How to setup continuous integration and deployment workflows for ReactJS using GitHub actions

Tiago Duarte
02 December, 2019

GitHub Actions: How to setup continuous integration and deployment workflows for ReactJS

Since the beginning of our times, that quality has always been a focal point at Coletiv and we pride ourselves by enforcing processes that prevent bad code from getting into production. Among others, continuous integration (CI) and continuous deployment (CD) have been since day one standard steps in our quality assurance (Q&A) process for all our projects.

Being heavy users of git, specially GitHub we couldn’t wait to get our hands into GitHub actions and experiment if it could be a good fit for our Q&A process. According to GitHub:

GitHub Actions makes it easy to automate all your software workflows, now with world-class CI/CD. Build, test, and deploy your code right from GitHub. Make code reviews, branch management, and issue triaging work the way you want.

We decided to try it on a ReactJS project and evaluate if it could be part of our tool belt.

Expected End Result

A very common setup we use in our projects is to have a production and a staging environment, we mirror these environments with two branches:

- master — contains the source for the **production environment**, containing the live system being used by the end users
- develop — contains the source for the **staging environment** and is used internally to test new features before they end up in the hands of our users

This means that every time there is a pull request into one of these branches, we run the continuous integration steps (e.g.: run unit and integration tests). If the steps don't report any error and other developers approve the code it is merged into the branch, triggering a deployment to the respective environment.

This is what we are going to implement in this tutorial. But let's stop with bla bla bla 🙊 and let's get our hands dirty! 🛠

Step 1 — Initial Setup

- Create a new GitHub repository if you don't have one already
- Clone the repository to your local machine

- We are going to use the create react app cli. Inside the cloned repository run npx create-react-app "." --typescript and commit the changes
- Create a new develop branch and push both master and develop to the remote repository

Step 2 — Setup Continuous Integration Workflow

- Create a new branch from the develop branch
- Create a .github/workflows repository at the root of the project and inside create a continuous-integration.yml file (you can pick a different name if you want)
- Paste the following content into the file:

```yaml
name: Continuous Integration

on: [pull_request]

jobs:
  buildAndTest:
    name: Build and Test
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v1

      - name: Install Dependencies
        run: yarn

      - name: Build
        run: yarn build

      - name: Test
        run: yarn test --watchAll=false
```

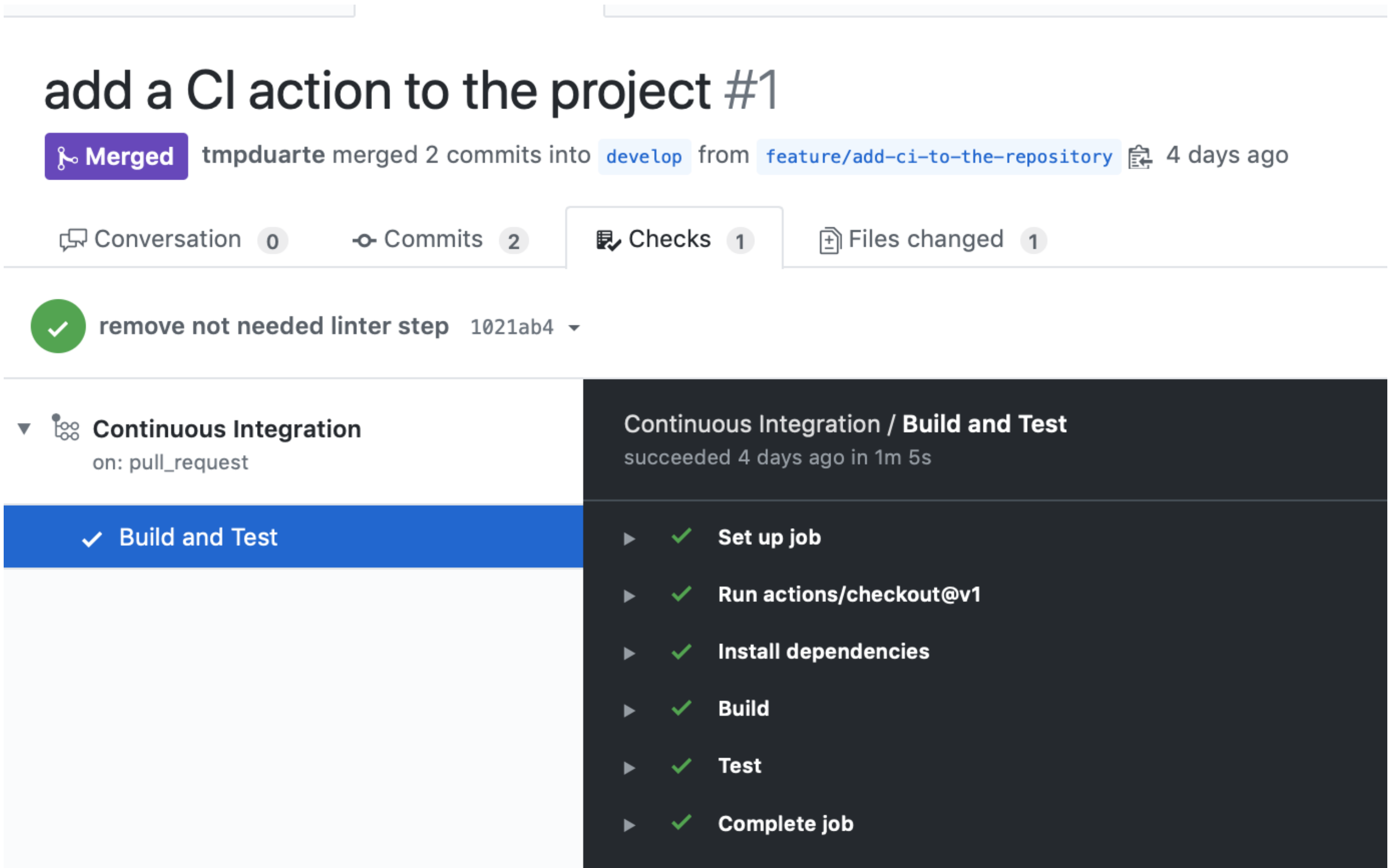
Translating this into a human readable form, we are creating **a workflow/action named *Continuous Integration* that runs on every pull request**.
This workflow consists of a single job called *Build and Test* that runs on `ubuntu-latest`. The job checks out the code submitted in the pull request, then installs all the dependencies, creates a build and runs all the tests once by passing the `--watchAll=false` option.

If any of the steps fail, the whole workflow fails and reports back to the pull request. As a best practice we always enforce the checks to succeed before allowing code to be merged.

Feel free to add more (quality control) steps like enforcing a linter to run so that code that does not comply with coding guidelines doesn't get merged.

For more information on the structure and all possible options you can use on a workflow file you can visit the workflow syntax for GitHub.

To test the newly created workflow, just push your branch, create a pull request and observe the workflow take place and report the status back to the pull request:


add a CI action to the project #1

Merged tmpduarte merged 2 commits into `develop` from `feature/add-ci-to-the-repository` 4 days ago

remove not needed linter step `1021ab4`

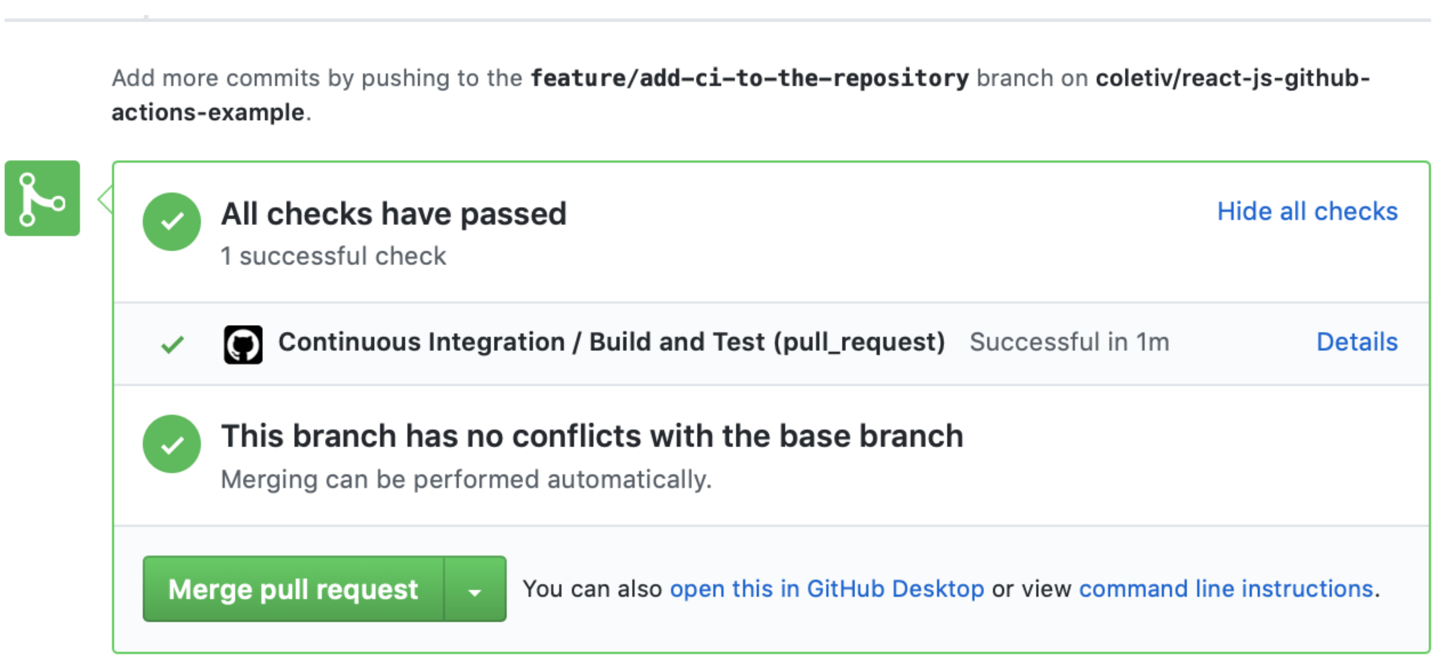
Continuous Integration
on: pull_request

✓ Build and Test

Continuous Integration / Build and Test
succeeded 4 days ago in 1m 5s

✓ Set up job
✓ Run actions/checkout@v1
✓ Install dependencies
✓ Build
✓ Test
✓ Complete job

GitHub workflow running



Add more commits by pushing to the **feature/add-ci-to-the-repository** branch on **coletiv/react-js-github-actions-example**.

All checks have passed
1 successful check

✓ **Continuous Integration / Build and Test (pull_request)** Successful in 1m

This branch has no conflicts with the base branch
Merging can be performed automatically.

You can also open this in GitHub Desktop or view command line instructions.

GitHub workflow reporting back to the pull request

Step 3— Setup Continuous Deployment Workflow

We decided to host our application on two distinct Amazon S3 buckets, one for each environment (staging & production). Feel free to use any other host for your application (e.g.: your own server) but keep in mind that you might need a different action to sync the build files (e.g.: ssh deploy action).

Moving on:

- Inside the .github/workflows folder at the root of the project create a continuous-deployment.yml file. You can pick a different name if you want
- Paste the following content into the file

```
name: Continuous Deployment
on:
  push:
    branches:
      - master
      - develop

    jobs:
      deploy:
        name: Deploy
        runs-on: ubuntu-latest

      env:
        SOURCE_DIR: 'build/'
        AWS_REGION: 'us-east-1'
        AWS_ACCESS_KEY_ID: ${{ secrets.STAGING_AWS_ACCESS_KEY_ID
        AWS_SECRET_ACCESS_KEY: ${{ secrets.STAGING_AWS_SECRET_ACC

      steps:
        - uses: actions/checkout@v1
        - name: Install dependencies
          run: yarn

        - name: Build
          run: yarn build

        - name: Deploy
          uses: jakejarvis/s3-sync-action@v0.5.0
          with:
            args: --acl public-read --follow-symlinks --delete
          env:
            AWS_S3_BUCKET: ${{ secrets.STAGING_AWS_S3_BUCKET }}
```

Let's translate again this into a human readable form. We are creating a **workflow named *Continuous Deployment* that runs every time code gets pushed to either the `develop` or the `master` branch**.
This workflow consists of a single job called *Deploy* that runs on an `ubuntu-latest` machine. The job checks out the freshly pushed/merged code, installs all the dependencies, creates a build and deploys the `build` folder into the AWS S3 bucket.

If you look closely we have introduced a few new things in relation to the CI action:

- `env` — the `env` key allows us to share common environment variables that can be used by the steps defined in the job (e.g. `SOURCE_DIR` is used by the deploy step). You can check here the documentation for the `env` key and how the values cascade from the job key to the steps key
- some values for the keys have this weird syntax `${{secrets.<SOME_NAME>}}`. This is a so called expression. This expression uses a secret, which is an automatically encrypted value defined by you in your repository, that you don't want to see exposed to anyone (e.g.: S3 bucket keys). You can read about secrets and how to define them here
- `jakejarvis/s3-sync-action@v0.5.0` — this was the action we chose to deploy the `build` folder into our S3 bucket. Please note that we pass some args to the action that basically tells it to delete any files that aren't in the current build and also make the files publicly readable (`--acl public-read`). You can read about all the `args` and `env`'s you can pass to the action in here

Step 4— Different Branches = Different Environments

You might have noticed that in the workflow defined in the previous step we would deploy to the staging environment code merged/pushed on both `develop` and `master` branches.

It is now time to deploy each branch to its respective environment. Update the steps key in the continuous-deployment.yml file with the following code:

```
  steps:
    - uses: actions/checkout@v1

    - name: Install dependencies
      run: yarn

    - name: Build
      run: yarn build

    - name: Deploy staging
      if: github.ref == 'refs/heads/develop'
      uses: jakejarvis/s3-sync-action@v0.5.0
      with:
        args: --acl public-read --follow-symlinks --delete
      env:
        AWS_S3_BUCKET: ${{ secrets.STAGING_AWS_S3_BUCKET }}

    - name: Deploy production
      if: github.ref == 'refs/heads/master'
      uses: jakejarvis/s3-sync-action@v0.5.0
      with:
```

```
      args: --acl public-read --follow-symlinks --delete
    env:
      AWS_S3_BUCKET: ${{ secrets.PRODUCTION_AWS_S3_BUCKET }}
```

We now have two deploy steps, one for staging and one for production, that only run when the code is merged/pushed to their respective branches. We achieve this by having an if key that checks which branch triggered the workflow: `if: github.ref == 'refs/heads/branch_name'`. The two steps differ in their `name`, `if` and `env` keys.

We tried different solutions to avoid step duplication, but all of them seemed very contrived.
We opted for a more explicit solution, although we are aware that it has a certain level of duplication. For example, if there is a new release of the sync action we need to update the version in both steps.

As an exercise, you can try to have different builds for each environment. Maybe the staging version of your app communicates with a mock server while the production version communicates with the real server.

To test the newly created workflow, we merged a pull request into the `develop` branch. You can observe the workflow skip the production deployment as we did not push code to the `master` branch:

GitHub actions — continuous deployment workflow

Step 5— Stretch Goal: CI & CD Badges

GitHub actions also provide an easy way to integrate the workflow status in your readme with badges. The syntax is as follows

`https://github.com/<OWNER>/<REPOSITORY>/workflows/<WOR`

In our case:

```
    ![](https://github.com/coletiv/react-js-github-actions-exampl
    ![](https://github.com/coletiv/react-js-github-actions-exampl
```

Which results in the following screenshot when you visit the GitHub project:

GItHub workflow badges example

By using the same workflow on two different branches can cause the badges to miscommunicate the status of the builds. Let me explain: if a deployment to the staging environment failed the *Continuous Deployment* badge would be red. If in the meanwhile a deployment to the production environment occurred properly, the badge would get back to green even though we haven’t fixed the staging deployment.
To fix this you would need a workflow for each branch which would allow you to have a separate set of badges per each environment.

We opted to have a single set of badges to avoid workflow duplication and in any case if a workflow fails you get an email informing you.

Step 6— Don't be shy! 😳 Go ahead try it yourself

The companion repository of this article is fully functional so feel free to do a pull request and see the actions being triggered on the actions tab.
If the pull request gets approved and merged, the continuous deployment action starts and you will be able to see your changes on the respective environment (staging / production).

Conclusion

GitHub actions are a serious contender to the CI / CD scene, specially due to the community of developers Github has which quickly led to the development of so many open source actions that you can cherry pick and use on your own actions.

So far the only complains we have is the difficulty to define `env` variables conditionally, as seen in step 4, which led us to duplicate a big part of the workflow. Also, we couldn't find a way to cancel a running job triggered by a pull request that got updated, it doesn't make much sense to continue the

action if the branch just got updated, we are just wasting resources.
Other than that we haven't found anything that actions couldn't do that we do on other CI/ CD tools (e.g. CircleCI) and vice-versa, so the choice of tool might come down to a matter of taste or bet in which platform will evolve better.

In our case, we like the fact that we have everything in a single place and we don't need to jump between sites to check why a certain job has failed. On the downside, you might be locking yourself down even more to a platform, which can cause you problems if you decide to change to another code hosting platform down the line.

Thank you for reading!

Thank you so much for reading, it means a lot to us! Also **don't forget to follow Coletiv on Twitter and LinkedIn** as we keep posting more and more interesting articles on multiple technologies.

In case you don't know, Coletiv is a software development studio from Porto specialised in Elixir, Web, and App (iOS & Android) development. But we do all kinds of stuff. We take care of UX/UI design, software development, and even security for you.

So, let’s craft something together?

Continuous Integration ReactJS JavaScript Web Development

Login

Add a comment

M↓ MARKDOWN ADD COMMENT

Upvotes Newest Oldest

g **gom tv**
0 points · 45 days ago

After we have successfully built to the develop branch, is it now time to merge it into master and production? Does this mean we never will do the pull request step on the master branch?

Ah, we create a PR from develop to master before pushing to prod?

I might lack some fundamental understanding about the workings of git.

Tiago Duarte
0 points · 45 days ago

Hi @gom, when you create a pull request to the master branch, after the code review and all the checks have passed, you can accept/merge the code. When that happens the Github Action will trigger a deployment of the code contained in the master branch to the production environment.

gom tv
0 points · 45 days ago

Thank you!

So I'm not far off in the following flow:

New feature needs to be added, we create branch #featurezyx
Pull request for #featurezyx to be merged into development/staging branch.
Pull request from development/staging branch to be merged into master.

Tiago Duarte
0 points · 45 days ago

Exactly! That's the usual flow we are following. When the pull requests are merged:

into development -> a deployment is triggered to the dev/staging environment so that internal testing can take place

into master -> a deployment is triggered to the production environment so that users can have access to the new features / fixes

When products / companies get bigger the flow tends to be more complex. For example in a project we are working on there is another environment called acceptance where the client tests the features.

Meaning on the

dev environment - developers test the features

acceptance environment - clients test the feature

production environment - users have access to the feature

ARTICLES

Related essays

See all →

SOFTWARE DEVELOPMENT

Using WebSockets With Cookie-Based Authentication

Tiago Duarte
08 May, 2020

ELIXIR, SOFTWARE DEVELOPMENT

How to protect Amazon S3 media links with Elixir and Arc

Daniel Almeida
30 April, 2020

ELIXIR, SOFTWARE DEVELOPMENT

Elixir — Sort Lists by Date

Nuno Marinho
16 April, 2020

Your full-service digital

Contacts

agency

humans@coletiv.com

+351 914 541 645

Company

Showcase

Services

Blog

About Us

Network

Coletiv

Adamant

Significa

Contacts

Email humans@coletiv.com

Phone +351 914 541 645