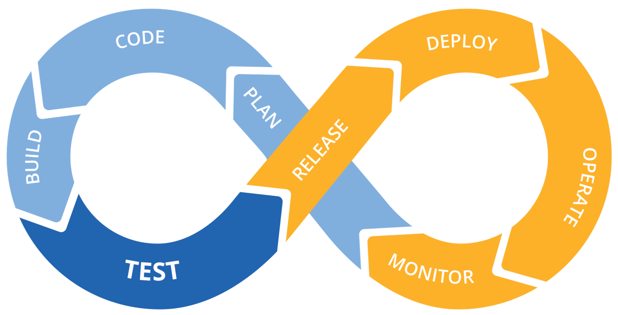
Lab 07 - DevOps

Introduction video [https://drive.google.com/file/d/12yIfn1iPEr6Qy2EGEE2RiqRv-K13y0Ar/view?usp=sharing]

Introduction



DevOps, or Developer Operations, is a culture that promotes collaboration between Development and Operations teams in order to improve the speed of code release and quality of the code.

It isn't a certain technology or set of tools, but rather a way of thinking about the deployment process overall.

DevOps teams usually have increased automatization in things such as code linting, testing, deployment and releasing.

What do we usually mean when talking about DevOps ?

Although DevOps isn't defined as a set of tools, commonly found technologies are:

- 🚚 Continuous Integration / Continuous Deployment (CI/CD) systems
  - Purpose: Builds and delivers the code we write to our clients
  - Software: Jenkins, Codeship, GitLab CI, Github Actions, etc.
- ✅ Testing frameworks
  - Purpose: Tests our code to ensure that it performs as intended. Depending on the scope, we can write unit tests (smallest tests, checking only one function) or complex E2E tests (which test our whole app and are usually run live on a test environment configured identically to production machines)
  - Software: Jest, Mocha, TestCafe, Cypress, Browserstack, etc.
- 🚑 Error analytics
  - Purpose: We need a way to find out if an app crashed and why without (or ideally before) the user finds out. These tools capture errors and relevant information, and can also provide graceful ways of handling them on the user side.
  - Software: Sentry, Rollbar

In this lab we'll go through a bit of all of these by building a simple JavaScript Hello World app.

🚚 App setup and CI/CD

We'll first need an app to build and test, as well as a repo to hold our code. The app we'll build will be a simple “hello world” vuejs app, as the focus is on the DevOps part.

If you need a step-by-step guide with images, you can try following this guide as well [https://medium.com/swlh/how-do-i-deploy-my-code-to-heroku-using-gitlab-ci-cd-6a232b6be2e4].

- 🗀 **1. Set up a GitLab Repo**

We'll first need a GitLab account, so head on over to GitLab [https://gitlab.com] and sign up for one.

Once you have your GitLab account set up:

- click the **New Project** button.
- select **Create blank project**
- set project name as **testvue**

Remember to add your SSH key to your GitLab account, or else we won't be able to add anything to the repo. Check this link for more information: https://docs.gitlab.com/ee/ssh/README.html#adding-an-ssh-key-to-your-gitlab-account [https://docs.gitlab.com/ee/ssh/README.html#adding-an-ssh-key-to-your-gitlab-account]

And your project is ready to go!

- 💥 **2. Build a sample app**

In order to start building our app, we'll first have to clone our repository. You can use the commands below in either Windows WSL [https://www.omgubuntu.co.uk/how-to-install-wsl2-on-windows-10], Linux, or macOS.

```
# Initial configuration
git config --global user.name "Your User Name"
git config --global user.email "yourgitlab@address.com"

# Go to the `Clone` dropdown and copy your repo URL, under the Clone with SSH section:
git clone <paste URL here>
```

This will create a new folder with your (for now, blank) repository. Let's build our app now:

```
cd testvue

npm install -g @vue/cli

# Mind the dot on the next line
vue create .

# Select "Manually select features"
# Select the following:
# * Babel
# * Linter / Formatter
# * Unit Testing

# When prompted to pick a unit testing solution, select Jest
# Keep defaults for all others
```

You should now have a Vue project with all the required tools installed.

- 🔧 **3. Configure GitLab CI to build the app**

Our app is ready to be built! Let's try building it locally, to see how the process works:

```
npm run build
```

After this command runs, we'll have a **dist** folder containing the files that we need to upload to our server.

Let's configure the GitLab CI to run our build for us. In order to do this, create a *.gitlab-ci.yml file in the root repository, with the following content:

```
build_app:
        image: node:14
        script:
                - npm install
                - npm run build
```

We can now add our changes to the repo, and push them to GitLab:

```
git add .
git commit -m 'initial commit'
git push
```

If we now go to the CI/CD tab on the left-hand menu in our repo, we'll see our build being run.

We now have our app in a repo, and a CI/CD pipeline to help it being deployed. But where will it be deployed to ? We'll set up a Heroku server to deploy it!

But before we can do that, we're going to need a small server that will serve the actual app to us. Luckily, that's pretty easy to do. First, run these commands:

```
npm install --save express serve-static
```

And then create the **server.js** file in the root of the repository, with the following content:

```
const express = require('express');
const serveStatic = require('serve-static');
let app = express();

app.use(serveStatic(__dirname + "/dist"));

const port = process.env.PORT || 8080;
app.listen(port, () => {
    console.log('Listening on port ' + port)
});
```

- 📝 **4. Sign up to Heroku and launch a new app**

- Go to https://www.heroku.com/ [https://www.heroku.com/] and sign up for a free account.
- Select “Create new app”, and give it a name
- Go to your account settings (top right), scroll down and copy the API key

We're going to need to add our Heroku API key to the GitLab config before we can continue.

- 🎢 **5. Set up environment variables in GitLab**

- In the GitLab web UI, go to Settings → CI/CD
- Expand the Variables section
- Add a new variable named HEROKU_API_KEY with the value being the API key you copied from the previous step

- 📝 **6. Configure GitLab to deploy our app**

We now have to add another step to our **.gitlab-ci.yml** file:

```
stages:
        - build
        - deploy
build_app:
        image: node:14
        stage: build
        script:
                - npm install
                - npm run build
        artifacts:
                paths:
                    - dist/
```

```
deploy_app:
        image: "ruby:2.5"
        stage: deploy
        script:
                - apt-get update -qy
                - apt-get install rubygems ruby-dev -y
                - gem install dpl
                - dpl --provider=heroku --app=$HEROKU_APP_NAME --api-key=$HEROKU_API_KEY --skip-cleanup
```

Notice that we've added **stages** to our GitLab CI configuration. This tells GitLab to run our build step before our deploy step.

We've also added artifacts to our build step - this tells GitLab to keep the files in the dist/ directory in the final deployment build.

We now commit our changes:

```
git add .
git commit -m 'heroku deploy changes'
git push
```

Now go to the CI/CD page on GitLab and monitor the deployment process.

Access your app by clicking the Open App button in the Heroku dashboard, and it should be live!

✅ Testing our app

Now, nobody wants to use a broken app, so let's test it and make sure it works right.

- **1. Unit Testing with Jest**

We already have an example test in the **tests/** folder. Let's run it and see how the output looks like:

```
npm run test:unit
```

The result shows that our test ahs passed - great!

Let's also add this to our CI configuration:

```
stages:
        - build
        - test
        - deploy
build_app:
        image: node:14
        stage: build
        script:
                - npm install --include=dev
                - npm run build

test_app:
        image: node:14
        stage: test
        script:
                - npm install --include=dev
                - npm run test:unit

deploy_app:
        image: "ruby:2.5"
        stage: deploy
        script:
                - apt-get update -qy
                - apt-get install rubygems ruby-dev -y
                - gem install dpl
                - dpl --provider=heroku --app=$HEROKU_APP_NAME --api-key=$HEROKU_API_KEY --skip-cleanup
```

We now add our new changes:

```
git add .
git commit -m 'add tests'
git push
```

Follow the progress in the GitLab CI/CD screen.

Try making the test fail and see what happens

- **2. End to End testing with TestCafe**

We can ensure that our app works correctly by testing it with end-to-end tests. Compared to unit tests, end-to-end tests run on a live environment, and test our app back-to-back (hence the end-to-end name). We don't have a backend on our demo app, but we can write a simple end-to-end test to get the hang of the steps required.

First, we'll need to install testcafe:

```
npm install --save testcafe
```

We have now added testcafe, but we do not have any E2E tests written. Here's an example one that you can place in the tests/ folder:

```
import {Selector} from 'testcafe';

// tests/e2e.js
// Fixtures are test categories - each test has to be a part of one
fixture `Hello World Tests`
    .page `your-url-here`;

test('Test1', async t => {
  // The t parameter we have here is our "control knob" - we can
  // use it to perform various tasks in the browser that we'll open

  // Tests usually do things like check if elements exist.
  // In this example, we'll check that the h1 element has the "Welcome" string
  // in it:
  const h1TitleElement = Selector('h1') // We use CSS selectors here


  await t.expect(h1TitleElement.textContent).contains('Welcome')
})
```

Now, we need to somehow run our test. We're going to use a npm script [https://www.freecodecamp.org/news/introduction-to-npm-scripts-1dbb2ae01633/] for this. Edit the scripts section of the package.json file and add a testcafe entry:

```
{
  //...
  "scripts": {
    //...
    "testcafe": "testcafe remote tests/e2e.js",
    //...
  },
  //...
}
```

Let's run it:

```
npm run testcafe
```

This will give you a URL to go to in order to start the test run. Copy it in your browser and see the test being run!

Integrating TestCafe into GitLab

So we've seen how the end to end tests work, but it's no use if we don't run them regularly. Let's add the tests to our .gitlab-ci.yml configuration:

```
stages:
        - build
        - test
        - deploy
build_app:
        image: node:14
        stage: build
        script:
                - npm install --include=dev
                - npm run build
e2e_tests:
        stage: test
        image:
                name: testcafe/testcafe
                entrypoint: ["/bin/sh", "-c"]
        script:
                - npm install --include=dev
                - /opt/testcafe/docker/testcafe-docker.sh chromium tests/e2e.js
test_app:
        image: node:14
        stage: test
        script:
                - npm install --include=dev
                - npm run test:unit

deploy_app:
        image: "ruby:2.5"
        stage: deploy
        script:
                - apt-get update -qy
                - apt-get install rubygems ruby-dev -y
                - gem install dpl
                - dpl --provider=heroku --app=$HEROKU_APP_NAME --api-key=$HEROKU_API_KEY --skip-cleanup
```

Don't forget to add the HEROKU_APP_NAME and HEROKU_API_KEY variables to CI CD Variables [https://docs.gitlab.com/ee/ci/variables/]

Let's commit the changes and see our output in the GitLab CI/CD pipelines listing:

```
git add .
git commit -m 'add e2e tests'
git push
```

Try adding a failing end-to-end test and see what happens.

Making sure it doesn't crash

Every app crashes now and then, but we can make sure that we fix issues ASAP if we find out about them ASAP. Let's set up Sentry and see how it works.

- **1. Create a Sentry account and link our app**

- Go to https://sentry.io [https://sentry.io] and sign up for a free account
- In the welcome tutorial, select Vue when prompted for a platform
- Follow the steps provided to add Sentry to our demo app

You'll have to paste the code snippet provided to you in the src/main.js file

Commit your changes and you should be all set for tracking errors with Sentry:

```
git add .
git commit -m 'add sentry'
git push
```

- **2. Triger an error and get info about it**

Now we have Sentry up and running, but how do we know if it works ? Let's manually trigger an error.

Open the App.vue file and add, between the template tags, the following line:

```
<template>
//...
<button onClick="() => throw new Error('err')">Click me to boom</button>
//...
</template>
```

Commit and push the changes, open your app in the browser, click the button we just made above and you should have an error pop up in the Sentry interface:

```
git add .
git commit -m 'add dummy error'
git push
```

In the Sentry interface you can see a lot of useful info about the error, such as:

- The browser and the OS the error originated from
- URL and IP address the user was on
- A complete stacktrace of your app

If you link the Sentry project with your GitLab repository, you can even see which commit caused the issue, and even the exact line of code that threw an error.

Try linking your GitLab account with Sentry and see what extra info you get!

Feedback

Please take a minute to fill in the **feedback form [https://forms.gle/NuXCJktudGzf4rLg6]** for this lab.

se/labs/07.txt · Last modified: 2021/12/06 18:50 by marius.calapod