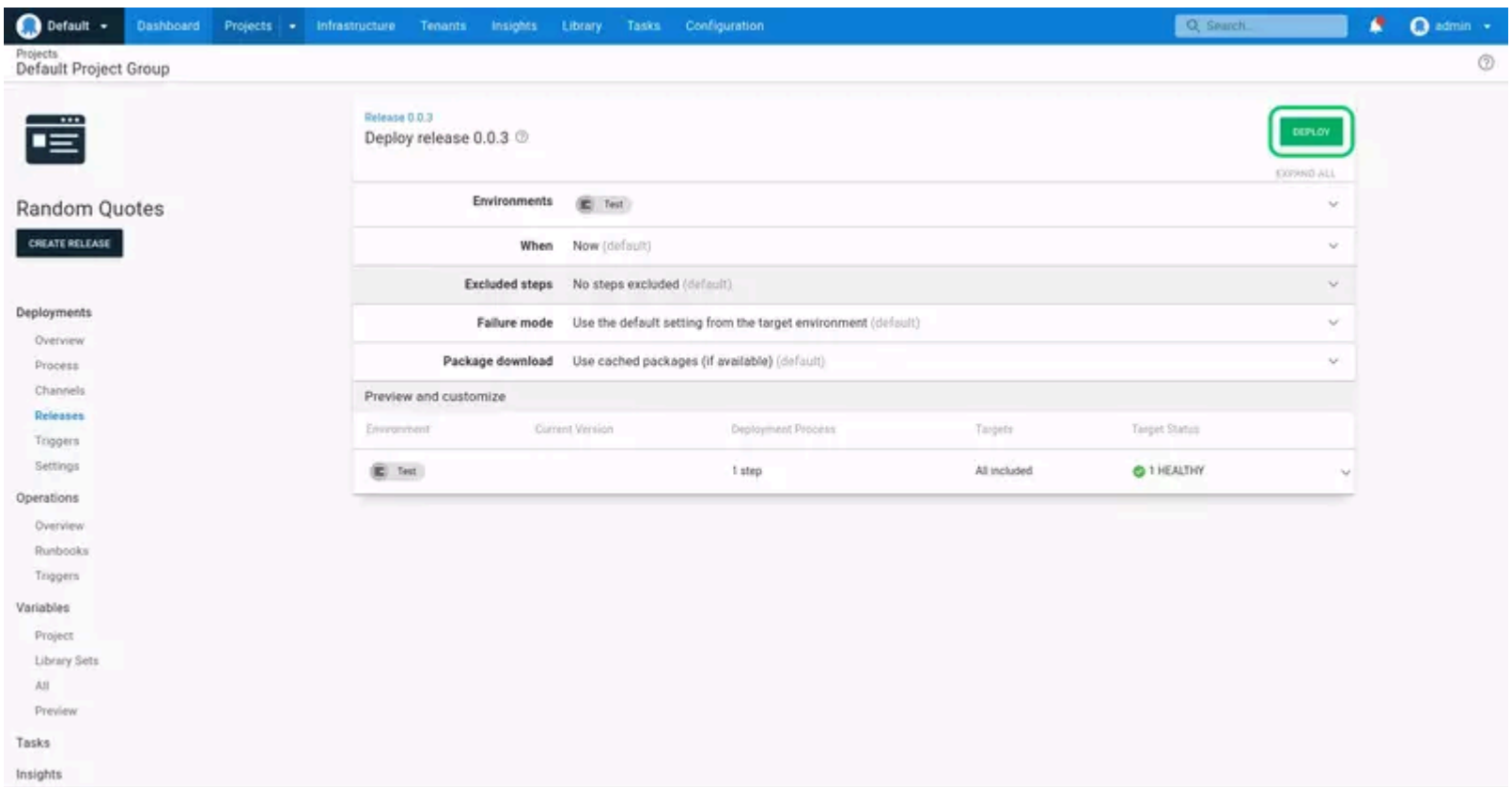
Because the **Dev** environment has been configured as optional, the initial release can be made to either the **Dev** or **Test** environments.

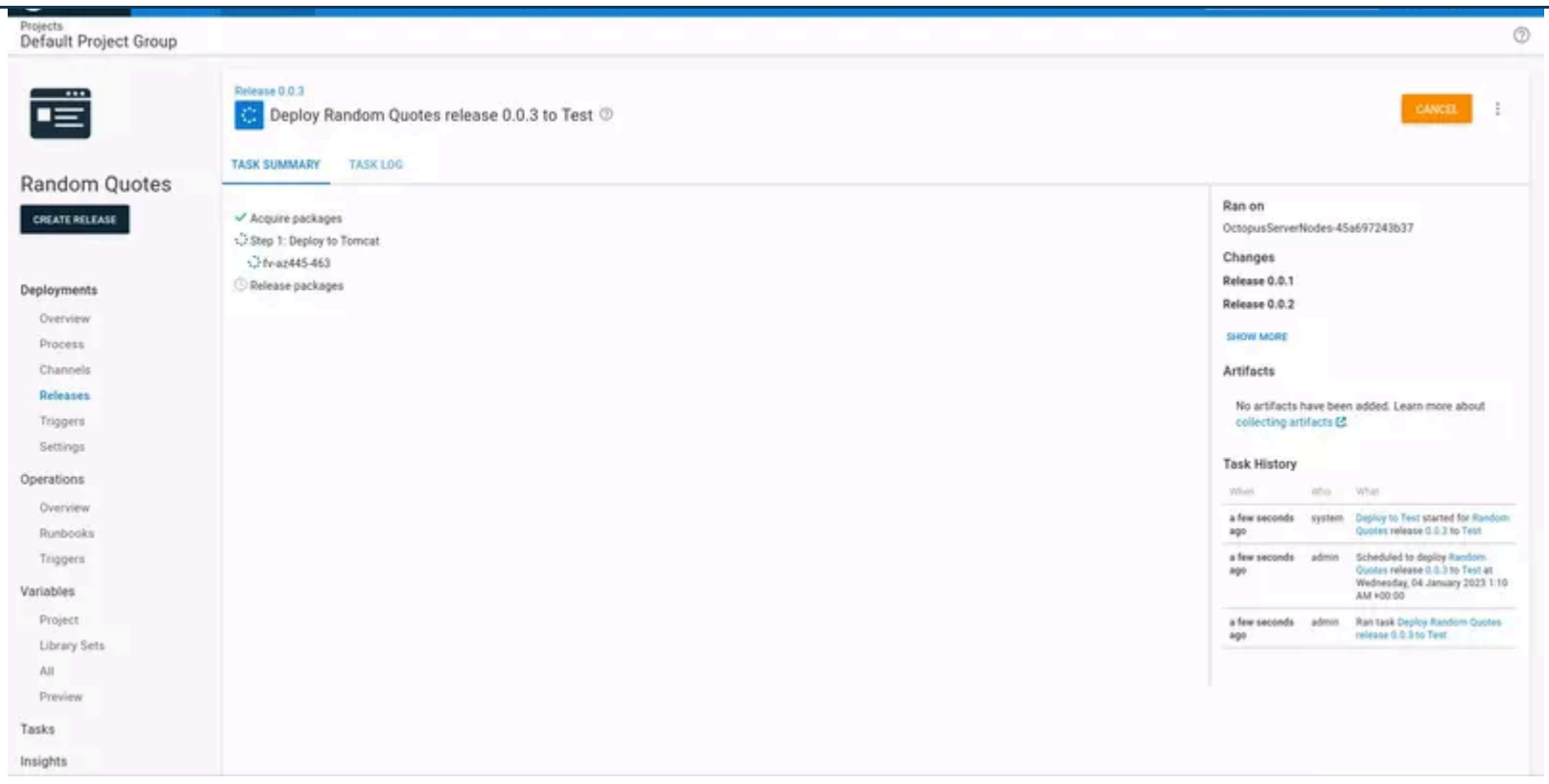
We'll skip the **Dev** environment and deploy straight to **Test**.

Click **DEPLOY TO...**, and select the **Test** environment:



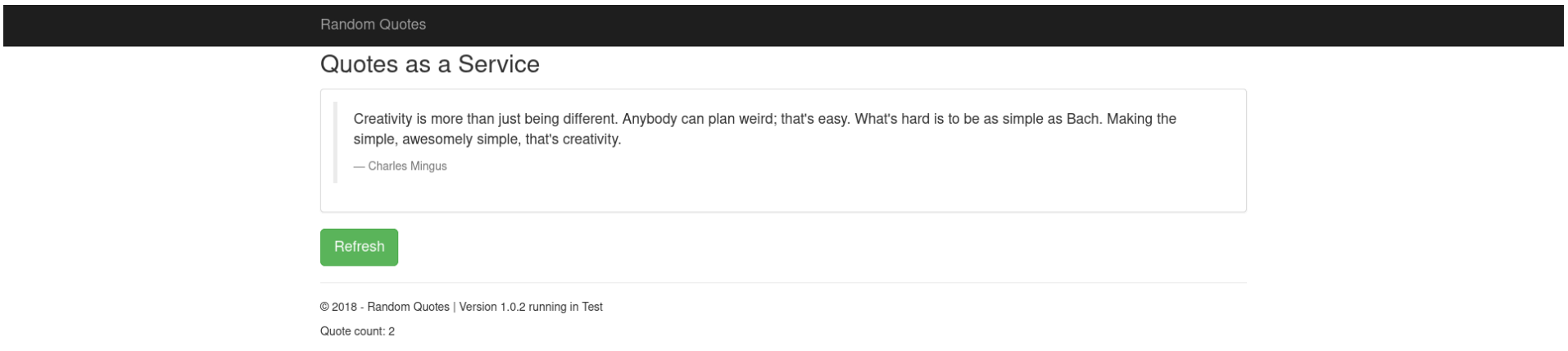
Click **DEPLOY** to deploy the application to the **Test** environment:





Opening http://localhost:9091/randomquotes-test displays the copy of the Random Quotes application deployed to the **Test** environment. Note the context path **randomquotes-test** is the templated value of the context path that we deployed the application to.

Also see the footer text that says **running in Test**. This is the result of the contents of the deployed-application.yml file being processed during deployment to define the current Spring Boot profile to be the name of the environment that the application was deployed into:



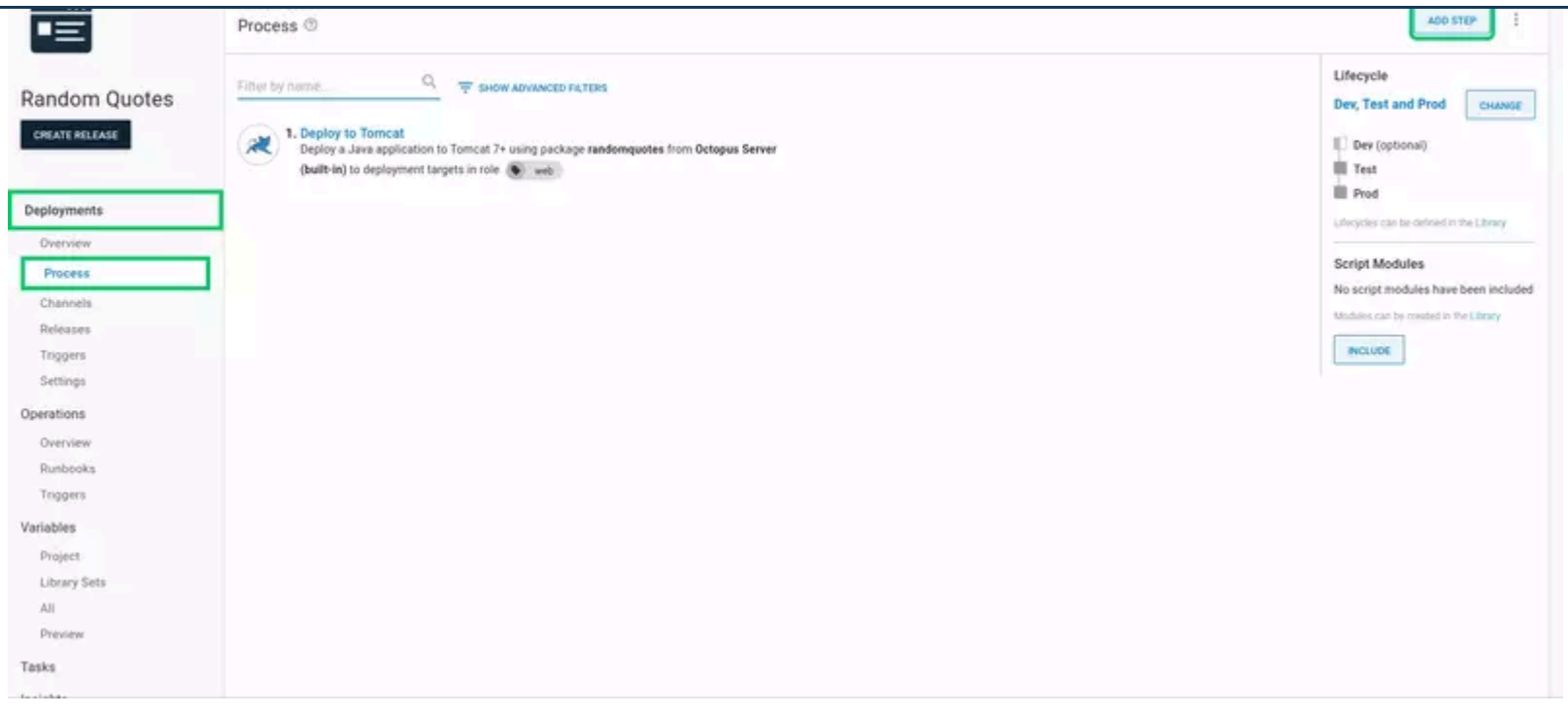
Approvals

It's a common business requirement to have testers or product owners manually verify that a particular build meets the requirements before a deployment can be considered successful.

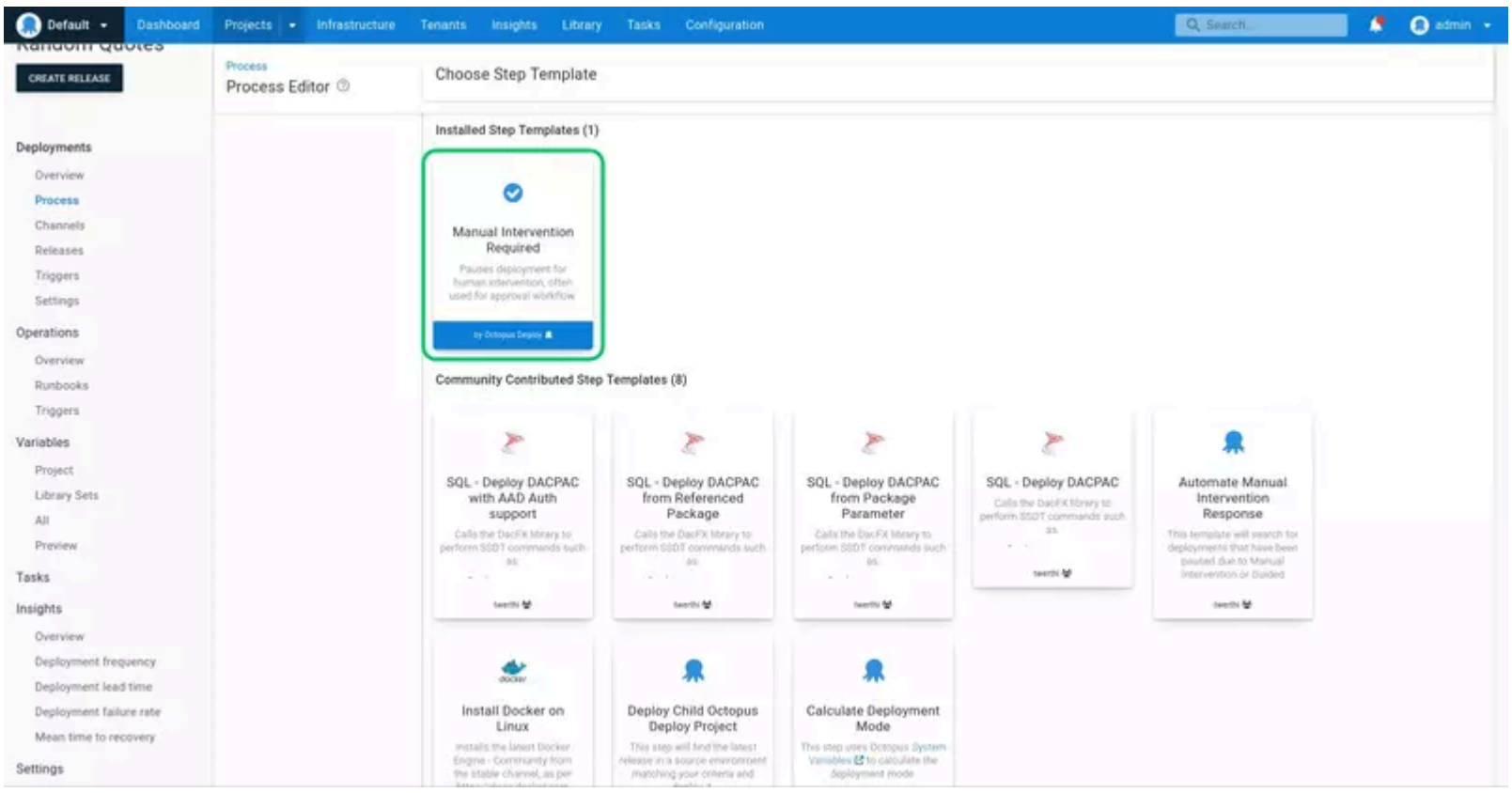
Octopus supports this workflow through the use of manual intervention steps. We'll add a manual intervention step to the end of the deployment process, which requires a responsible party to verify the build meets the requirements.

Open the Random Quotes project, click the **Process** link, and click the **ADD STEP** button:

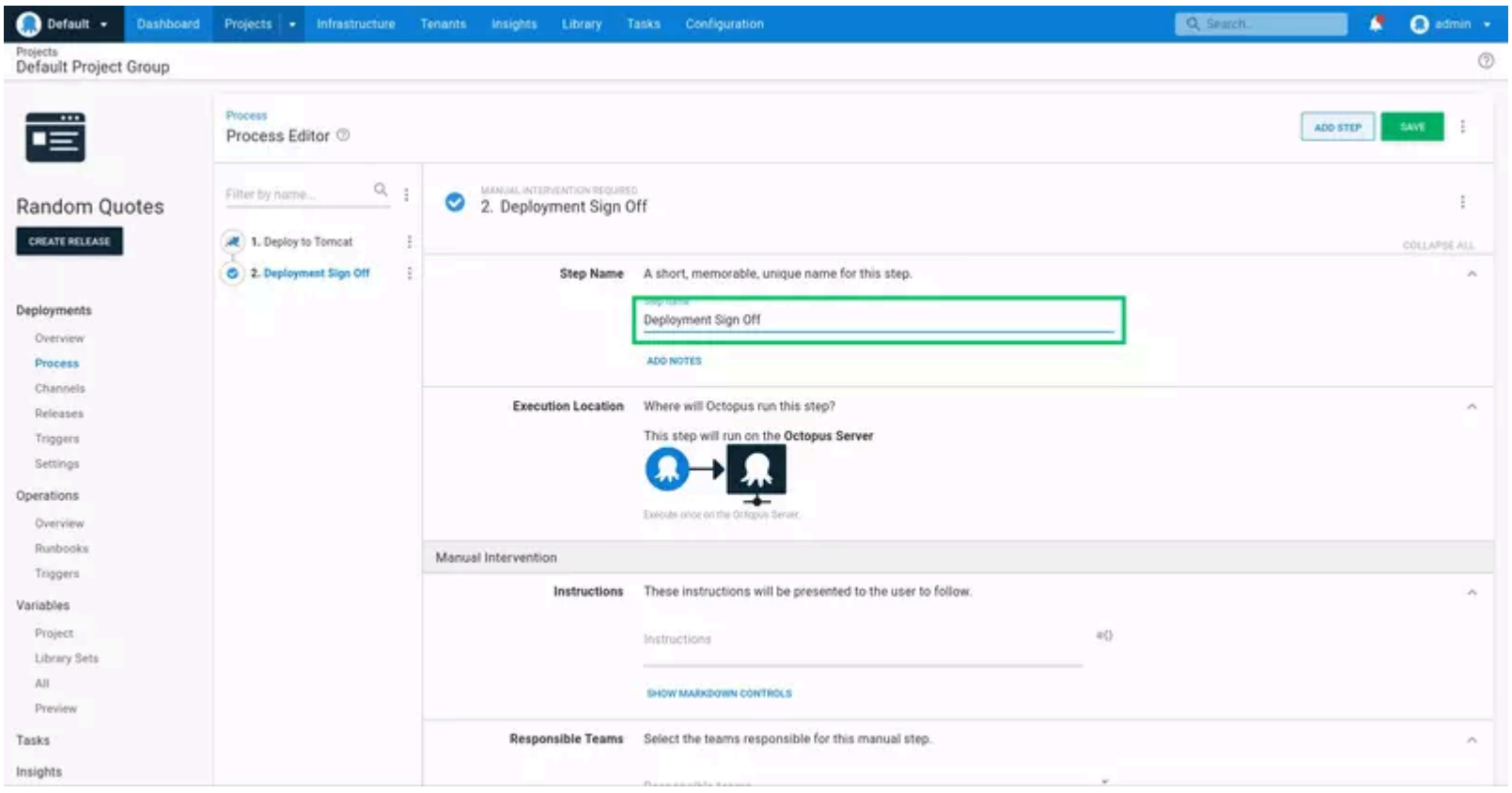
Octopus Deploy



Search for the **Manual Intervention Required** step, and add it to the process:

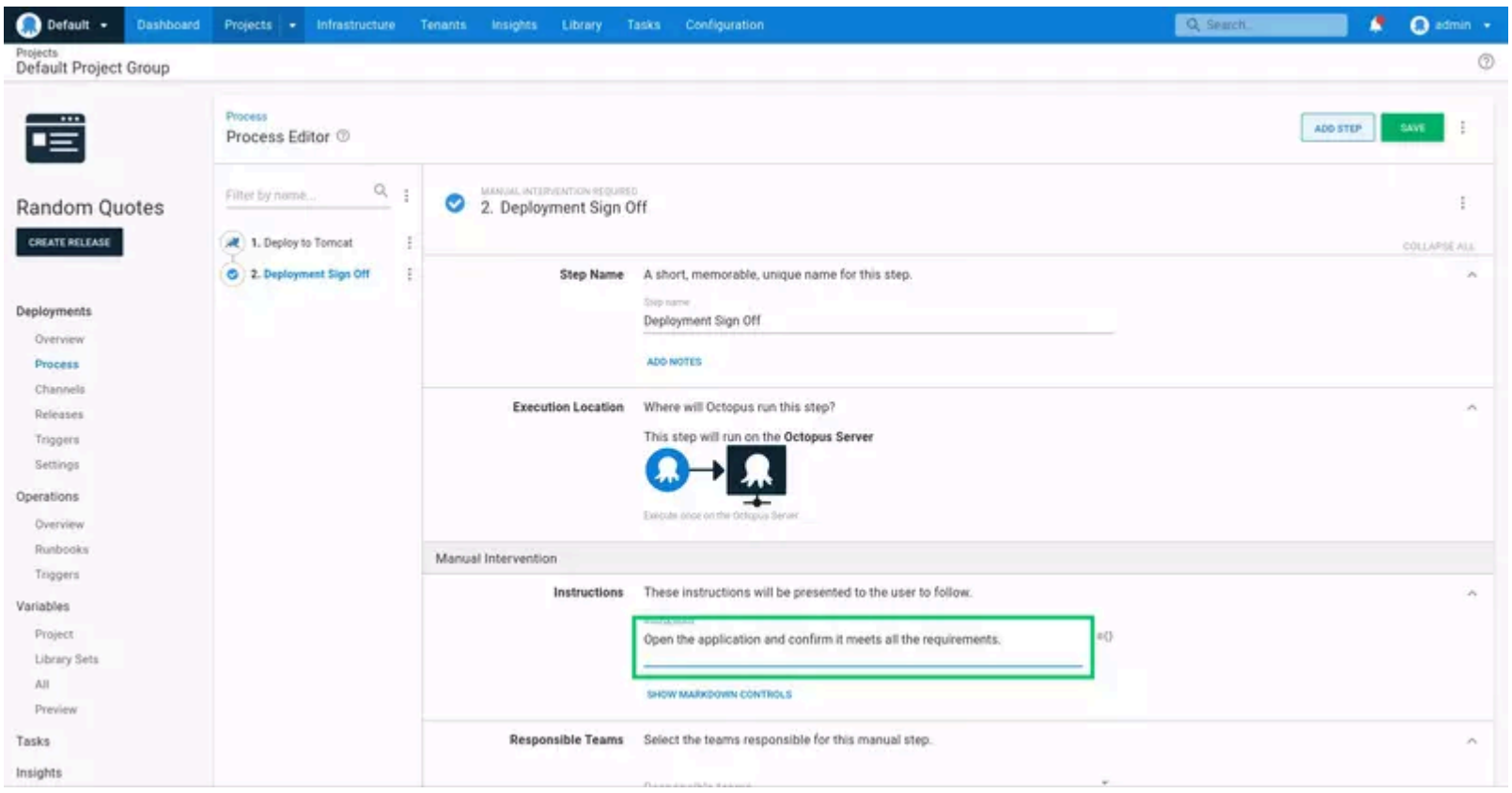


Enter **Deployment Sign Off** for the **Step Name**:



Enter the following for the **Instructions**:

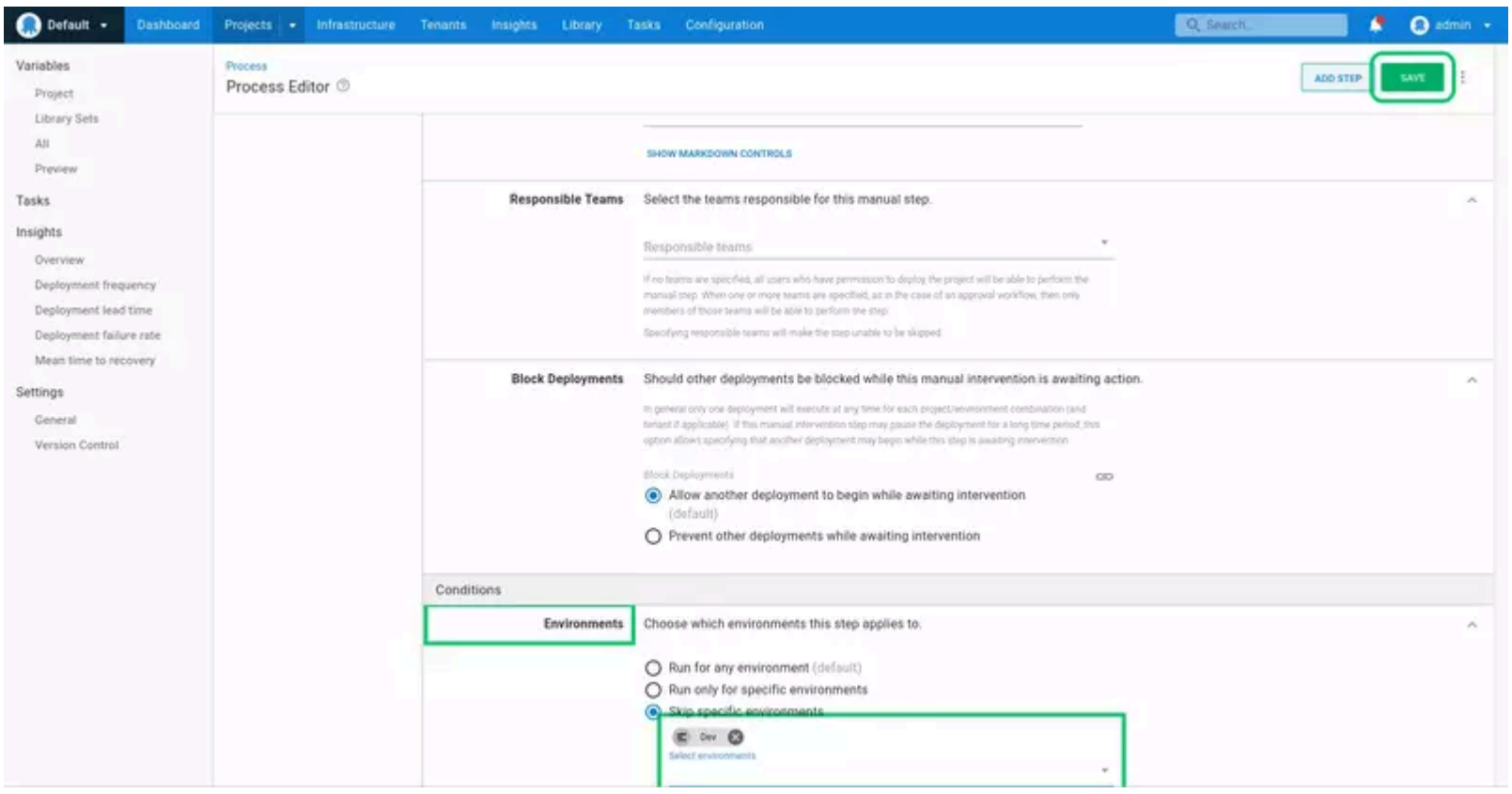
Octopus Deploy



Because every build is automatically deployed to the **Dev** environment, it doesn't make sense to force someone to manually approve all those deployments. To accommodate this, we do not enable the manual intervention step for deployments to the **Dev** environment.

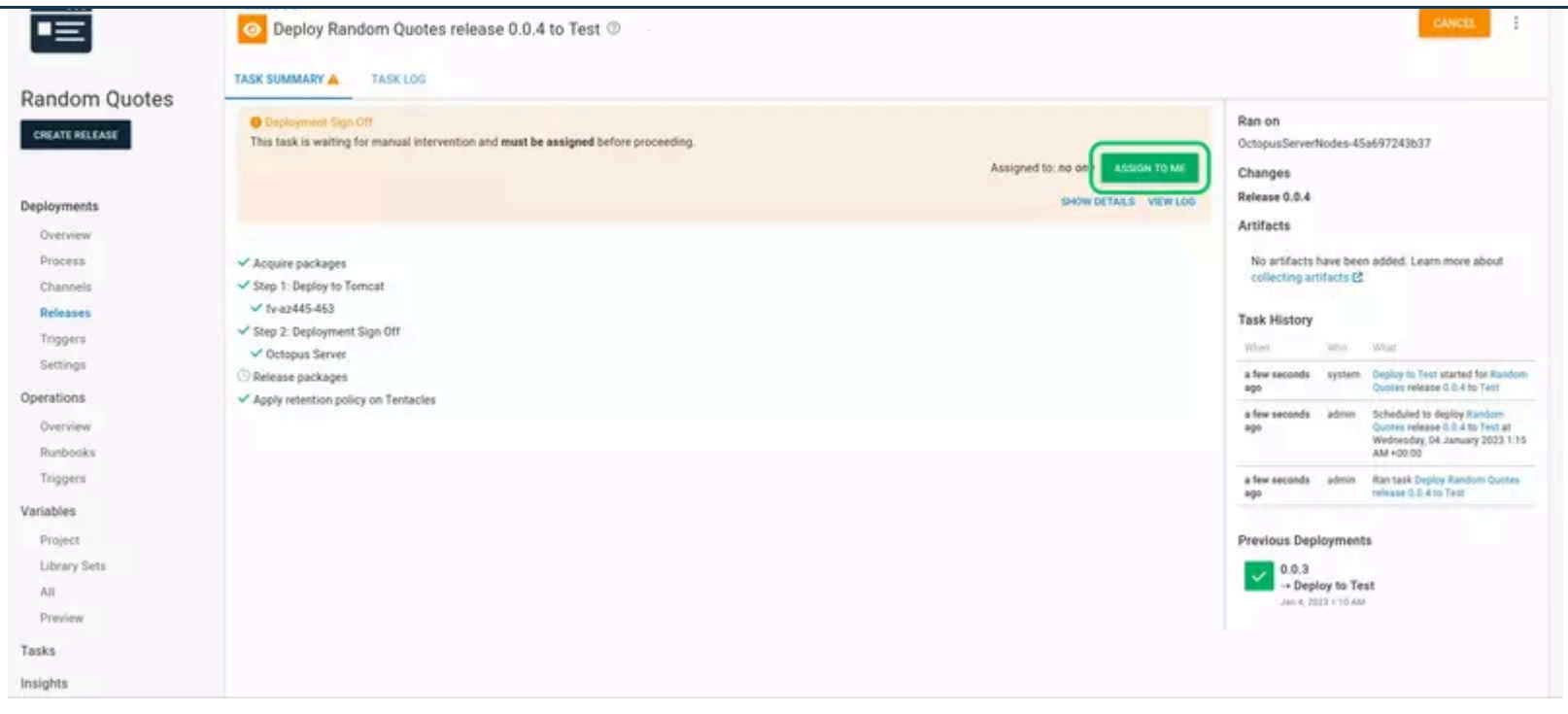
Expand the **Environments** section under the **Conditions** heading, select the **Skip specific environments** option, and select the **Dev** environment.

Click **SAVE** to save the step:

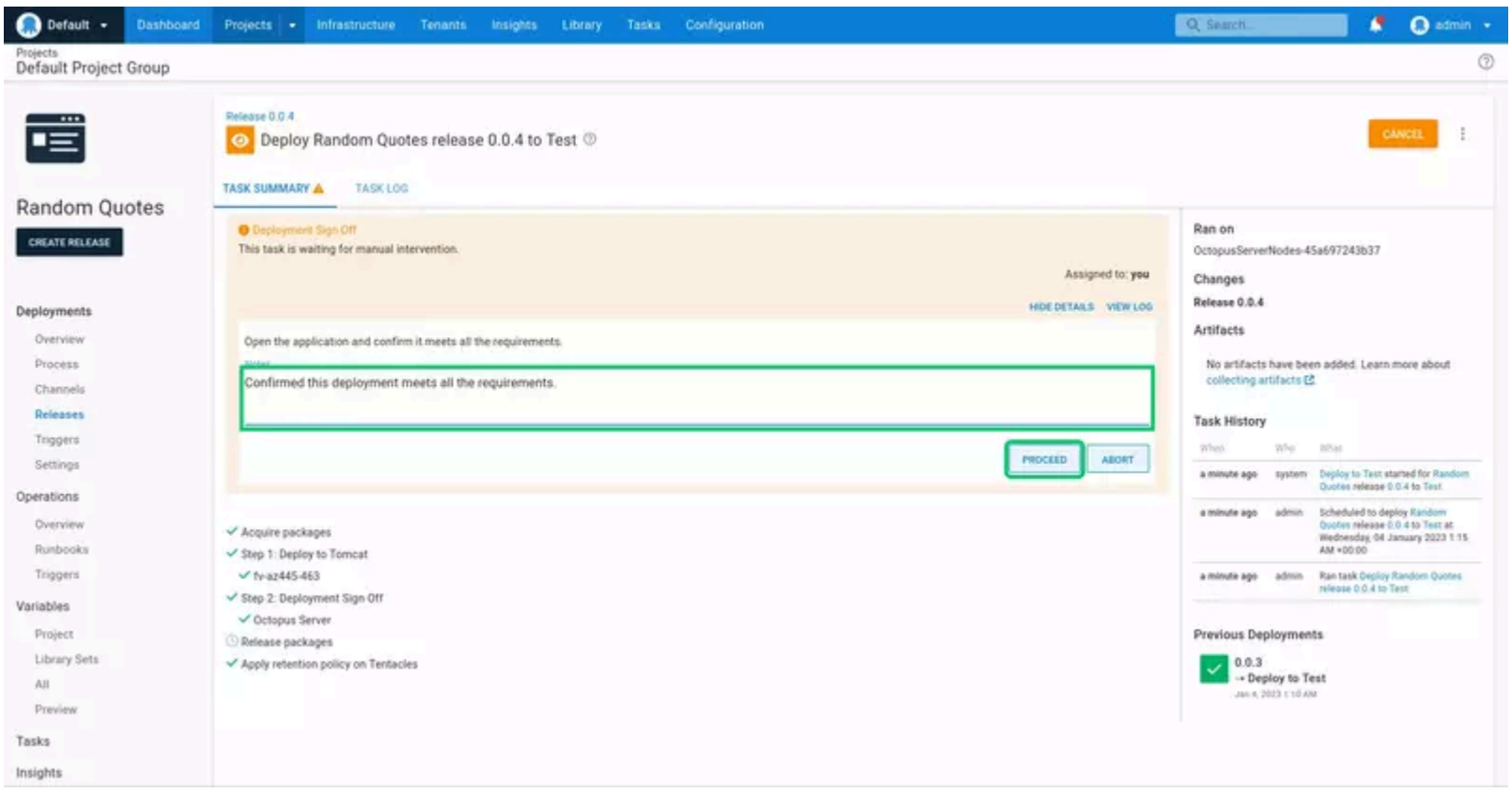


When this application is deployed to the **Test** or **Prod** environments, a prompt will be displayed requiring manual sign off. Click **ASSIGN TO ME** to assign the task to yourself:

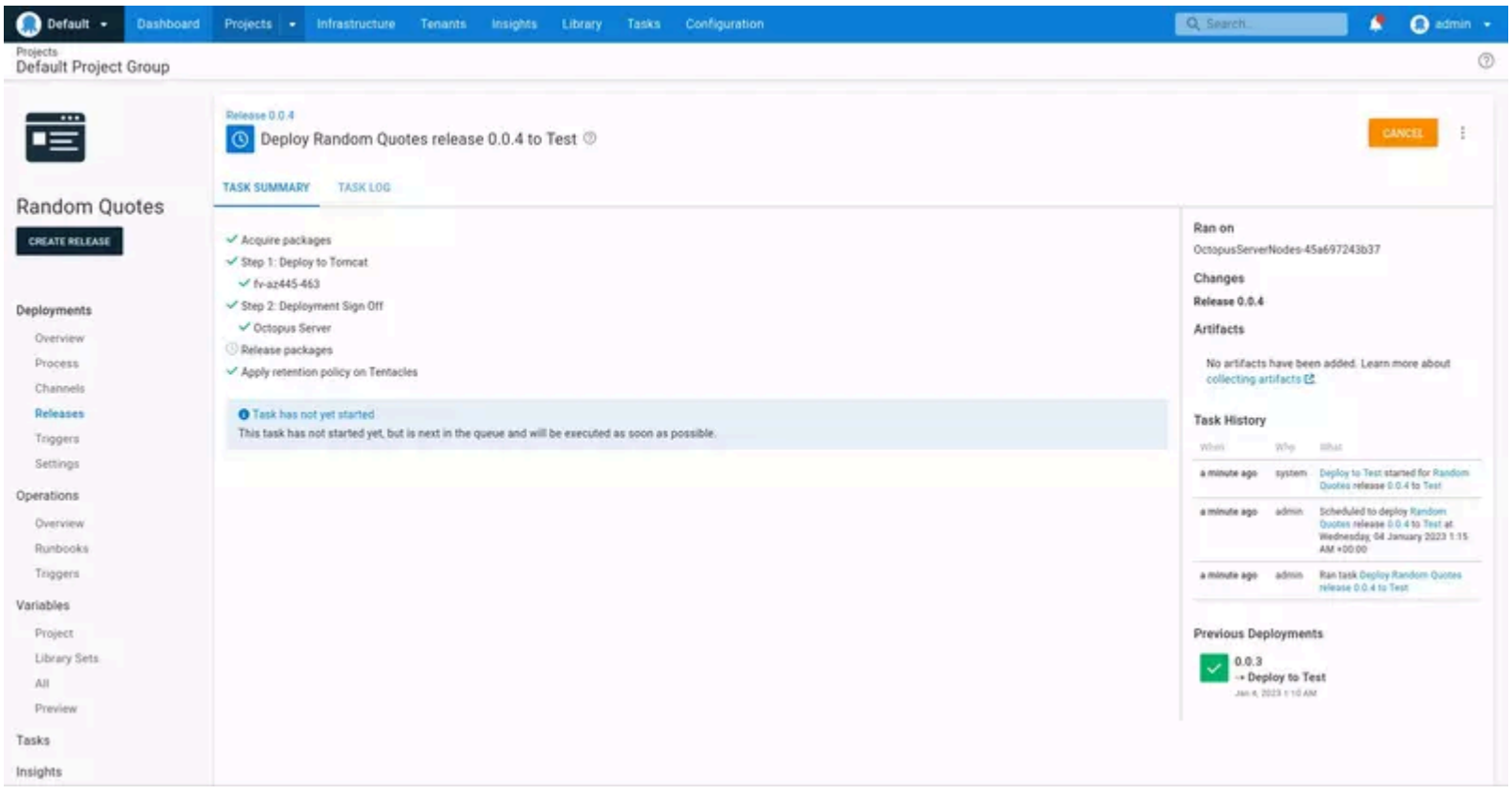
Octopus Deploy



Add a note in the provided text box, and click **PROCEED** to complete the deployment:



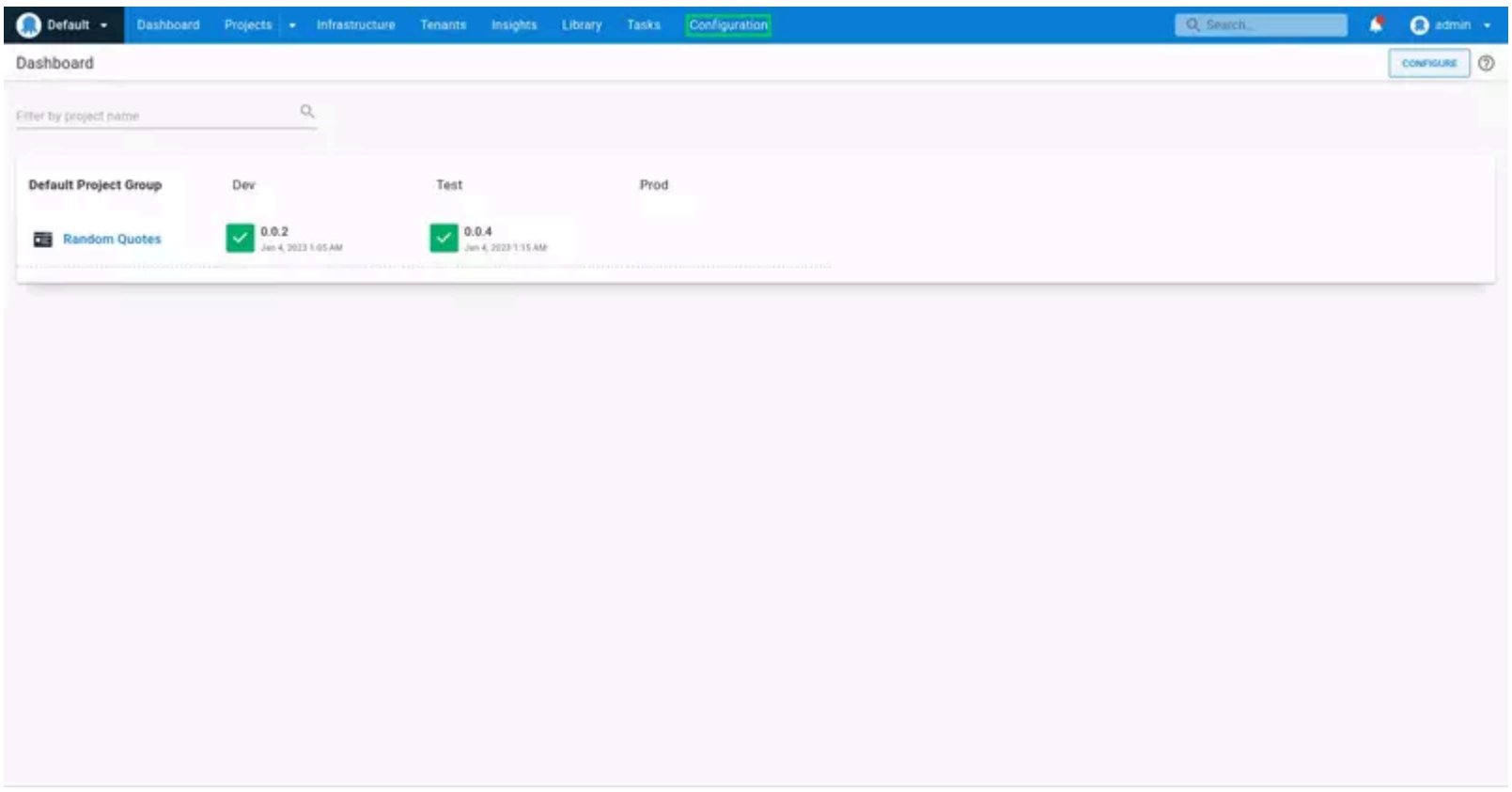
The deployment will then complete successfully:



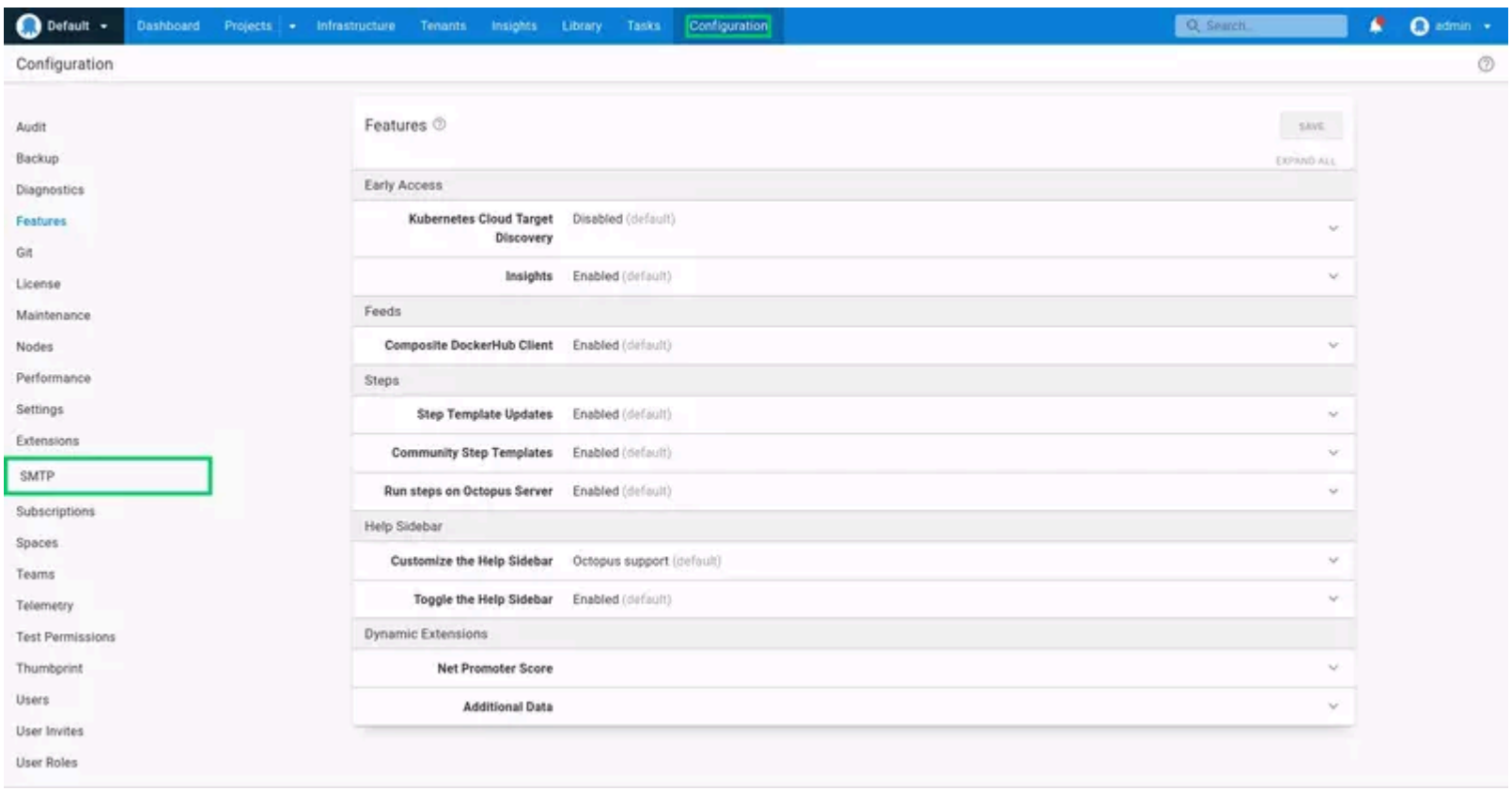
process to let people know when a release has been deployed to an environment.

To start, we need to configure an SMTP server to send our emails. For this guide, we'll use the free SMTP server provided by Google.

Click the **Configuration** link:



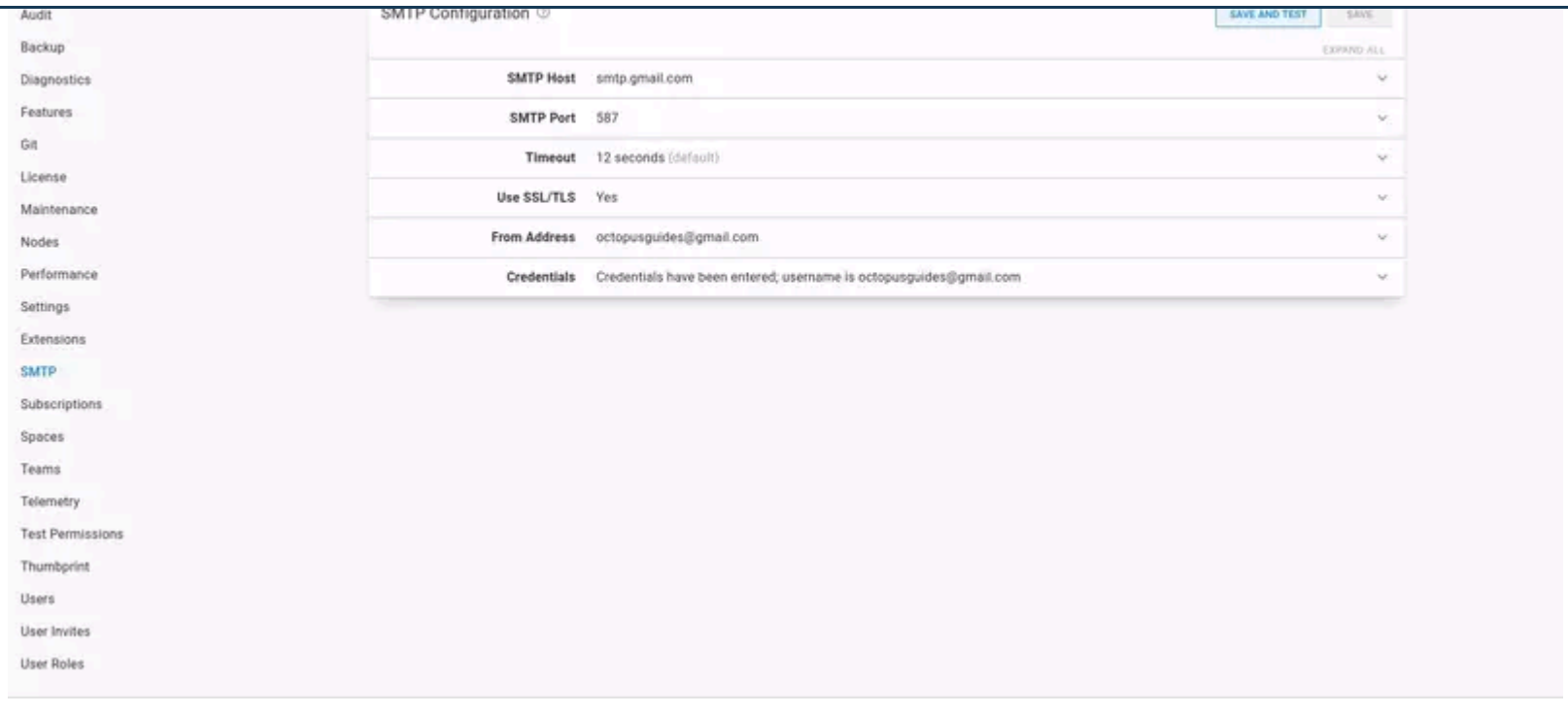
Click the **SMTP** link:



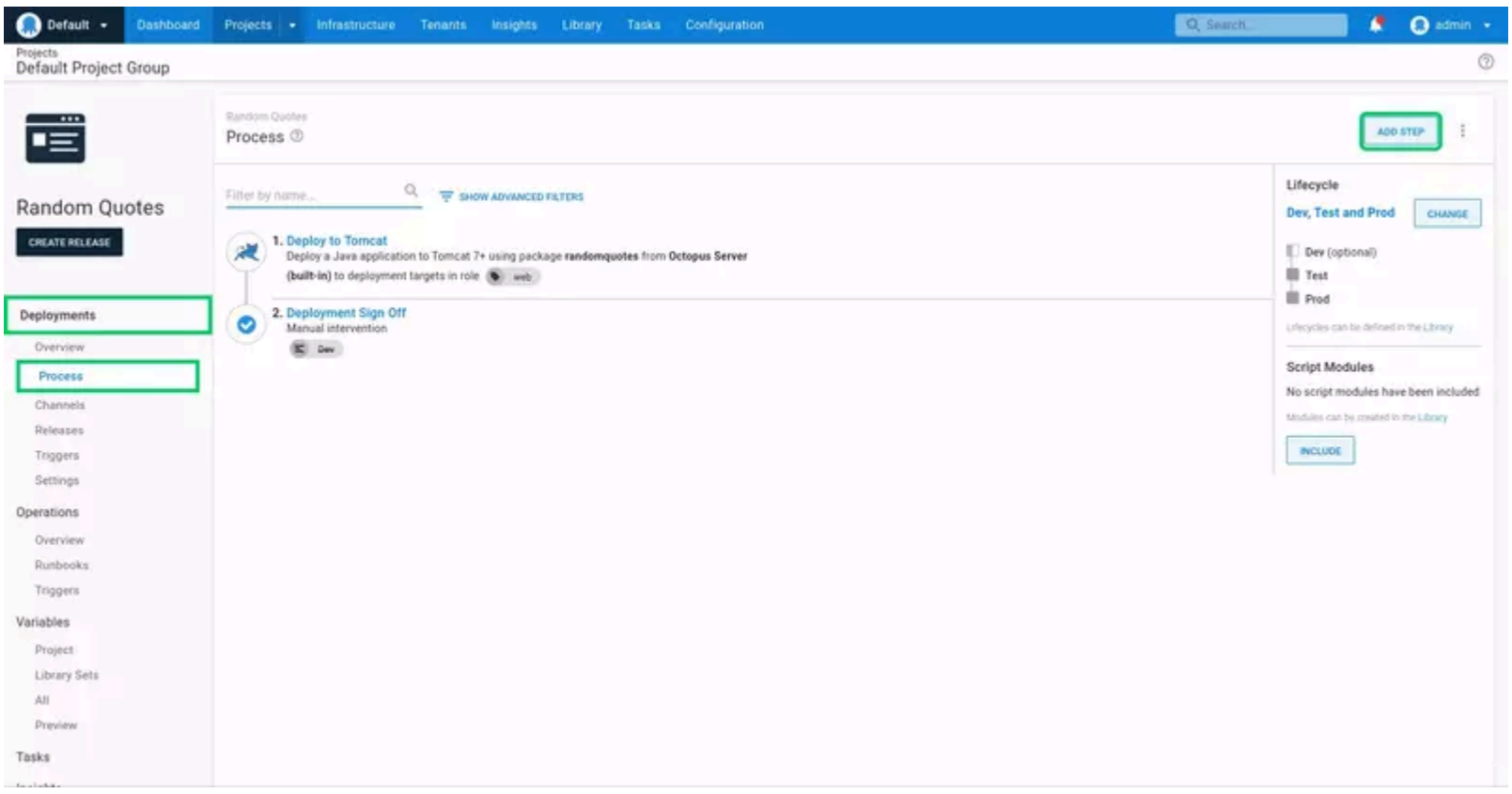
- Enter **smtp.gmail.com** as the **SMTP Host**.
- Enter **587** as the **SMTP Port**.
- Enable the **Use SSL/TLS** option.
- Enter your Gmail address as the **From Address**.
- Enter your Gmail address and password in the **credentials**.

> You will enable the **Less secure apps** option on your Google account for Octopus to send emails via the Google SMTP server.

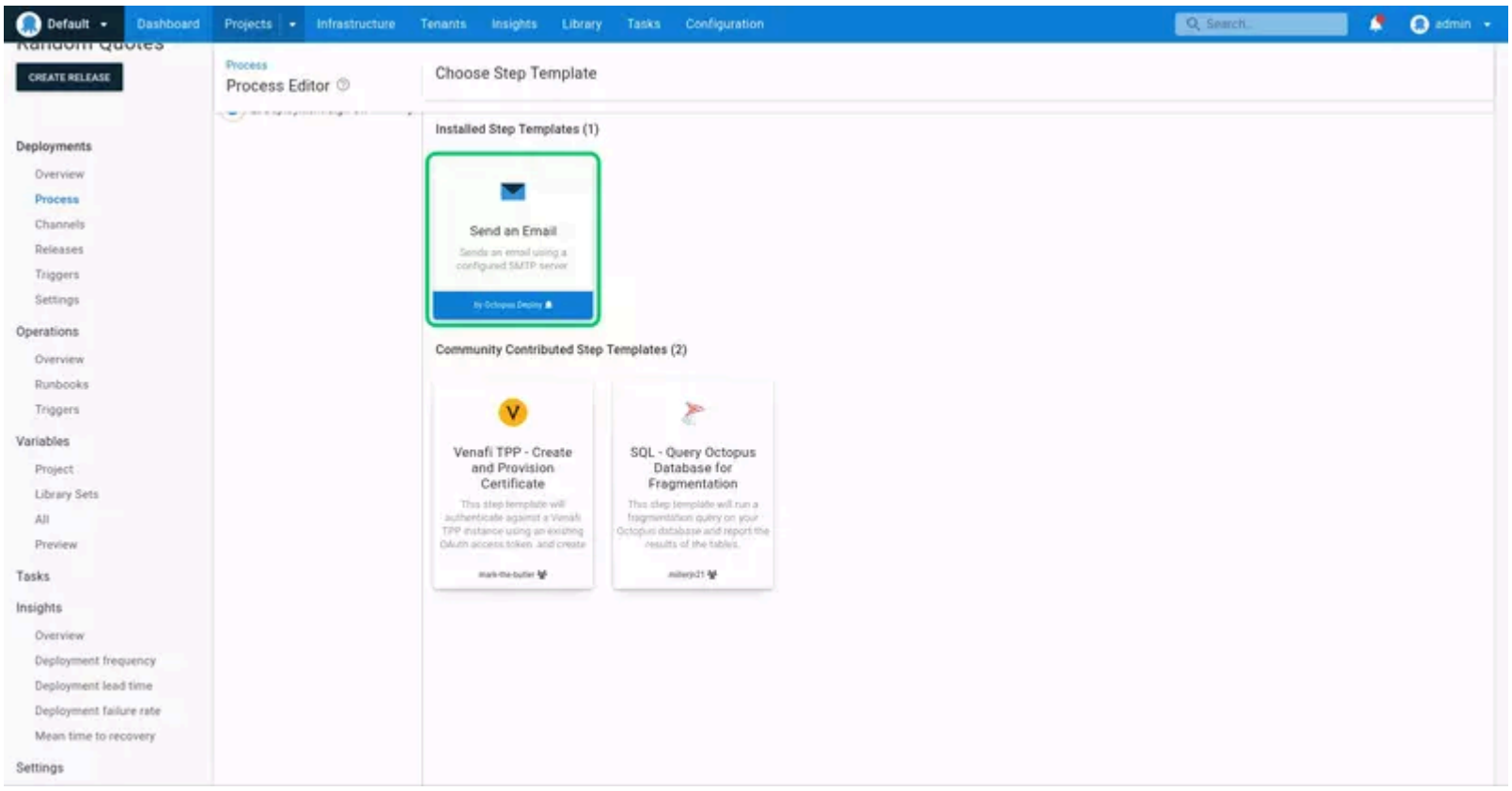
Octopus Deploy



Open the Random Quotes project, click the **Process** link, and click **ADD STEP**:

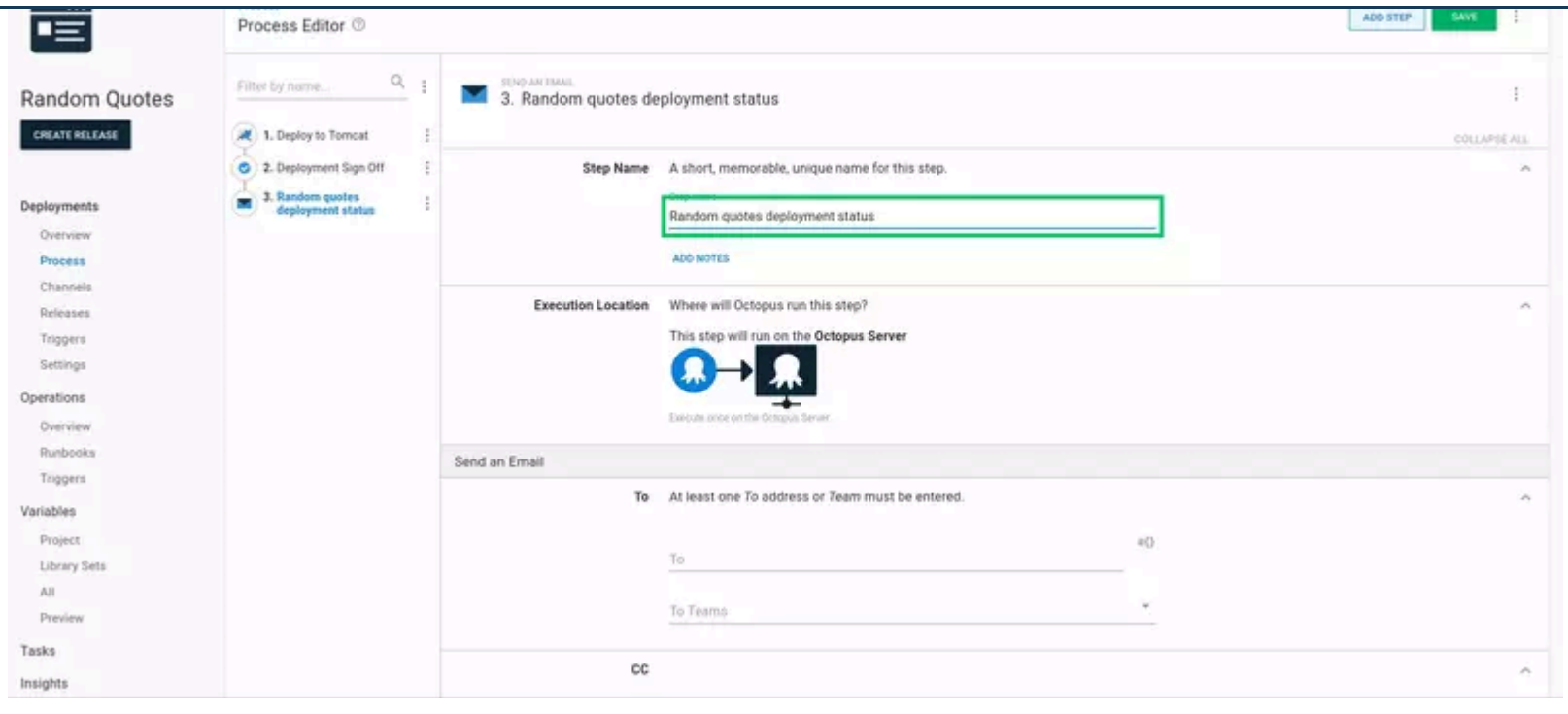


Search for the **Send an Email** step, and add it to the process:

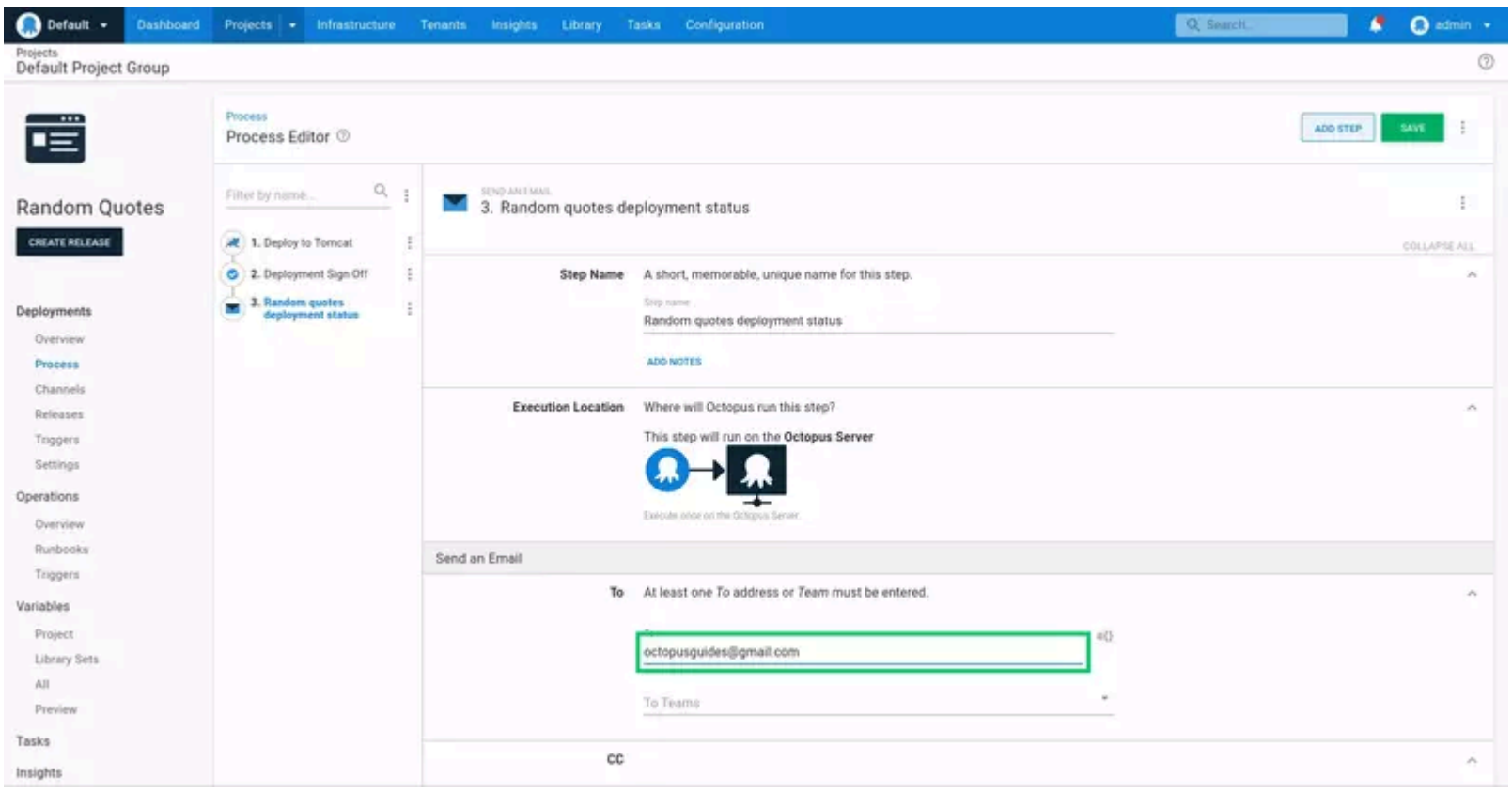


Enter **Random quotes deployment status** for the **Step Name**:

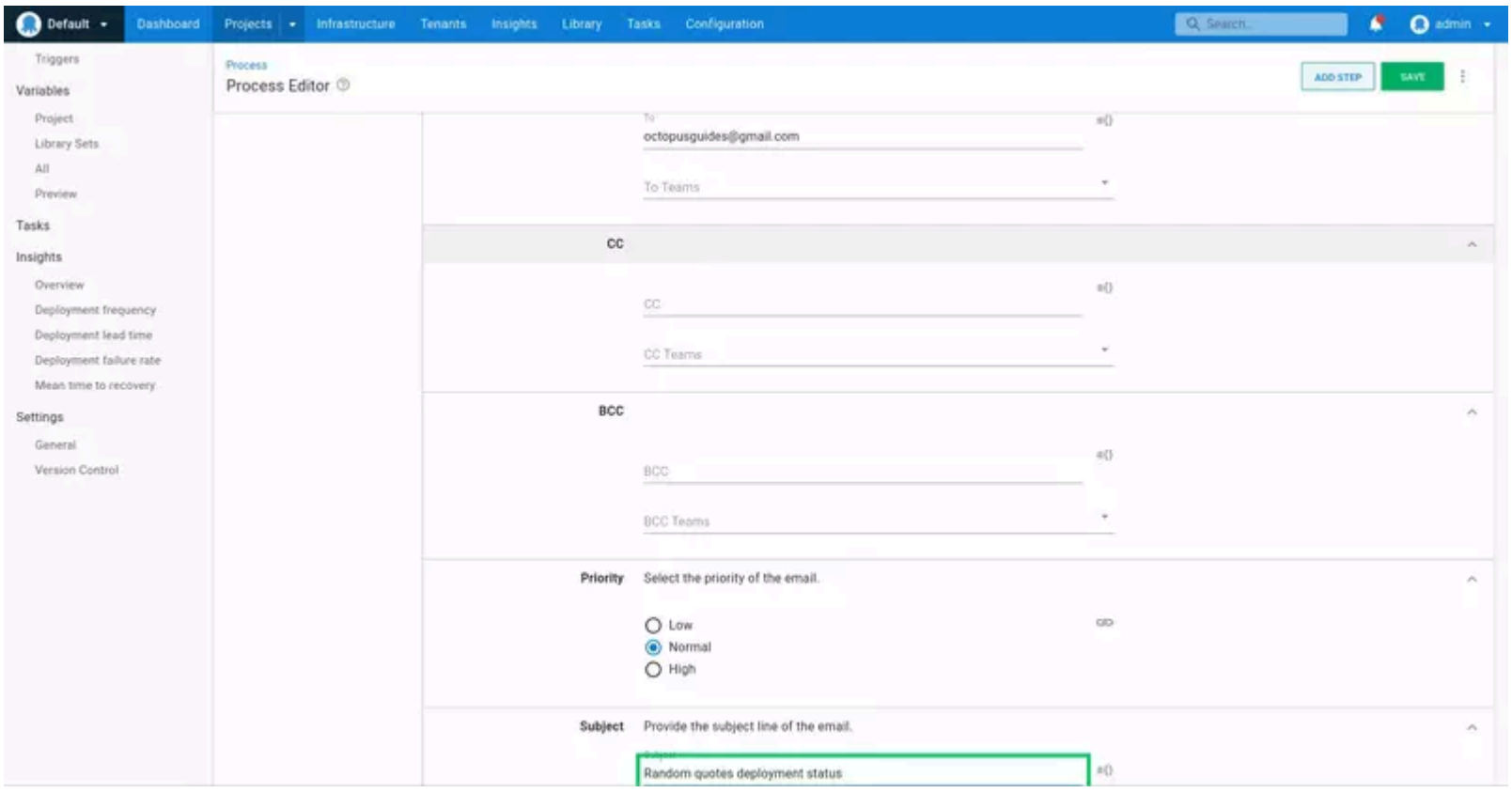
Octopus Deploy



Enter the email address to receive the notification in the **To field**:



Enter **Random quotes deployment status** as the **Subject**:



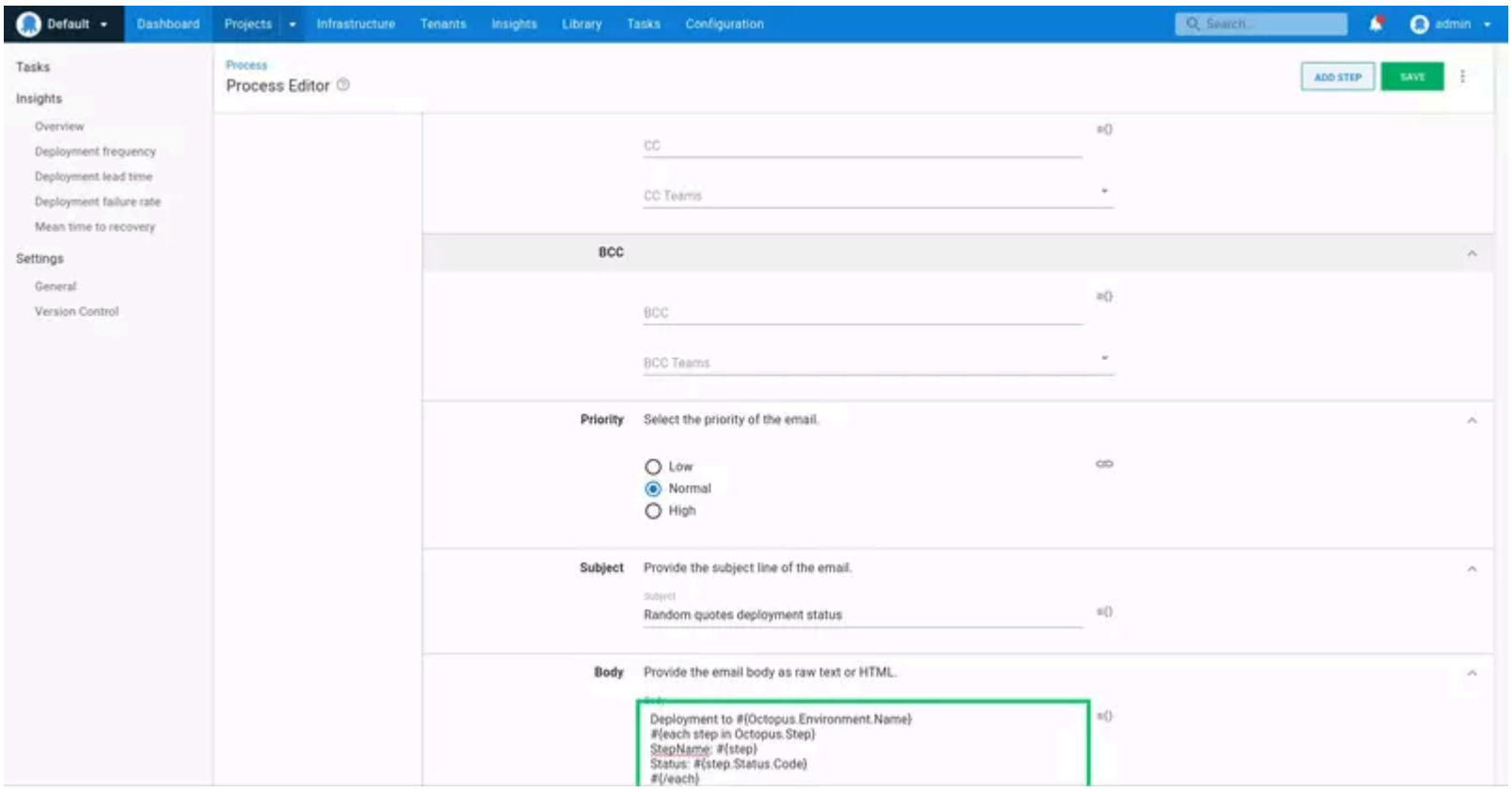
Enter the following as the **Body**:

Octopus Deploy

```
Status: #{step.Status.Code}
#{/each}
```

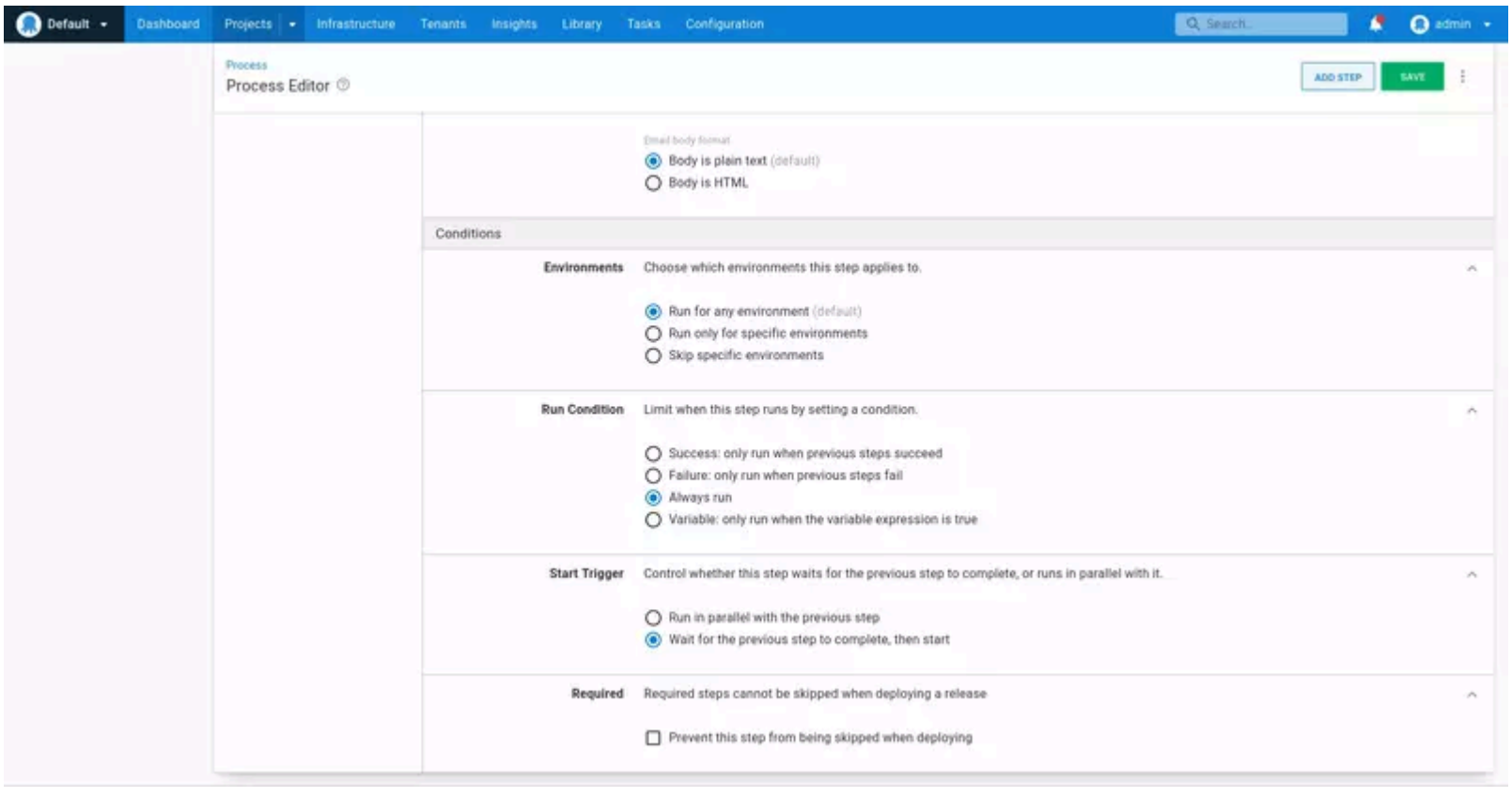
Here we use the **#{Octopus.Environment.Name}** variable provided by Octopus to add the name of the environment that was deployed to, and then loop over the status codes in the **#{Octopus.Step}** variable to return the status of each individual step.

The complete list of system variables can be found in the Octopus documentation.



We want to be notified of the status of this deployment regardless of whether the deployment succeeded or failed. Click the **Run Conditions** section to expand it.

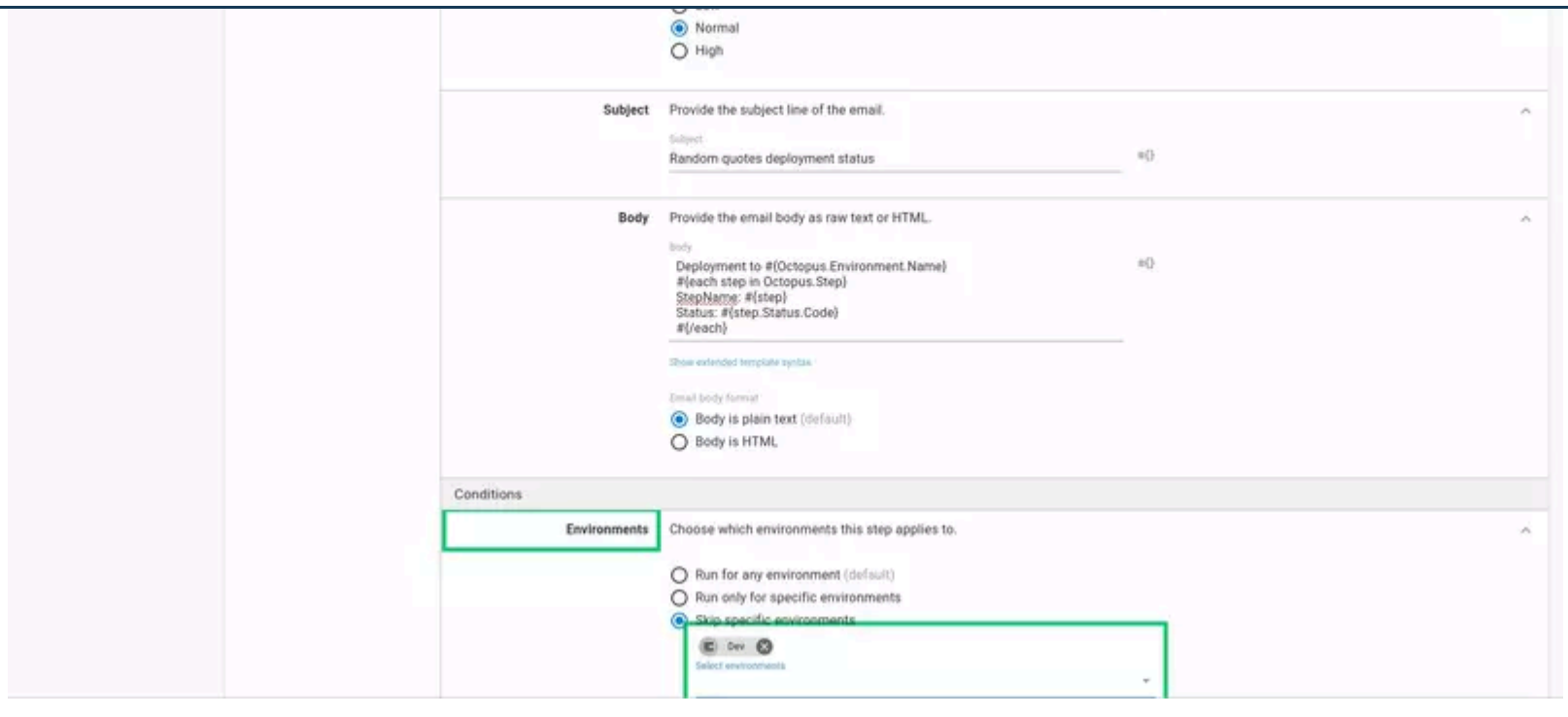
Select the **Always run** option, which ensures the notification email is sent even when the deployment or the manual intervention fail:



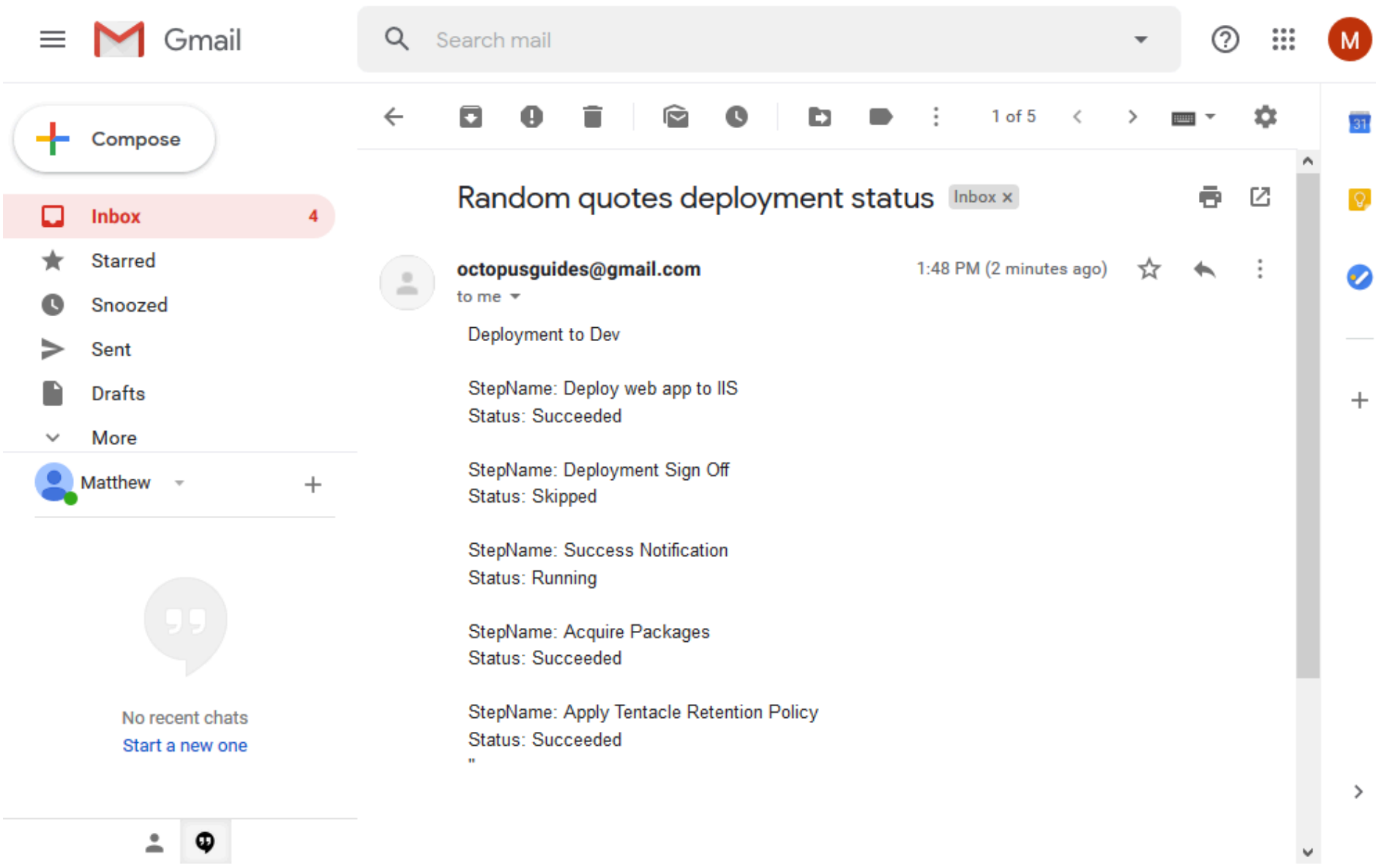
Given every change to the source code will result in a deployment to the **Dev** environment, we do not want to generate reports for deployments to this environment.

Click the **Environments** section to expand it. Select the **Skip specific environments** option, and select the **Dev** environment to skip it.

This is the last piece of configuration for the step, so click **SAVE** to save the changes:



Deploy the project to the Test or Prod environments. When the deployment succeeds, the notification email will be sent:

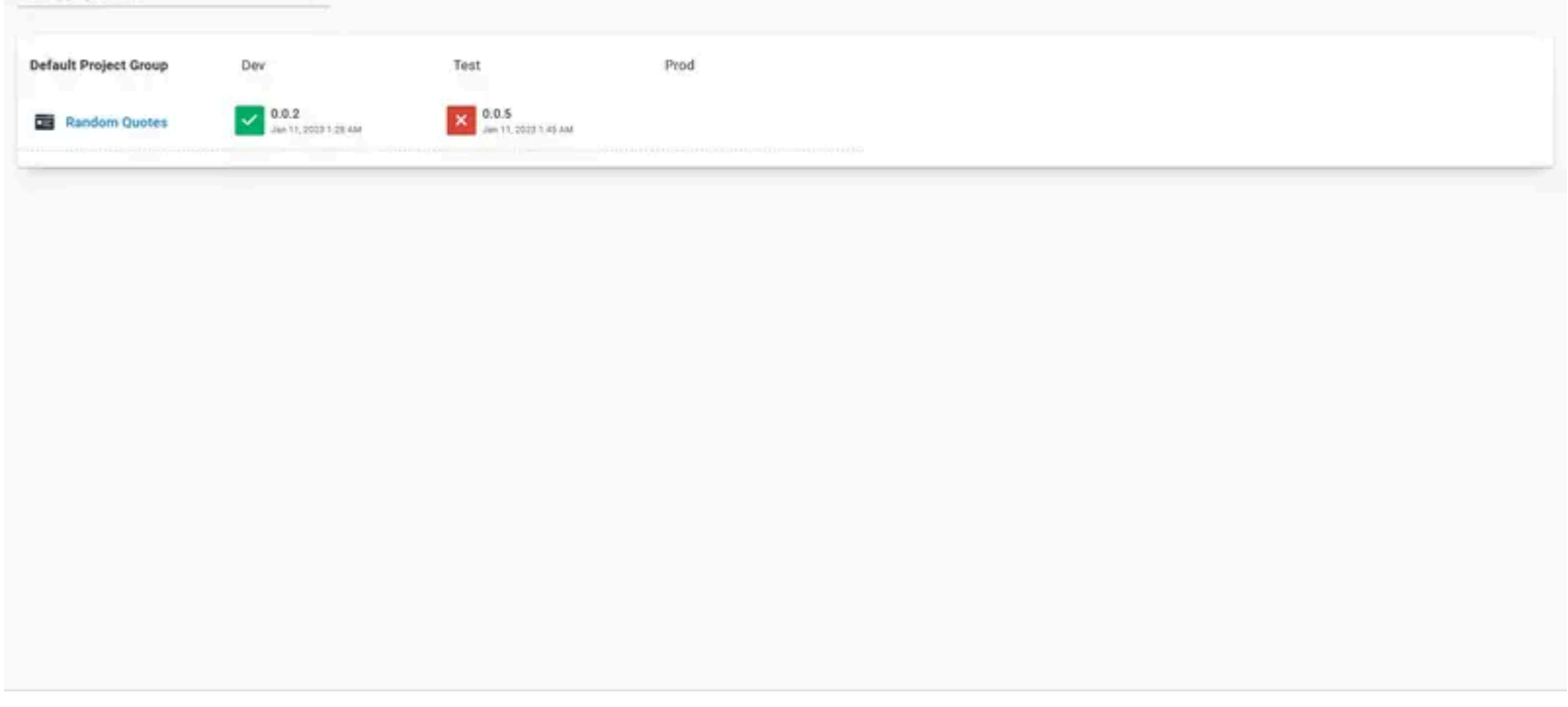


Permissions

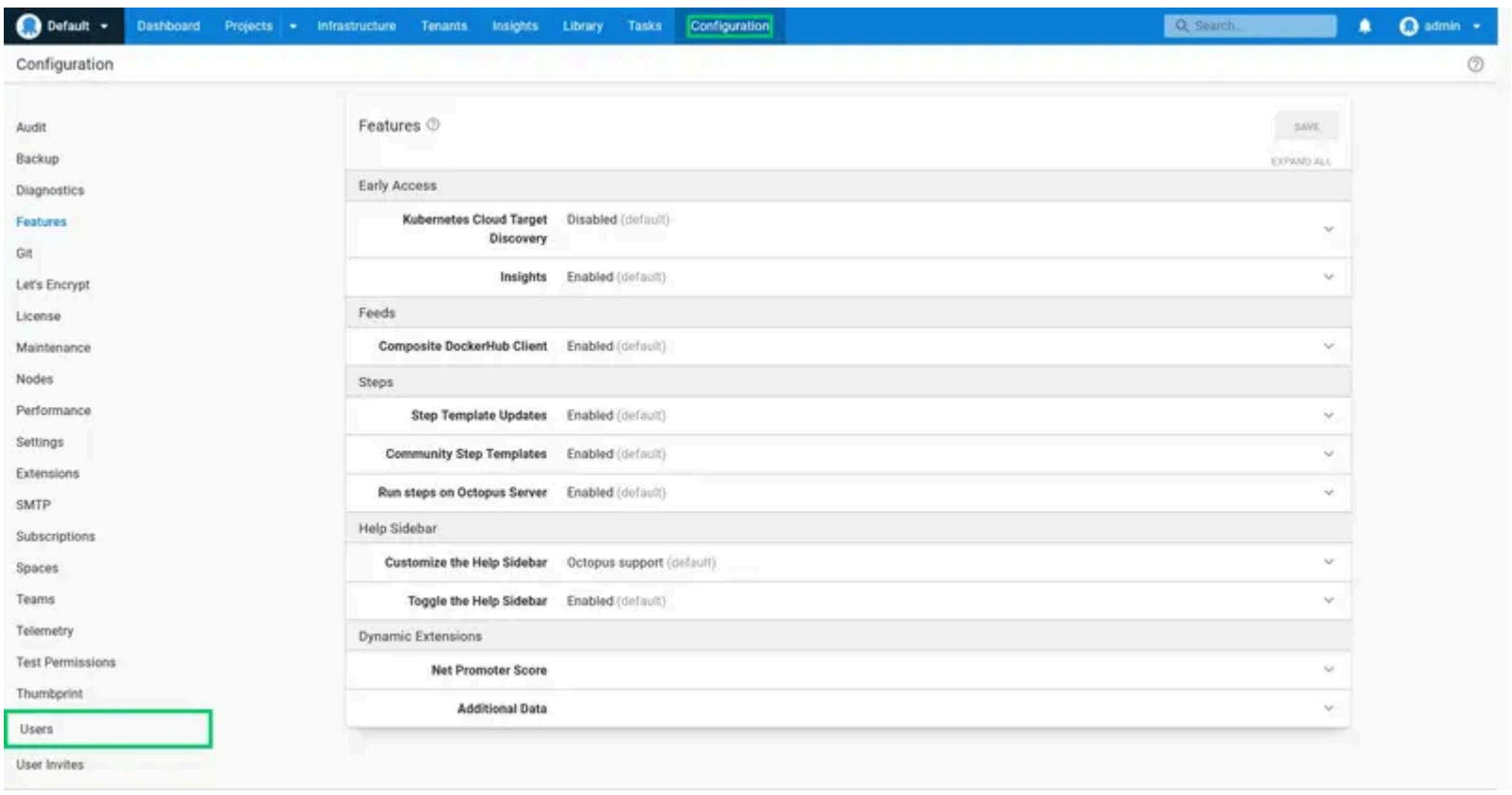
One of the strengths of Octopus is that it models environments as first-class entities. This means the security layer can apply rules granting access only to specific environments. We'll take advantage of this ability to model and secure environments to create two users, an internal deployer who can deploy to the **Dev** and **Test** environments, and a production deployer who can deploy to the **Prod** environment.

We start by creating the users. Click the **Configuration** link:

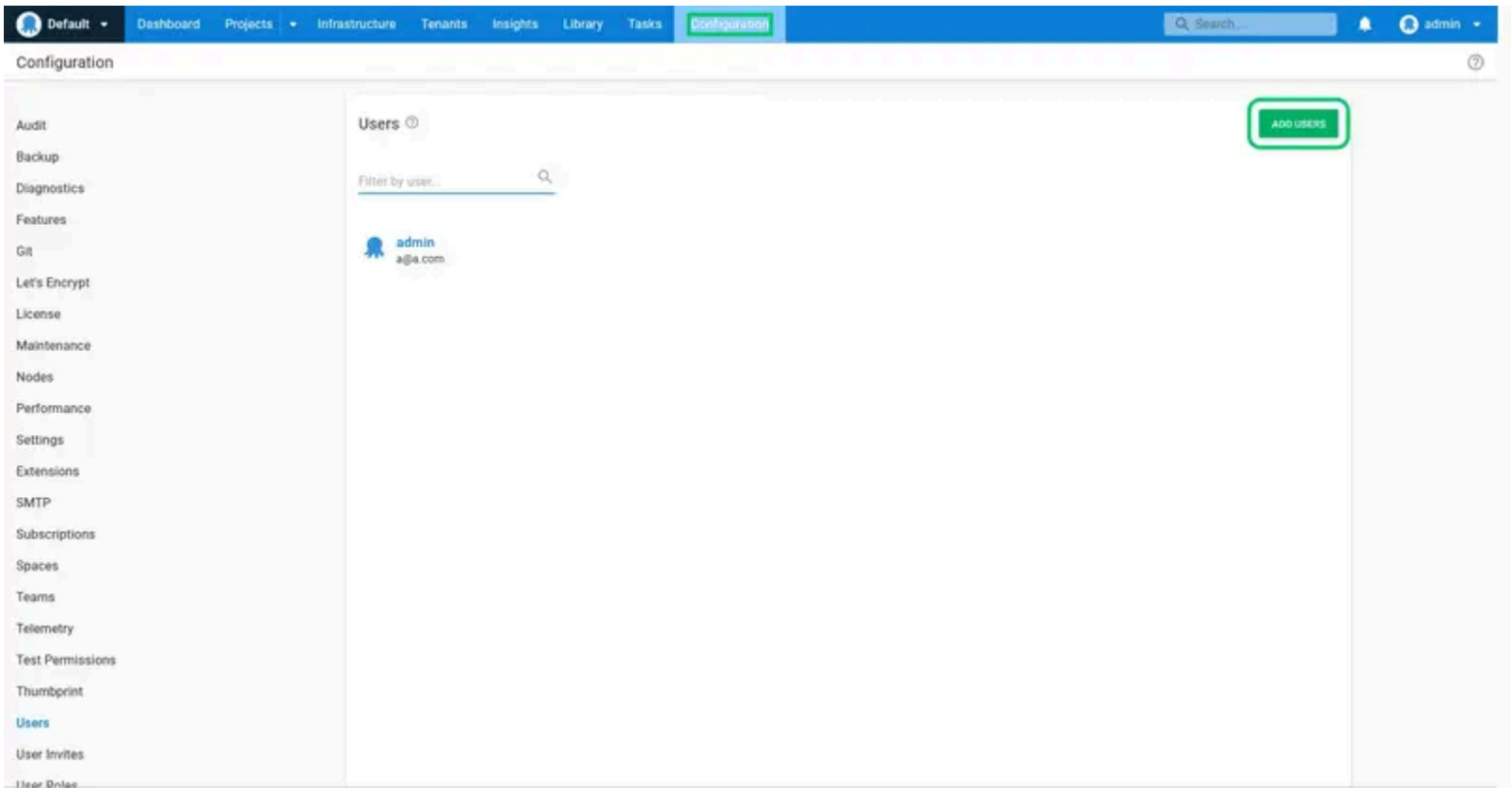
Octopus Deploy



Click the **Users** link:

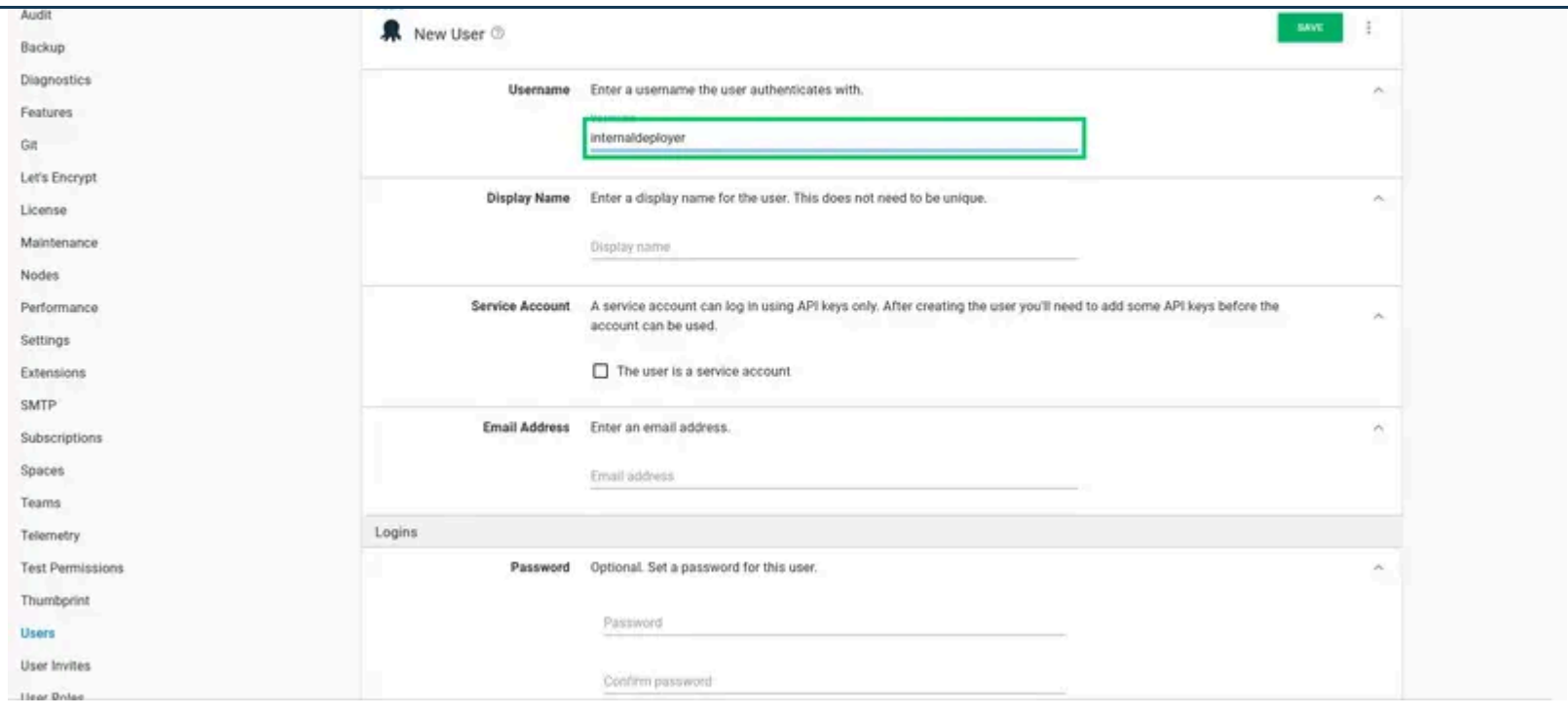


Click **ADD USER**:

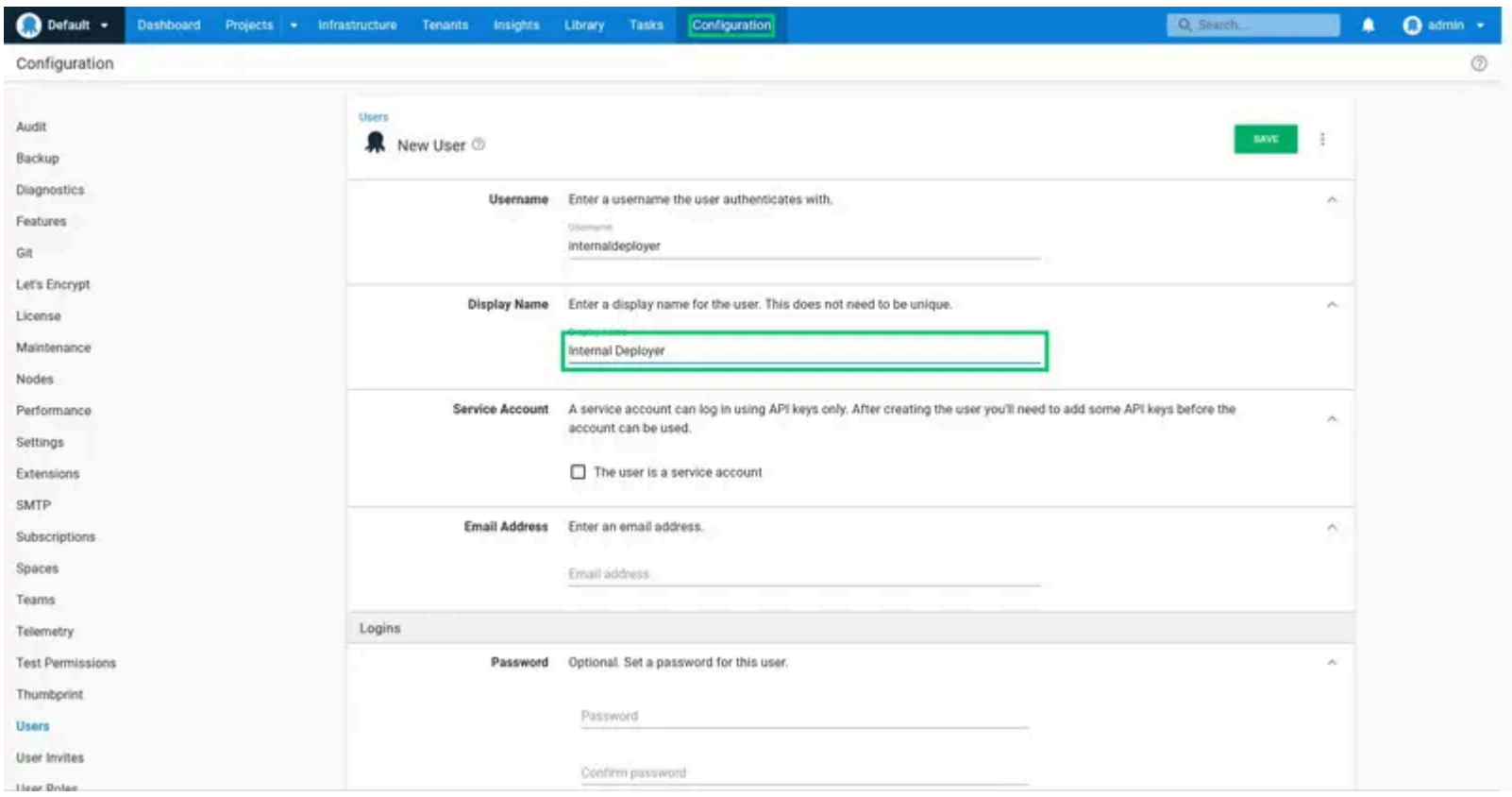


Enter **internaldeployer** as the **Username**:

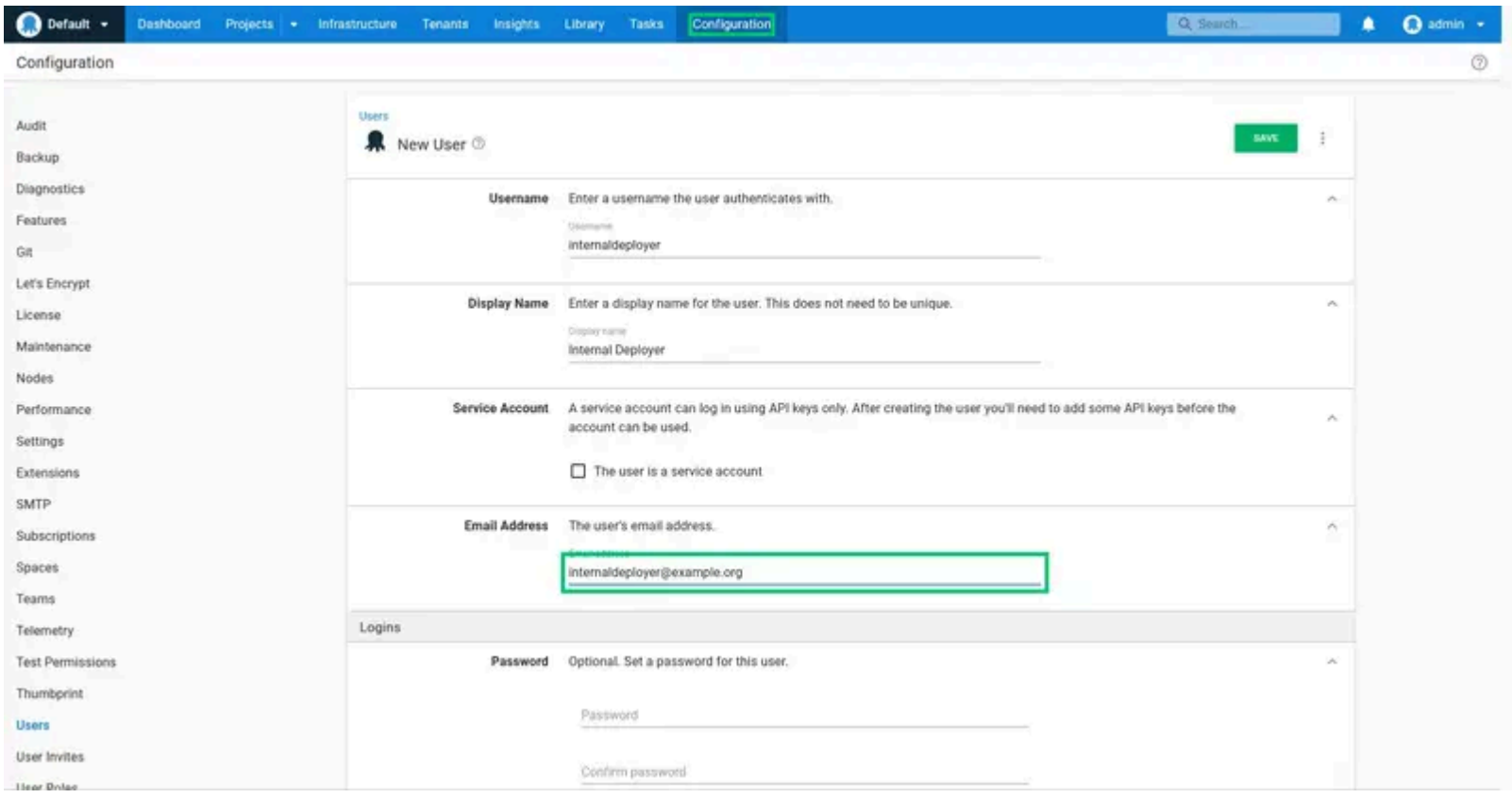
Octopus Deploy



Enter **Internal Deployer** as the **Display Name**:

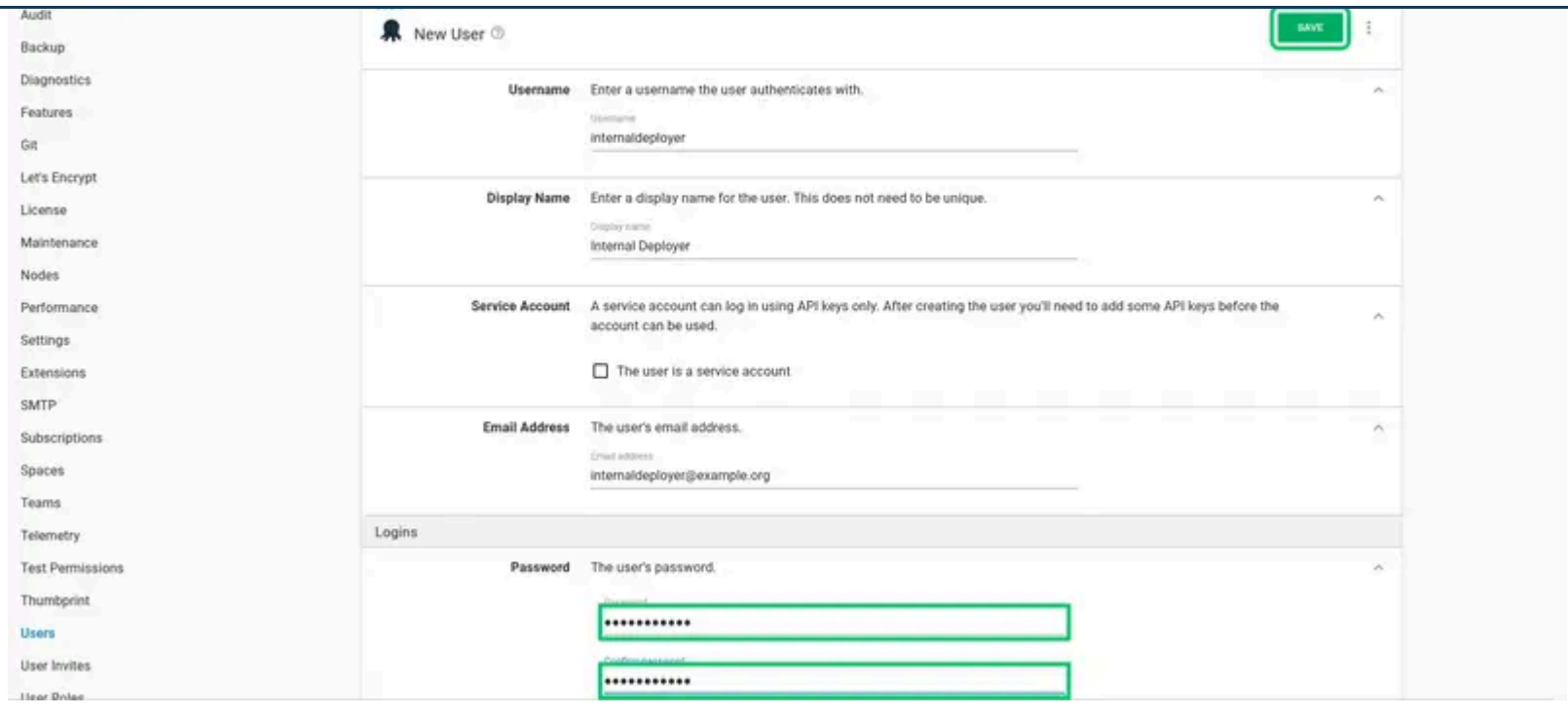


Enter the user's **email address**. We have used a dummy address of **internaldeployer@example.org** here:

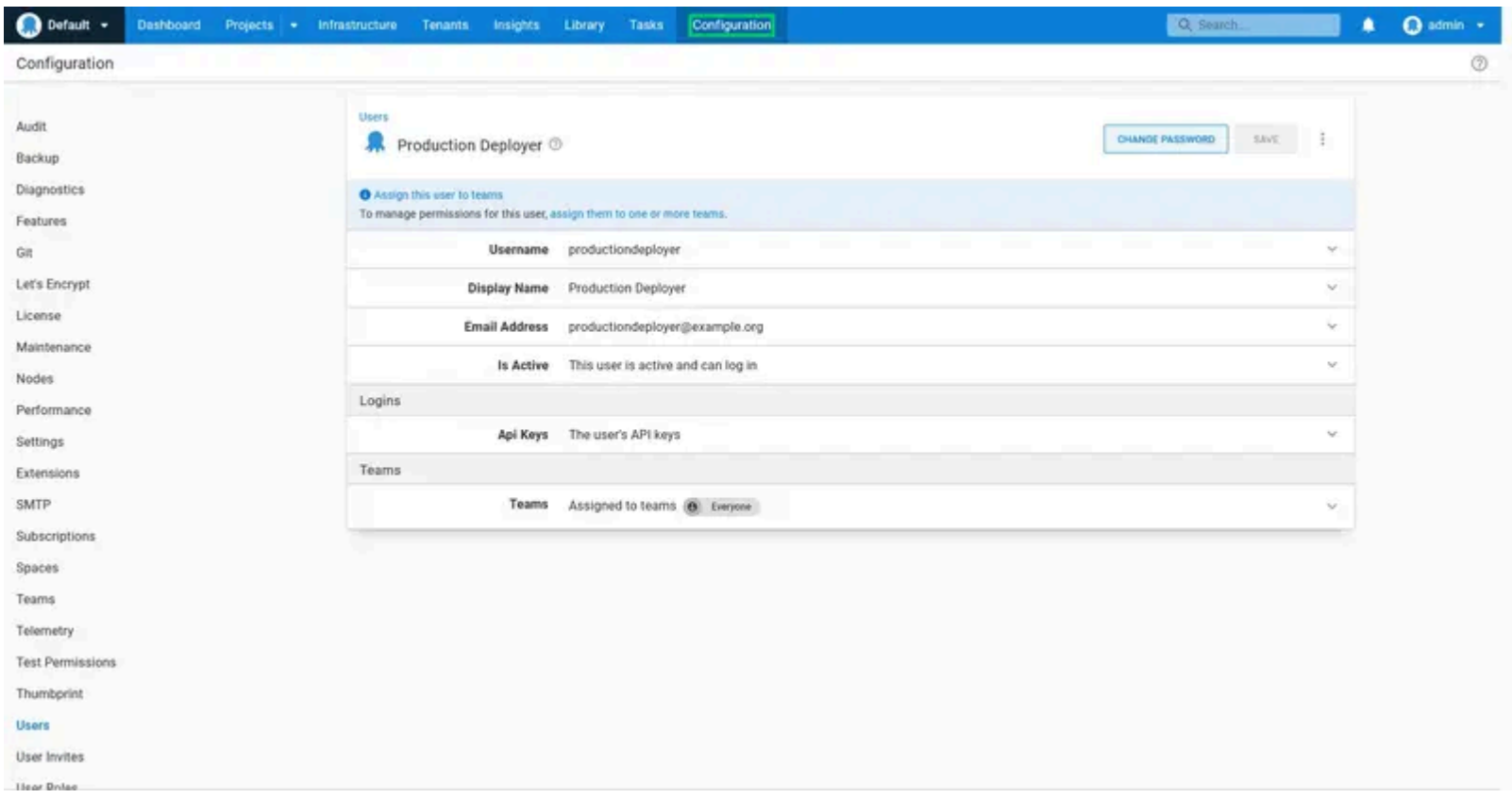


Enter a password and confirm it. Then click **SAVE** to create the user:

Octopus Deploy



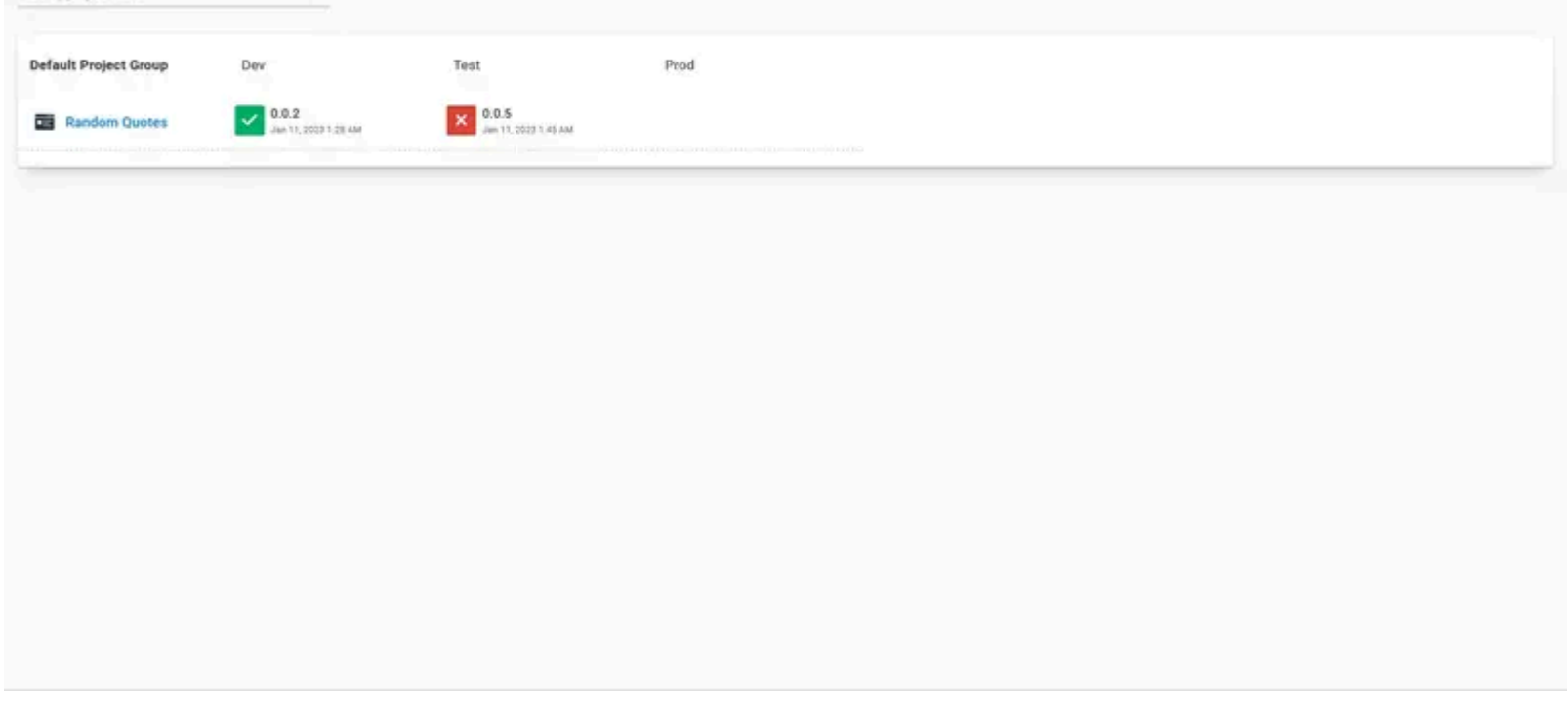
Repeat the process for a user called **productiondeployer**. The summary for the **productiondeployer** user is shown below:



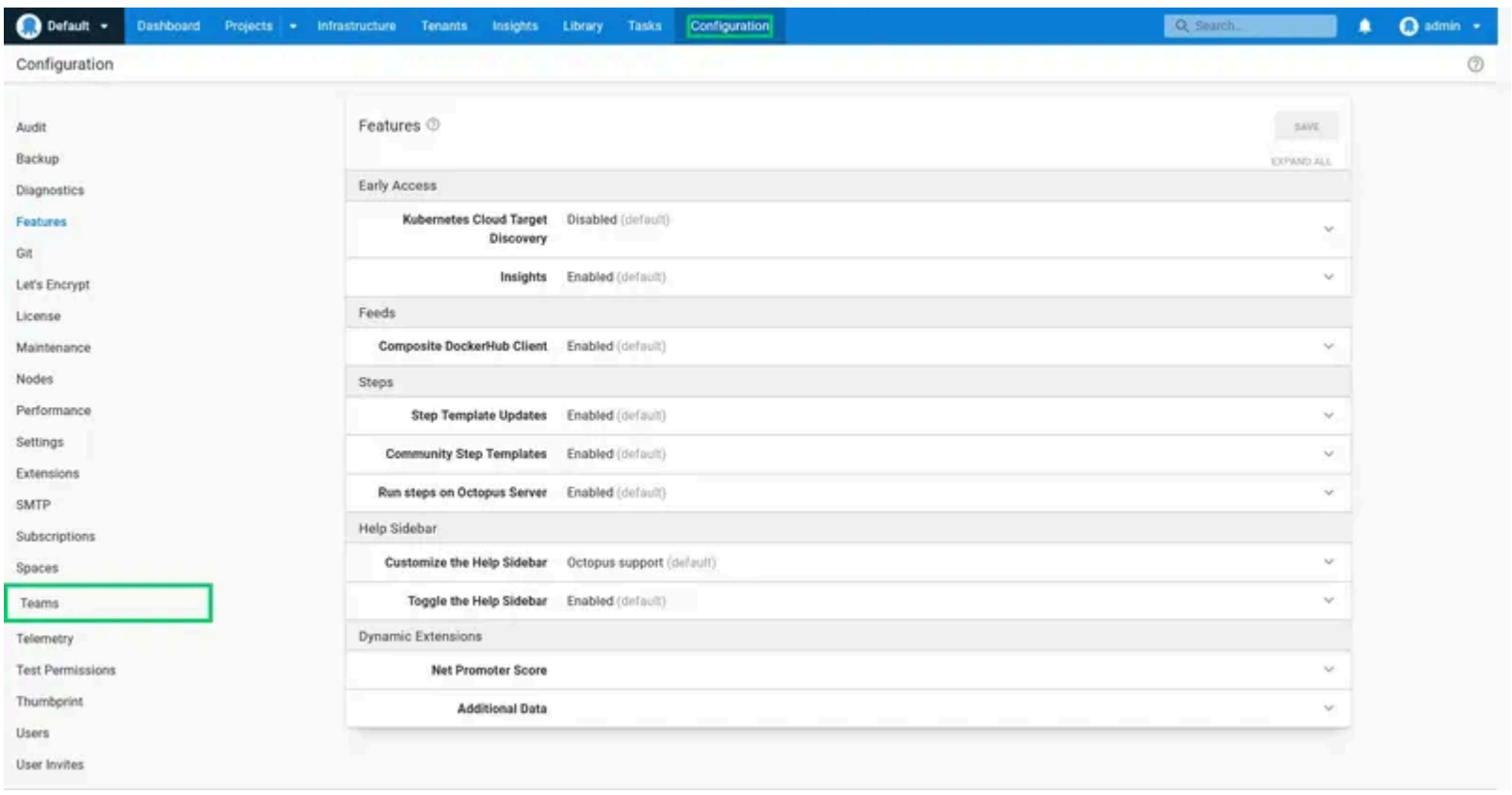
The newly created users are not assigned to any teams and have no permissions to do anything. To grant them permissions, we must first create two teams. The internal deployment team will grant access to deploy to the **Dev** and **Test** environments, while the production deployment team will grant access to deploy to the **Prod** environment.

Click the **Configuration** link:

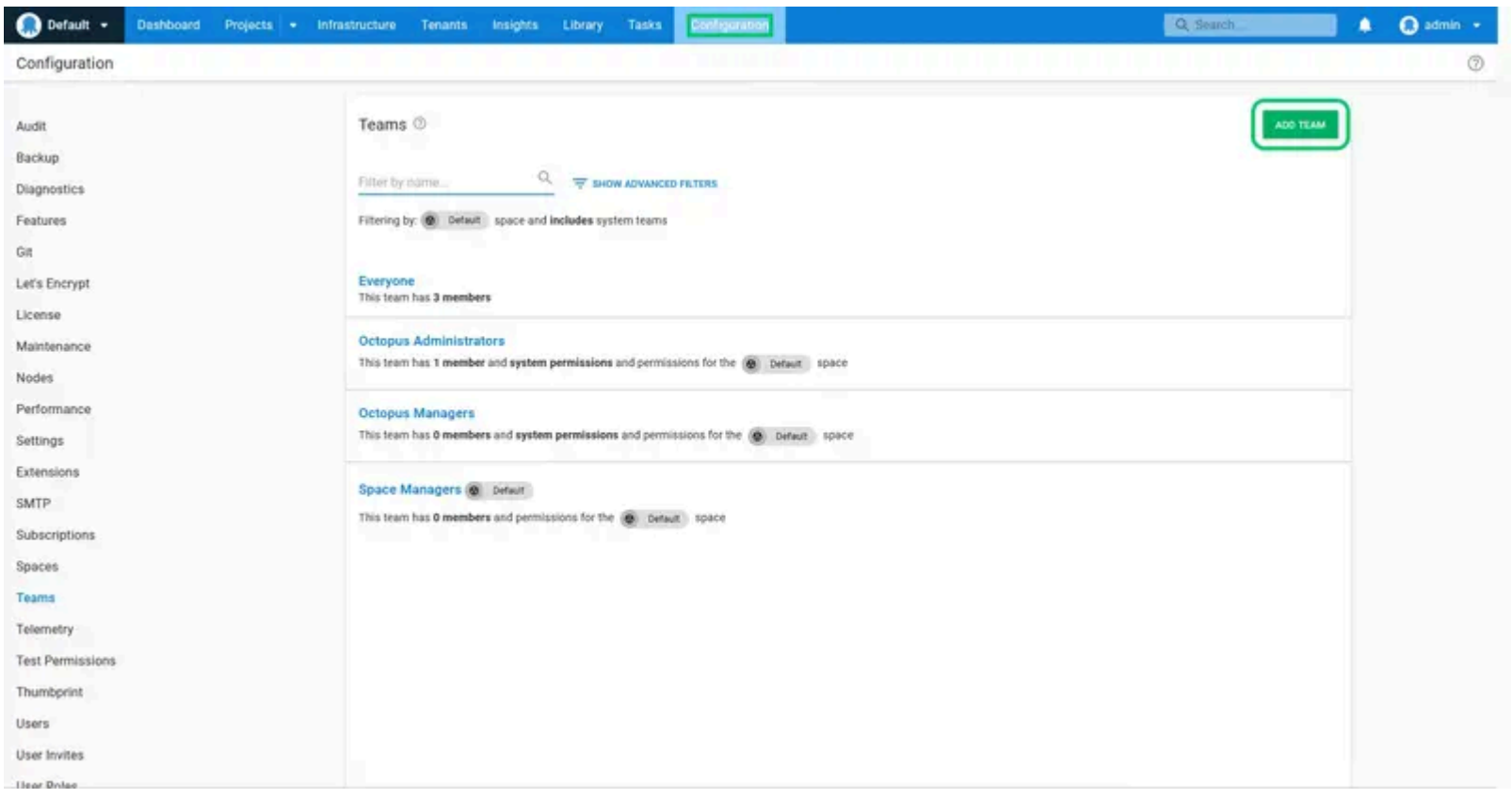
Octopus Deploy



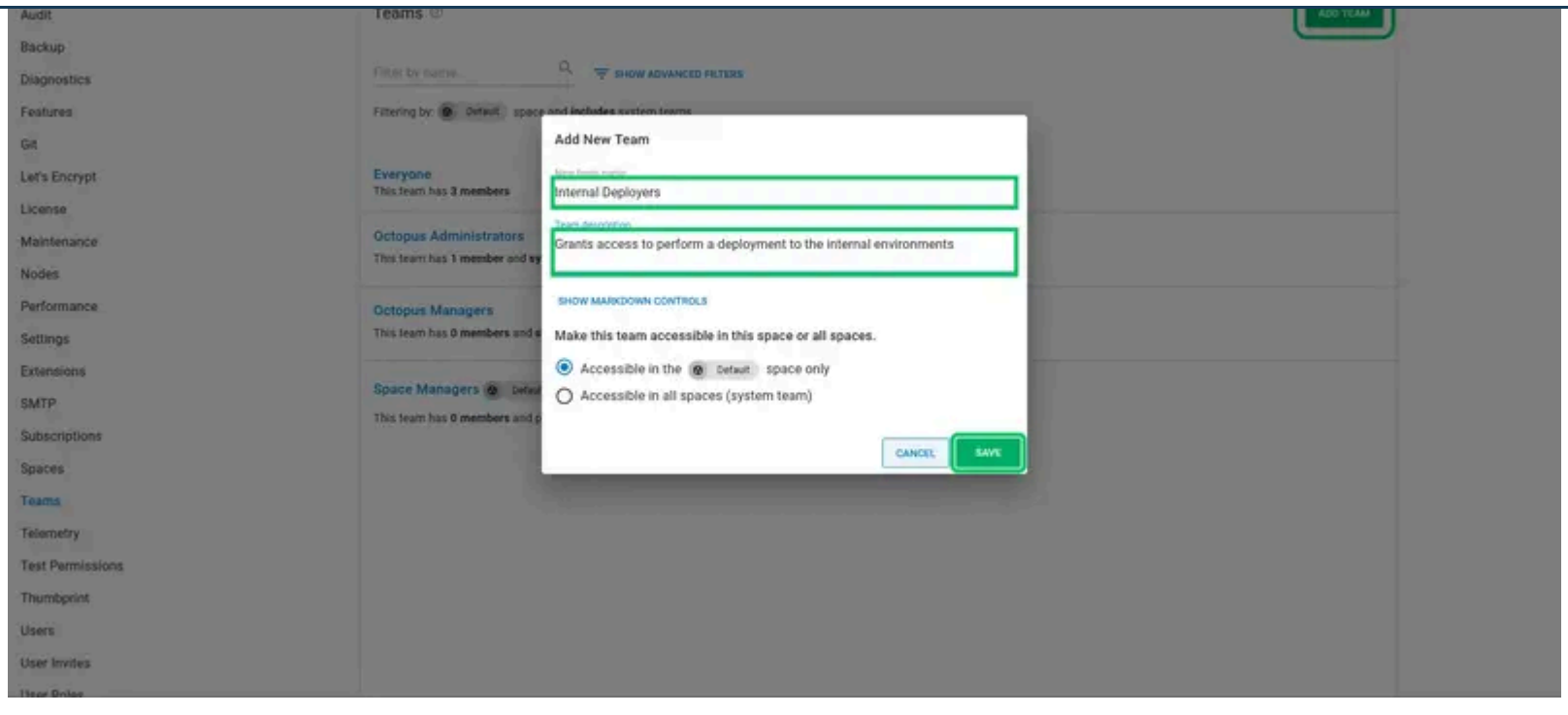
Click the **Teams** link:



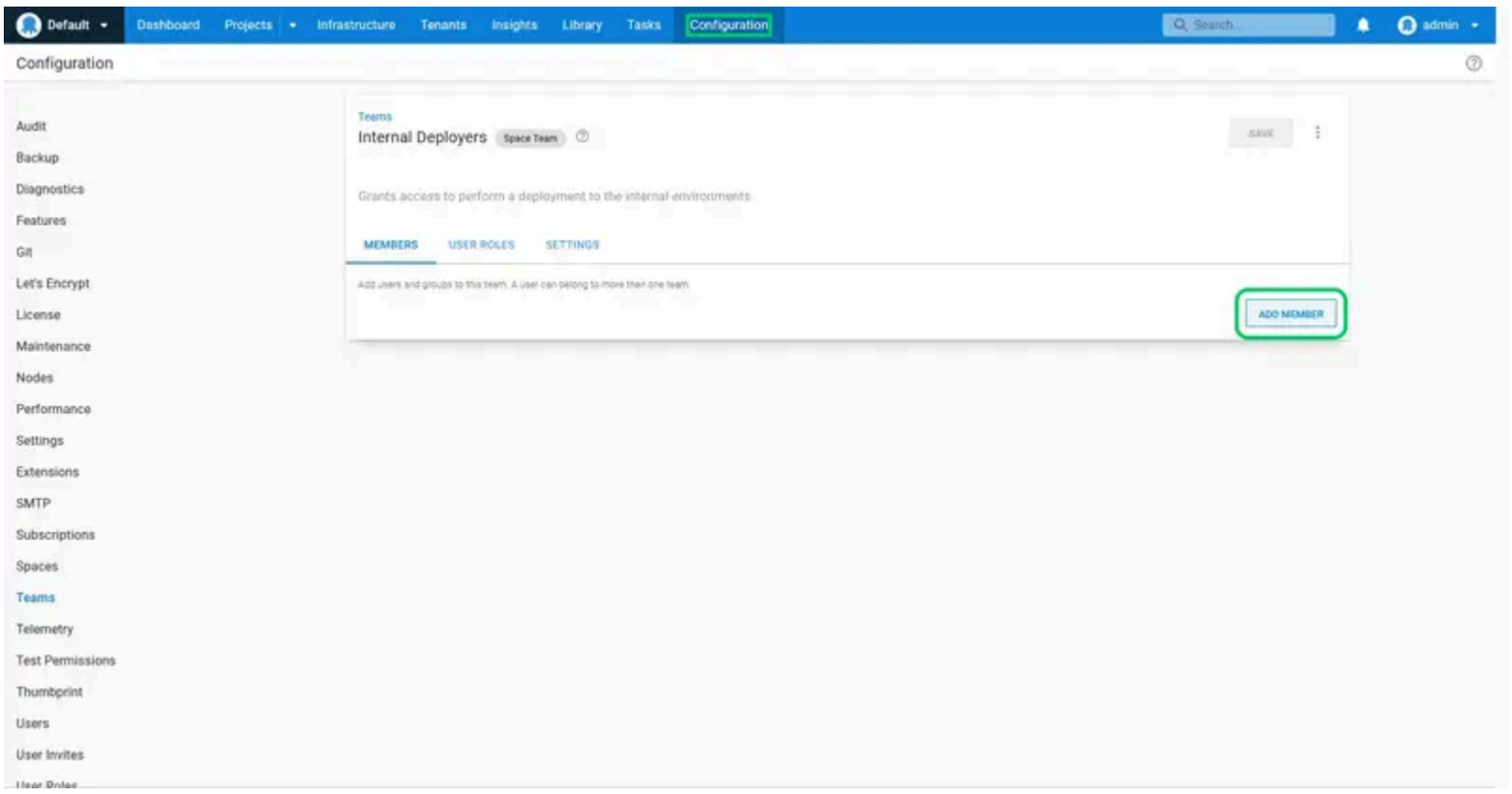
Click **ADD TEAM**:



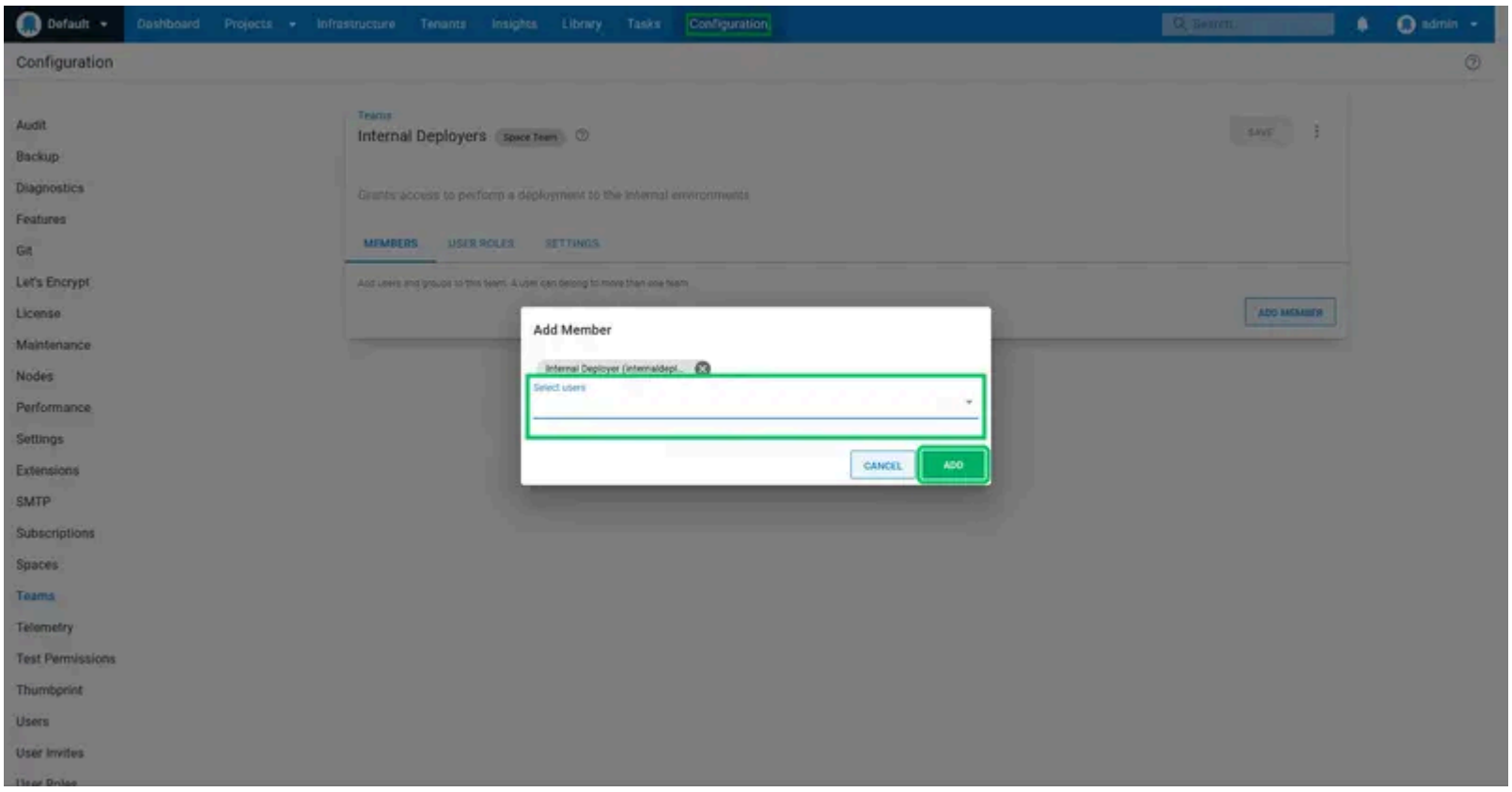
Enter **Internal Deployers** for the **New team name**, and **Grants access to perform a deployment to the internal environments** for the **Team description**. Click **SAVE** to create the team:



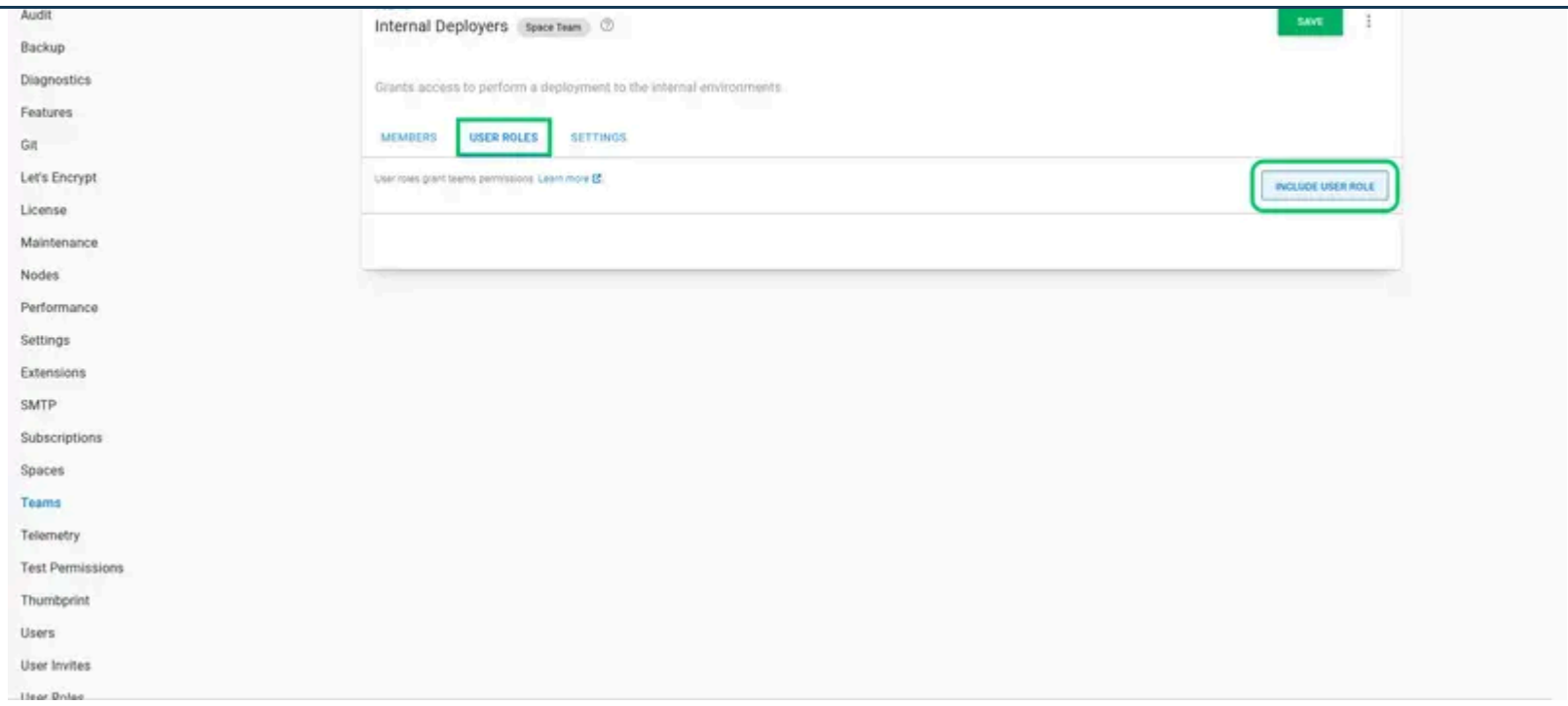
We need to add the **internaldeployer** user to this team. Click **ADD MEMBER**:



Select the **Internal Deployer** user from the dropdown list, and click **ADD**:

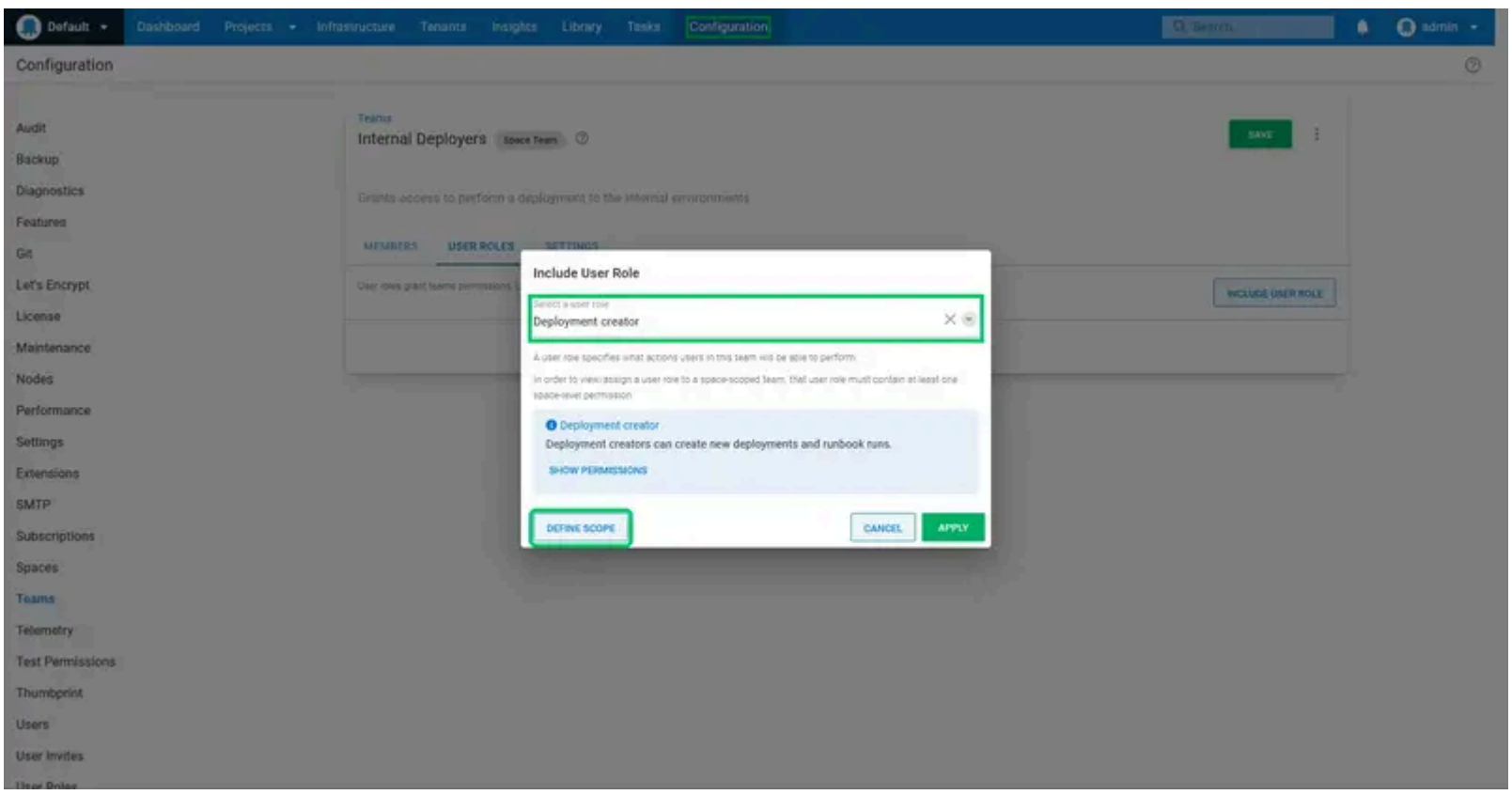


The team does not grant any permissions yet. To add permissions click the **USER ROLES** tab, and click **INCLUDE USER ROLE**:



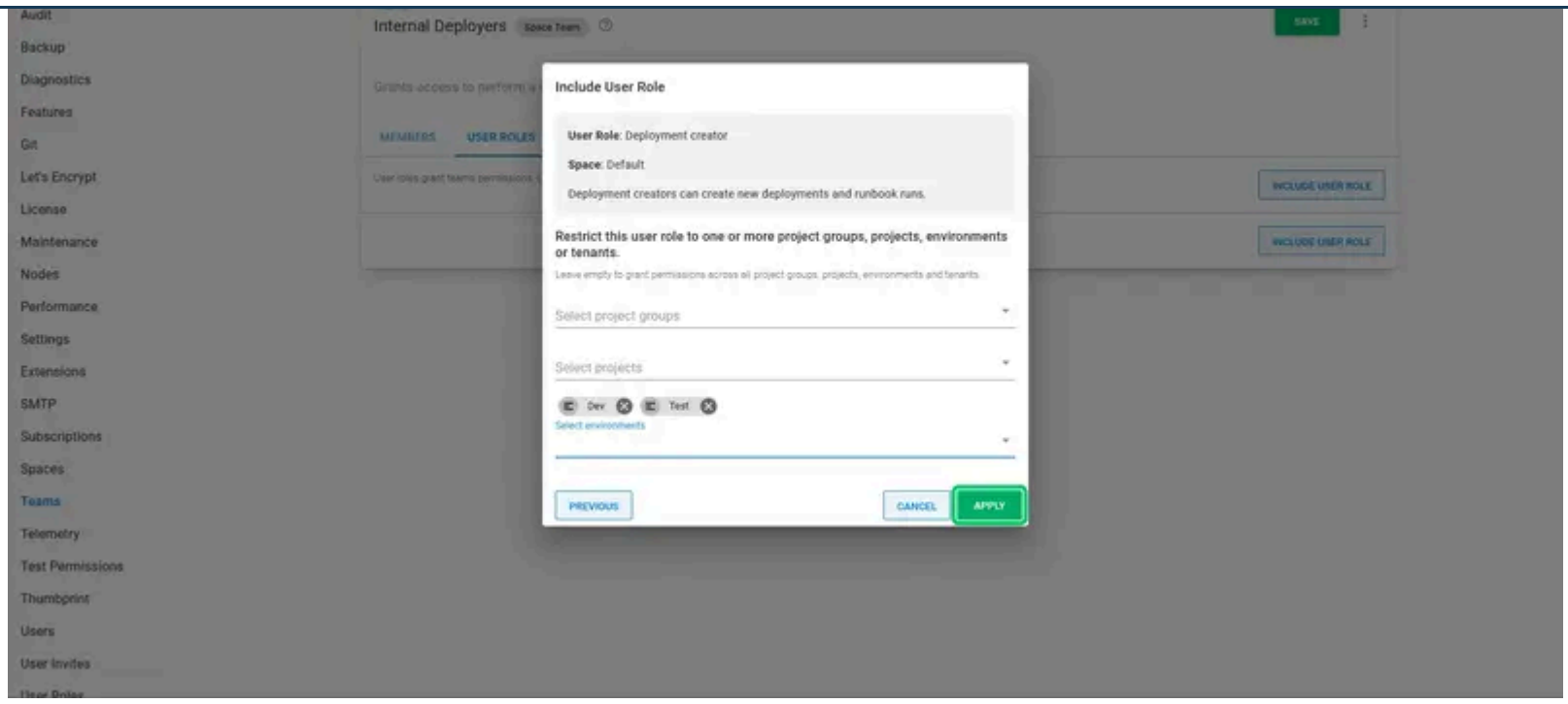
Select the **Deployment creator** role from the dropdown list. As the name suggests, this role allows a user to create a deployment, which results in the deployment process being executed.

Click **DEFINE SCOPE**:

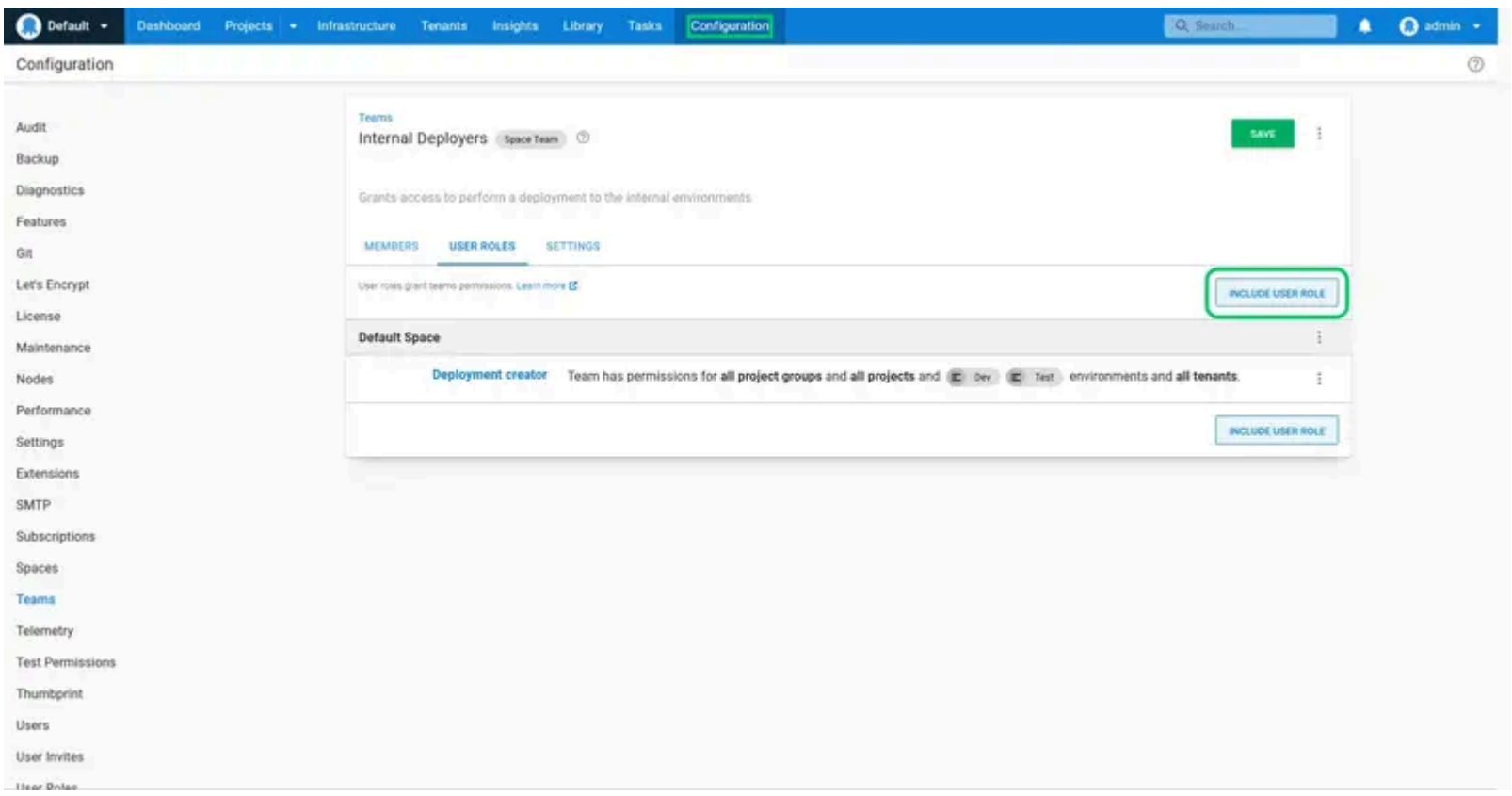


We only want to allow the internal deployer to deploy to the internal environments. Select the **Dev** and **Test** environments, and click **APPLY**:

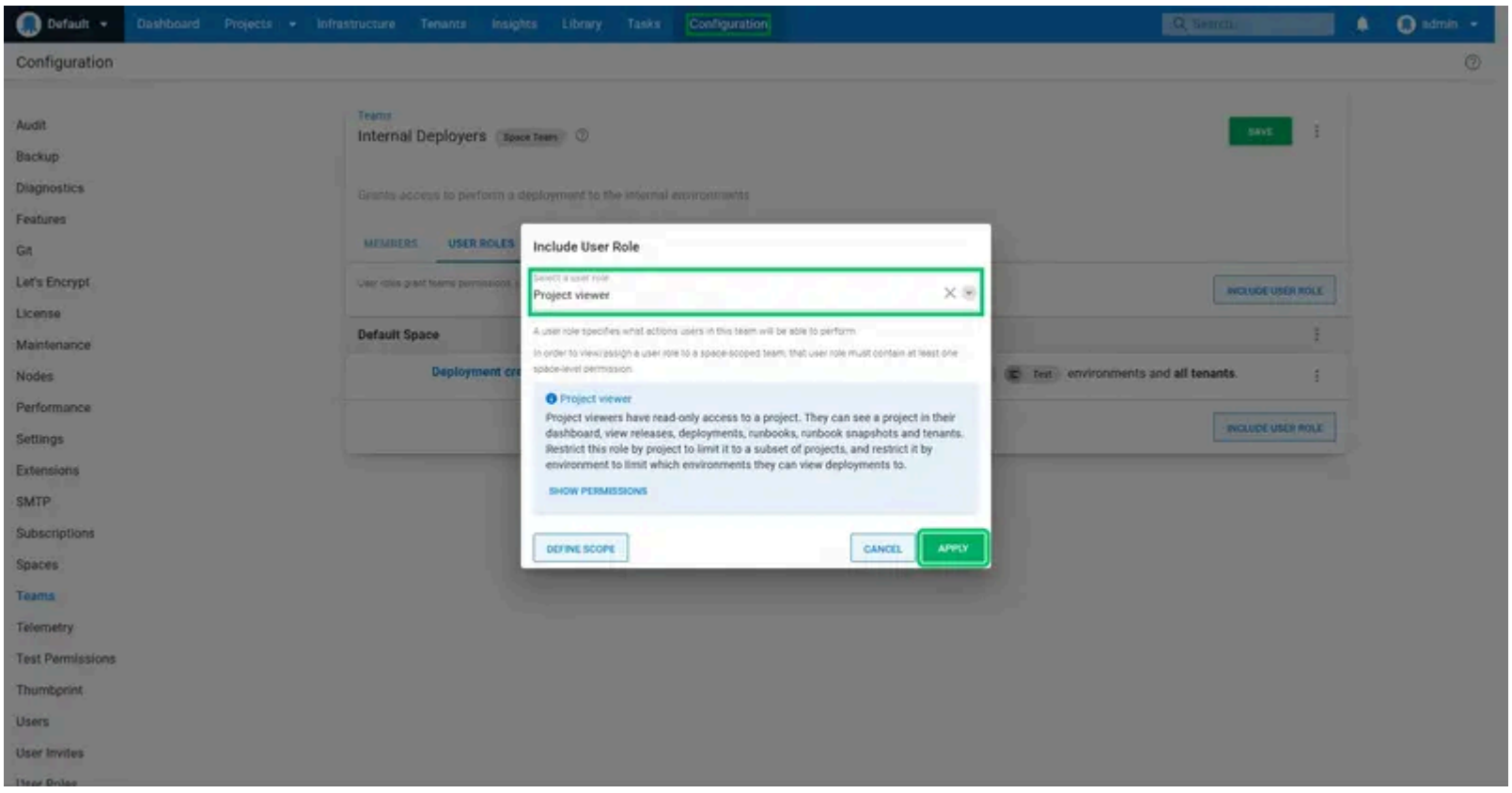
Octopus Deploy



The permissions are then applied. We need a second permission to allow the internal deployer to view the projects dashboard. Click **INCLUDE USER ROLE** again:

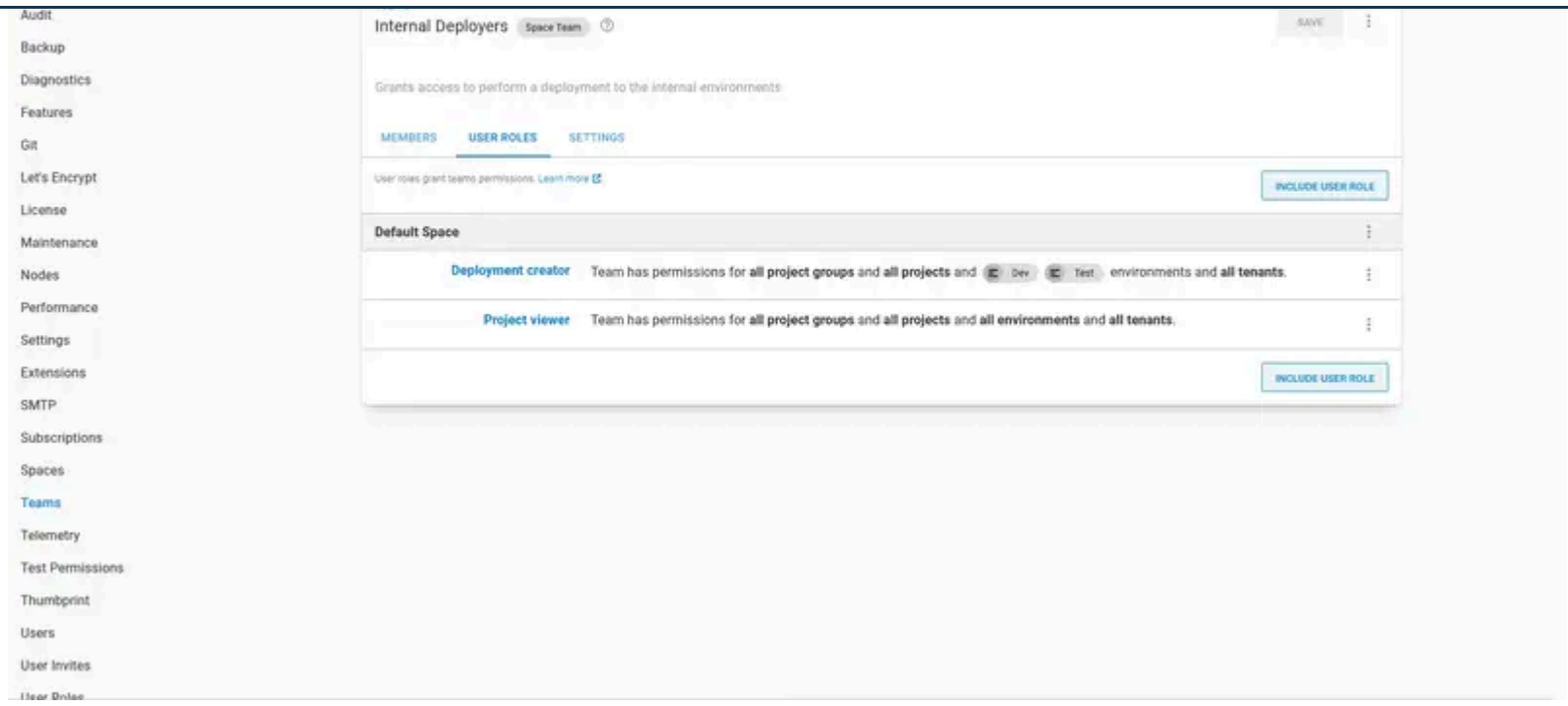


Select the **Project viewer** role. This role does not need to be scoped, so click the **APPLY** button:

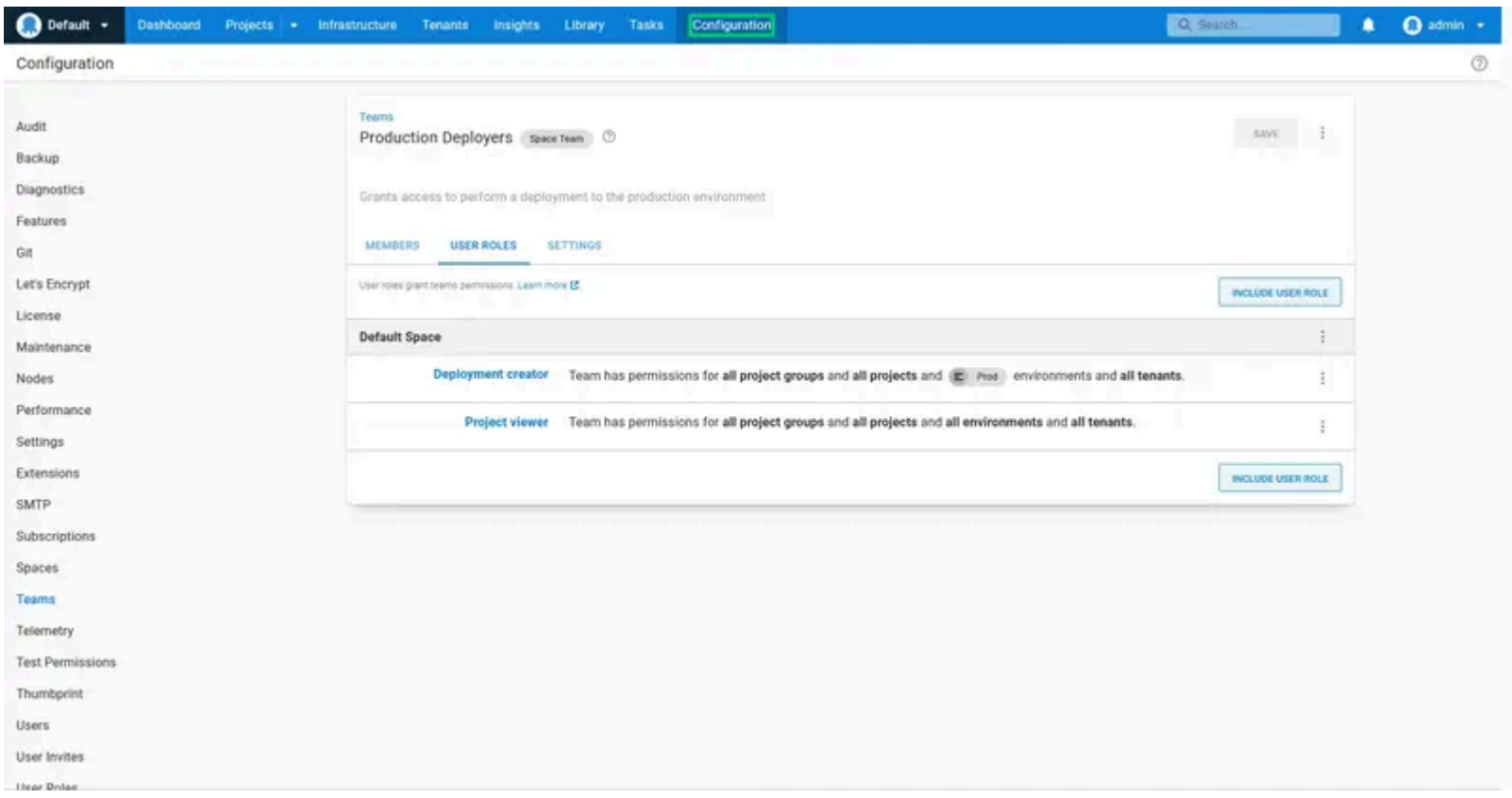


Here are the final set of roles applied to the team:

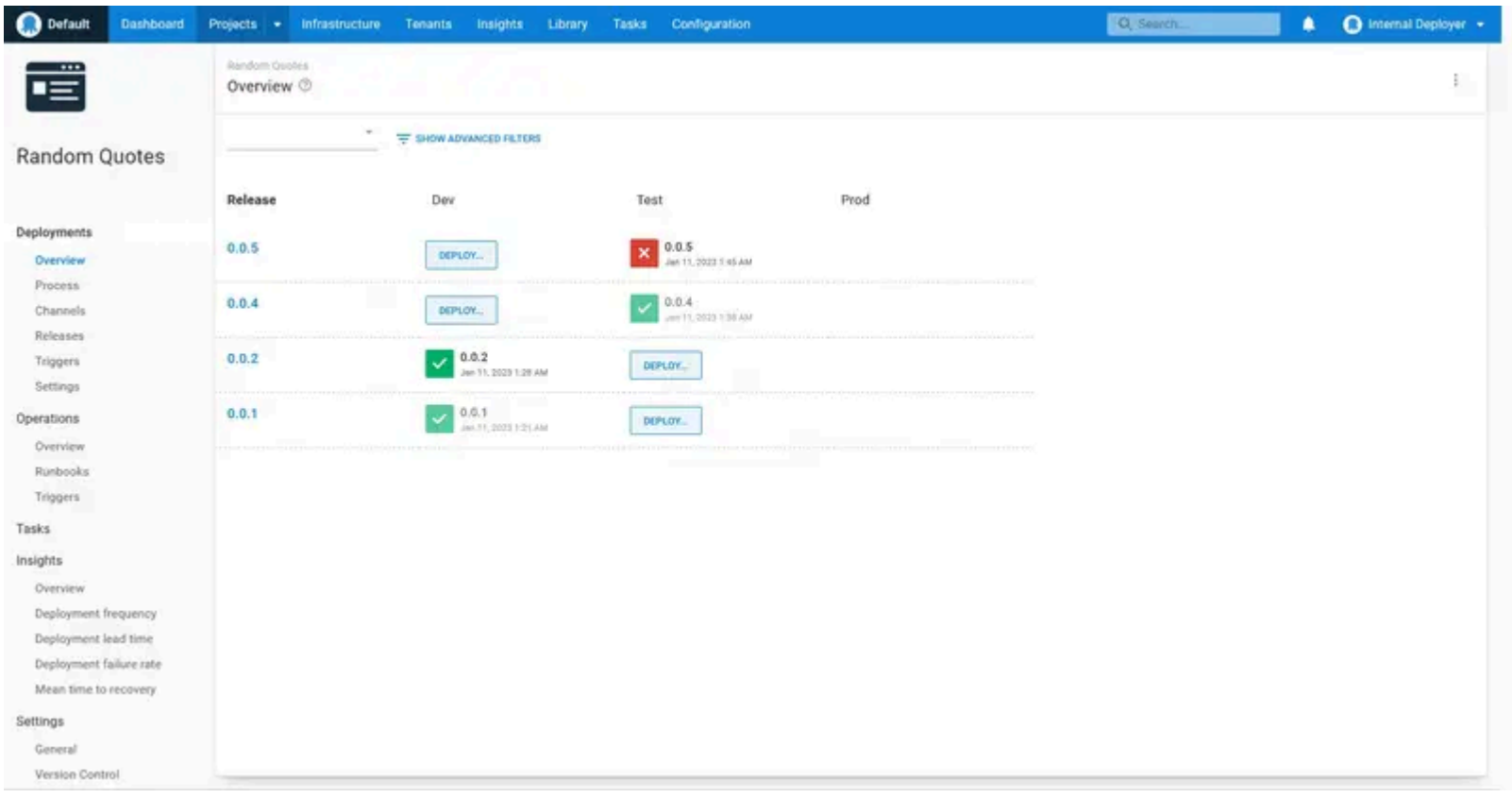
Octopus Deploy



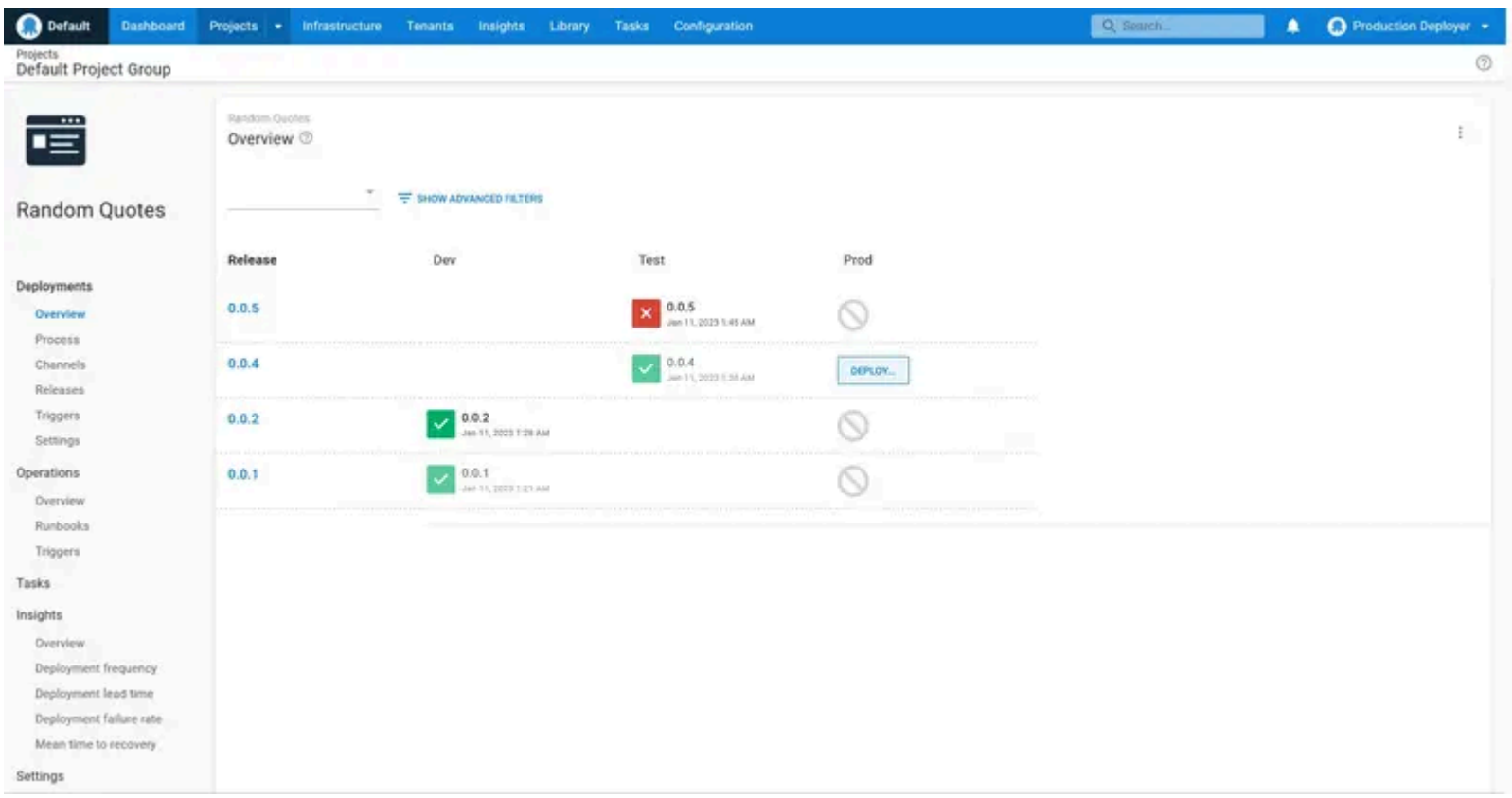
Repeat the process to create a team called **Production Deployers** that includes the **productiondeployer** user, and grants the **Deployment creator** role scoped to the **Prod** environment:



When we log in as the **internaldeployer** user, we see that the Random Quotes project dashboard shows **DEPLOY...** buttons for the **Dev** and **Test** environments. Any deployments in the production environment will be visible, but they cannot be created by this user:



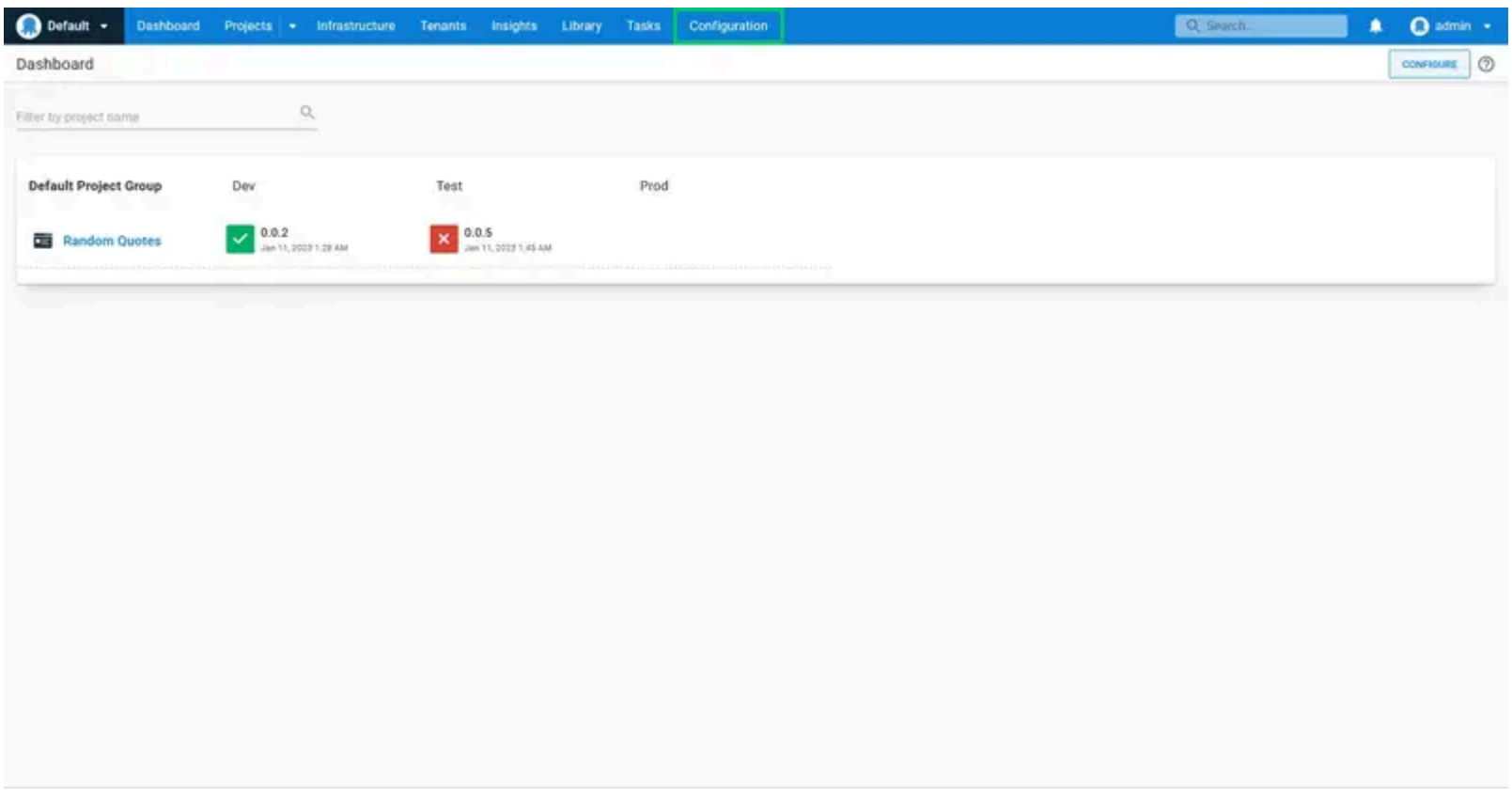
available to be promoted to the next environment.



Audit Log

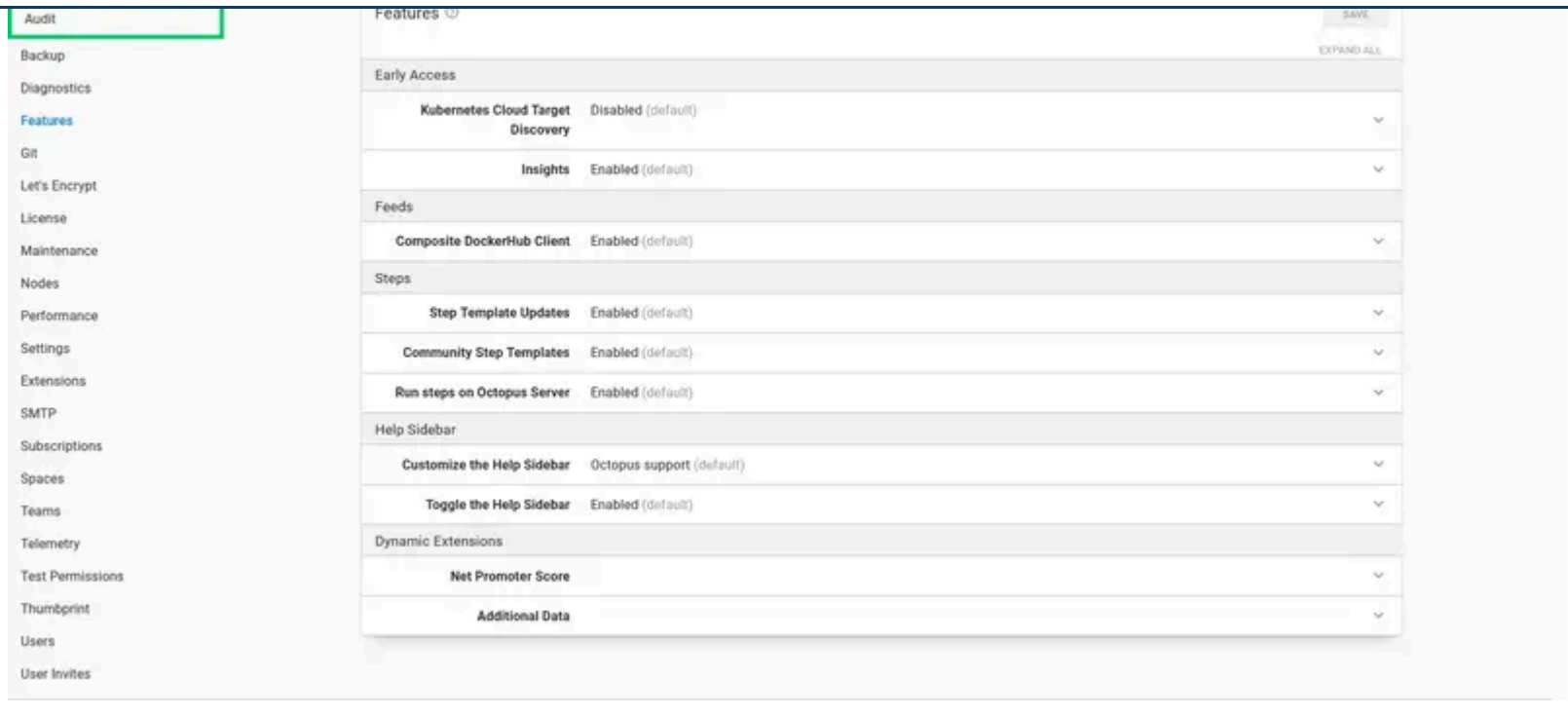
Important interactions in Octopus are tracked in the audit log. This is useful for teams that have security or legislative requirements to track the changes and deployments made to their infrastructure.

To view the audit log, click the **Configuration** link:

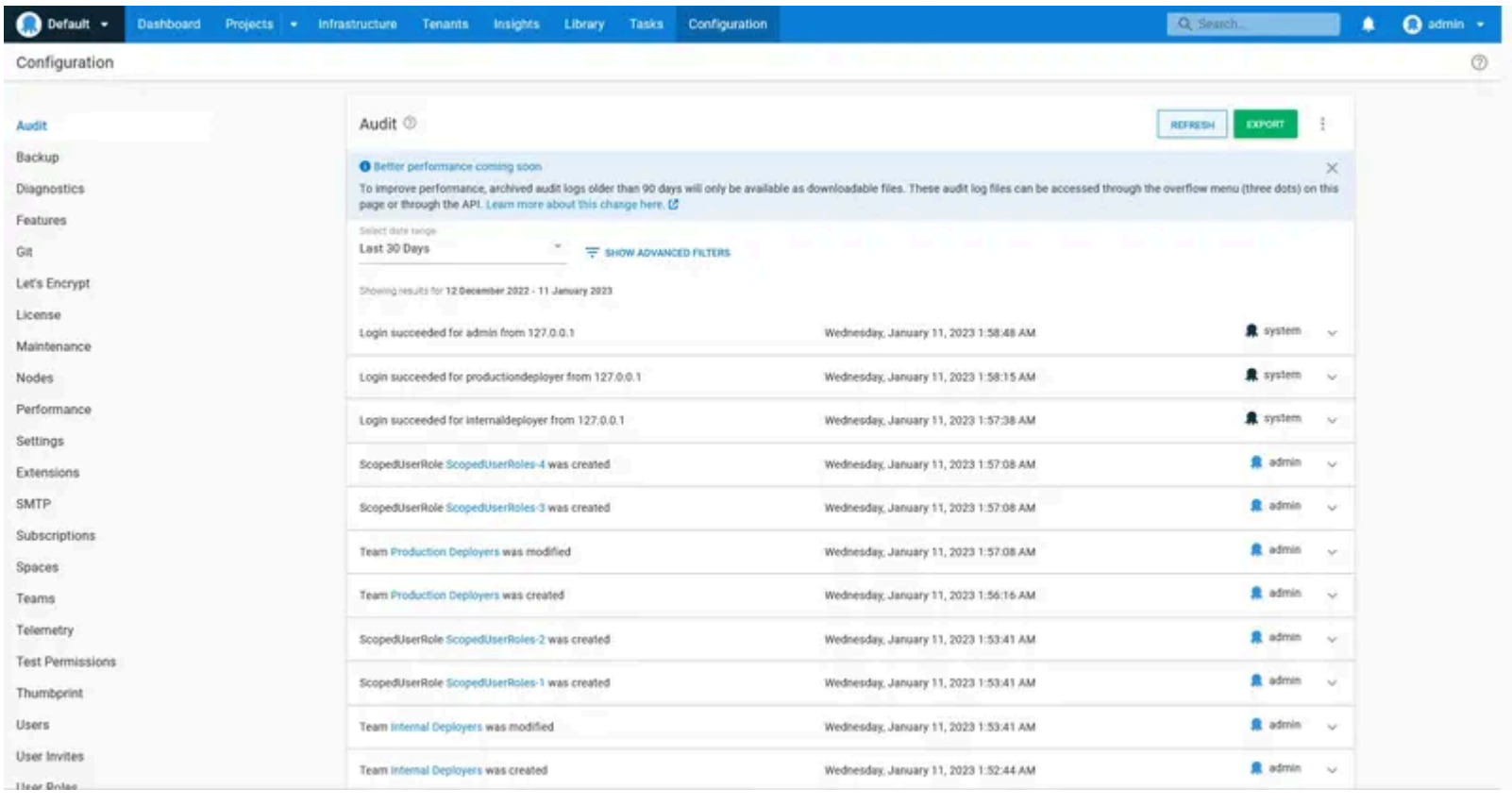


Click the **Audit** link:

Octopus Deploy



A complete list of records are shown, with filtering available to help find specific events:



Conclusion

In this guide we ran through the process of building a complete CI/CD pipeline with:

- GitHub Actions building and testing the source code and pushing the package to Octopus.
- Octopus deploying the package to the **Dev**, **Test**, and **Prod** environments.
- Email notifications generated when deployments succeed or fail.
- Manual sign off for deployments to the **Test** and **Prod** environments.
- Users with limited access to create releases in a subset of environments.

This is a solid starting point for any development team, but Octopus offers so much more! Below are more resources you can access to learn about the advanced functionality provided by Octopus:

- The Octopus documentation
- The Octopus blog
- Upcoming events and webinars
- The Octopus community slack channel