Continuous Integration and Continuous Delivery with GitHub Actions

- Having the ability to iterate quickly on a software project can be crucial in today's software development world. In particular, if we have the ability to deploy new code to production easily and often, it can give us the opportunity to fix bugs promptly as they arise, to get new features into users' hands frequently, and to respond to users' feedback quickly.

- However, being able to deploy code frequently requires having processes set up to make deployment easier and to ensure that the code that gets deployed is high-quality.

- The need to be able to deploy high quality code often has brought increased focus in recent years on the practices of ***continuous integration*** (***CI***) and ***continuous delivery*** (***CD***), often abbreviated together as ***CI/CD***.

- Let's take a minute to understand what each of these practices entails.

Continuous integration

- The generally accepted high-level definition of ***continuous integration*** looks something Martin Fowler's:

  Continuous Integration is a software development practice where members of a team integrate their work frequently... Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.

- There are two important aspects of this definition we'll focus on. The first is that, under a continuous integration practice, each developer's changes are incorporated frequently together with the rest of the team's work.

- If we think in terms of the GitHub flow workflow we discussed in this course, this aspect of continuous integration suggests that no feature branch should be too long-lived. Instead, we should work in small increments, frequently merging feature branches back into the main branch (via pull requests, of course).

- The motivation behind integrating frequently is that the longer code remains unintegrated, the more likely it is to have fallen out of sync with the rest of the codebase, making it more challenging to integrate (e.g. increasing the likelihood and severity of merge conflicts).

- Integrating frequently, on the other hand, ensures that every time we merge a feature branch into the main branch, it represents a small step that can be easily reconciled with the rest of the code. This allows developers to spend more time writing code to move a project forward and less time trying to get their code to work with the rest of the codebase.

- The other aspect of Fowler's definition to focus on is the practice of building and testing the codebase each time new code is integrated (e.g. every time a pull request is merged). Indeed, testing is what most people think of when they think about continuous integration.

- By testing code each time it's integrated, we ensure that we get frequent feedback (from our tests) on new code. This, in turn, helps to ensure high code quality, since the sooner we are alerted to problems with our code, the easier it is to fix those problems.

- Modern tooling is helpful here. There are many platforms and tools designed to help us automatically run tests at various stages of the development cycle, even as frequently as every commit. We'll explore some of this tooling below.

Continuous delivery

- ***Continuous delivery*** is often the other side of the coin from continuous integration. Under a robust continuous delivery practice, not only is a software project built and tested every time new code is integrated, it can also be deployed essentially at any point in its lifecycle (e.g. after every new change is merged into the main branch).

- What continuous delivery looks like in practice can vary depending on the size and complexity of the software project.

- For some projects, it may be possible to fully automate deployment so that the entire project is automatically built and released to production every time new code is merged into the main branch (full automation like this is often referred to as ***continuous deployment***).

- For other projects, it may not be possible (or desirable) to automatically deploy each time code is pushed to the main branch. For these projects, continuous deployment might mean having an automated pipeline set up so that, when a release is prepared and ready to go, someone can simply "push a button" to deploy that release.

- In any case, there are some key features that should be present in any continuous delivery practice:
    - Under a healthy continuous delivery practice, code is deployed to production frequently. The motivations behind deploying frequently are similar to those behind integrating frequently. Specifically, the more frequently a project is deployed to production, the smaller each deployment is. This helps to minimize risk and also to increase the velocity of deployment.
    - Automation is an important element of a healthy continuous delivery practice. As much of the process of deployment as possible should be automated in order to reduce or eliminate repetitive overhead. The more manual intervention needed to deploy a project, the more likely it is for errors to occur during deployment.

- As a project grows in size and complexity, additional considerations are needed when deploying frequently through a continuous delivery practice. The Google engineers who authored the "Continuous Delivery" chapter of Software Engineering at Google have some helpful guidance for continuously delivering larger projects:
    - Guard new features behind ***feature flags***, which are simple configuration settings that can be used to turn specific features "on" or "off" within the deployed project. This provides a high degree of control over who sees each feature and when.
    - Use ***staged rollouts***, where new features are released slowly over time to increasing percentages of the userbase. This can simplify things when a bug does make it into production, since fixing things for a smaller number of users is easier than fixing them for everyone all at once. Staged rollouts can also help ensure that enough infrastructural capacity is available to support new features.
    - Use data to help make decisions about what to deploy when. Practices like ***A/B testing***, where different sets of users use two different versions of the product (version "A" and version "B") and metrics are collected about

each version, can be helpful for supplying this kind of data.

- Again, like with continuous integration, modern tooling can help make it easier to maintain a reliable continuous delivery practice. Let's start to take a look at some of this tooling.

Building a CI/CD pipeline with GitHub Actions

- There are many wonderful tools and platforms for setting up continuous integration and delivery processes for a software project. For example, Travis CI and CircleCI are both widely used platforms for creating highly configurable CI/CD pipelines that can automatically build, test, and deploy a software project.

- Here, we'll explore how to use a feature of GitHub called GitHub Actions to set up a Ci/CD pipeline for a software project. GitHub Actions is a powerful platform for automating workflows tied to a GitHub repository.

- There's a lot you can do with GitHub Actions, including everything from automatically welcoming a new contributor to a repository, to triaging bug reports, to performing a security scan of your codebase, but one of its most popular uses is to create CI/CD pipelines that run automatically when new code is pushed to the repository.

- We'll focus on the CI/CD use case for GitHub Actions here. However, the syntax and approach we learn here for setting up a CI/CD pipeline with GitHub Actions can be applied to do just about anything with GitHub Actions. You should refer to the GitHub Actions documentation to build on what we learn here.

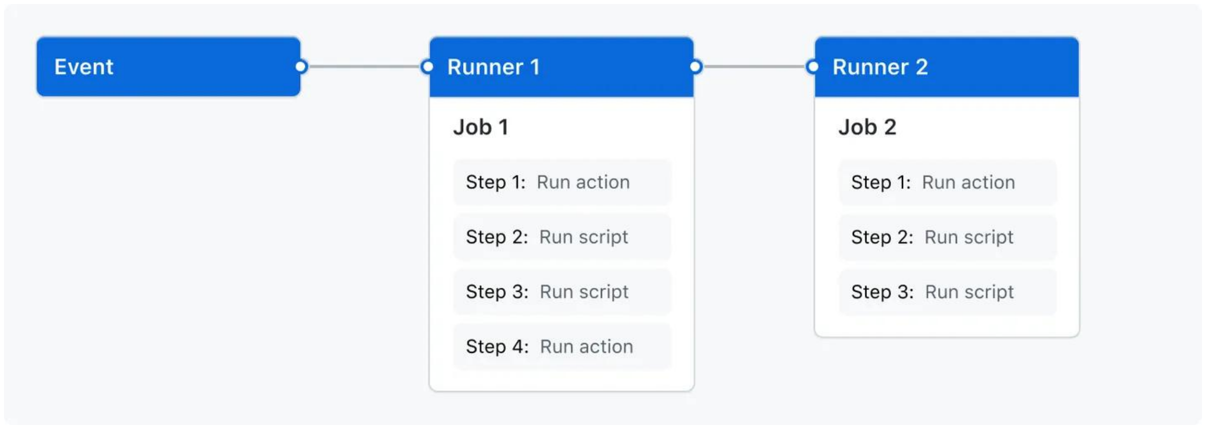
- GitHub Actions are centered around ***workflows***, which are automated processes that are triggered when one or more ***events*** occurs within the associated GitHub repo.
    - There are many different events that can trigger a workflow, ranging from someone making a comment on one of the repo's issues to someone pushing new code to a specific branch within the repo.

- Each workflow runs one or more ***jobs***, each of which consists of one or more steps that each perform some particular task, specified either as a shell script (which can itself run other commands) or as an ***action***, which is a reusable building block that performs a given task.

    - GitHub has a Marketplace where you can find open-source actions created by the GitHub community, or you can build your own custom actions.

- A workflow's jobs can be configured to run in parallel or sequentially, and every job runs inside its own virtual machine called a ***runner***. GitHub provides hosted Linux, MacOS, and Windows runners.

- An entire workflow looks something like this:



From the GitHub Actions documentation

- To build an automated CI/CD pipeline using GitHub Actions, we can create one or more workflows whose jobs build, test, and deploy our application and set these workflows up to run whenever new code is pushed to our repository. Let's start to explore how to do this.

A simple "build and test" GitHub Actions workflow

- Let's start exploring how to use GitHub Actions by creating a simple workflow that will build and test our application every time new code is pushed to our repository on GitHub.

- We'll assume we have an application set up to work with and that this application already lives in a repository on GitHub. Further, let's assume our application already has the following commands set up for building and testing the application:
    - **`npm run build`** – Builds the application into a directory named `build/`.
    - **`npm run test:integration`** – Runs integration tests using Jest.

    - **`npm run test:e2e`** – Runs E2E tests headlessly using Cypress.

- We'll set up our workflow to execute these commands.

- All GitHub Actions workflows are defined using YAML file syntax in a directory called `.github/workflows/` (note the leading `.` character, which is important), so let's begin by creating the following file:

  ```
  .github/workflows/build-and-test.yml
  ```

- YAML is in many ways somewhat like JSON, allowing us to create files representing structured data. Most of the time, a YAML file is used to define a map that contains keys and values, much like a JSON object. This is how our workflow will be structured.
    - You can learn more about YAML syntax here.

- The complete YAML syntax used by GitHub actions workflows is documented on GitHub. It mainly consists of a set of predefined YAML keys, each of which represents a specific configuration we can set on the workflow.

- For example, the first configuration we'll set in `build-and-test.yml` is `name`, which simply allows us to assign a name to the workflow to be displayed in the GitHub Actions UI:

  ```
  name: Build and test
  ```

- Next, we'll use the on configuration to list the GitHub repository events we want to trigger our workflow. The value for this configuration can be an array listing multiple events if we want, but we'll just specify a single one, `push`, which means our workflow will be triggered whenever new code is pushed to our repository:

  ```
  name: Build and test
  on: [ push ]
  ```

- Note that the `push` event allows us to further configure the workflow only to run when certain branches are pushed to, but for now, we won't do this. This means our workflow will run any time code is pushed to *any* branch. For now, this is fine, since we're only working on the `main` branch anyway. We'll update things later for special processing of pull requests.

- Next, we'll use the `jobs` configuration to list the jobs associated with this workflow. Each individual job is listed under `jobs` and must be assigned a unique identifier. **The way we do this is important, since indentation is syntactically significant in YAML.** Each job must be indented one level under `jobs`:

```
name: Build and test
on: [ push ]
jobs:
  build-and-test:
```

- Under the `build-and-test` job, we can configure the job itself and list its steps, indented one additional level beyond the job ID. For now, we'll just set one job configuration, `runs-on`, which allows us to specify what kind of runner the job should run on. Here, we'll use an Ubuntu Linux runner (with the latest available version of Ubuntu):

```
...
jobs:
  build-and-test:
    runs-on: ubuntu-latest
```

- As an aside before we specify the steps for our job, it might be helpful to better understand the structure of the data we're specifying in this YAML file if we look at what this data would look like if we were using JSON to specify it. In particular, keys at the same indentation level in a YAML file are the same as keys within the same object in a JSON file, and every new level of indentation creates what would be a new, nested object in JSON. In particular, here's what the file we've written so far would look like in JSON:

```
{
  "name": "Build and test",
  "on": [ "push" ],
  "jobs": {
    "build-and-test": {
      "runs-on": "ubuntu-latest"
      }
    }
  }
```

- Now we're ready to list the steps of our `build-and-test` job using the [`steps` configuration](). This configuration's value should be an array listing the different steps of the job. We've seen that in YAML, arrays can be listed in square brackets `[ ]`, like the value we specified for the `on` configuration above. Arrays can also be broken over multiple lines by prefixing each individual element of the array with a dash `-` character.

- For example, we'll add a first step that simply checks out our GitHub repo so the code is available for the rest of the job. We'll do this using the [Checkout]() action from the GitHub Actions Marketplace. To indicate this, we'll use the [`uses` configuration]():

```
...
jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
```

- Once the code is checked out, we'll use another action from the Marketplace to [set up a Node.js environment]() for us to use to run and test our application. This particular action accepts an input to allow us to specify a specific version of Node.js to use, and we can specify this using the `with` configuration:

```
...
steps:
  - uses: actions/checkout@v3
  - uses: actions/setup-node@v3
    with:
      node-version: 18
```

- Now we're ready to start doing some work with our application. Our first step will be to install all of the dependencies listed in `package.json`, which we can do by running the command `npm ci` (where `ci` is short for "clean install"), a command that's similar to `npm install` but intended for use in automated environments like the one our workflow represents.

- To run terminal commands like `npm`, we'll use the `run` configuration to specify the step instead of `uses`:

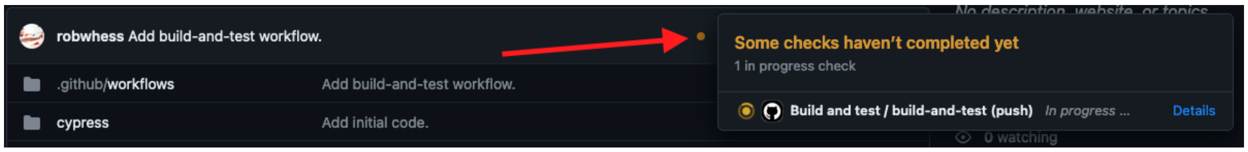
```
...
steps:
  ...
  - run: npm ci
```

- Finally, we can build our application and run both its integration tests and its unit tests. We'll do this in three separate steps, so the output of the workflow will be more organized and the results of different operations more easily located in that output:

```
...
steps:
  ...
  - run: npm run build
  - run: npm run test:integration
  - run: npm run test:e2e
```

- That's all we'll do with our workflow for now. If we commit it and push it to GitHub, we should see it execute. In particular, now when we visit our repository on GitHub, we'll see a new small icon appear next to the hash of the most recent commit in the status line at the top of the list of files.

- If our workflow (or any workflow) is currently running, this icon will be a yellow dot. If we click on this dot, a dialog will appear with more information about the running workflow, including a "details" link we can click to see the output of the workflow:



robwhess Add build-and-test workflow.

.github/workflows   Add build-and-test workflow.

cypress   Add initial code.

Some checks haven't completed yet

1 in progress check

Build and test / build-and-test (push)   In progress ...   Details

0 watching

- Once all of our workflows complete, the yellow dot icon will change either to a green checkmark if all steps of all workflows completed successfully or to a red "X" if any step of any workflow failed (e.g. if one or more of our tests failed).

- We can also see information about all of our repo's workflows on the "Actions" tab of our repository's GitHub page.

Saving workflow artifacts

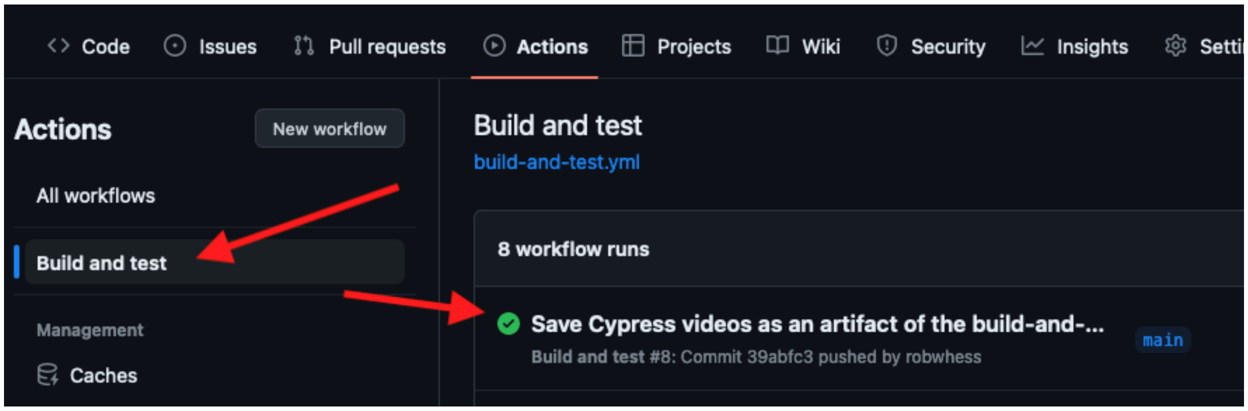
- Sometimes, our workflows will generate artifacts such as build and test outputs that we'll want to save for later use. For example, in the "Build and test" workflow we created above, we might want to save the videos of our end-to-end tests being executed that are automatically recorded by Cypress, so we can watch those videos to help debug if any of the tests fail.

- This is very easy to do because there's an official action on the GitHub Actions Marketplace for uploading workflow artifacts to be saved after the workflow is complete.

- We can specify this action as the last step of our `build-and-test` job in order to store the Cypress videos our workflow produces. We'll need to specify two inputs to this action, one to provide a name for the artifact being stored, and one to indicate the path where the artifact lives within the working directory for the workflow (which is the top-level project directory of our repo). We'll store the entire `cypress/videos/` directory:

```
...
steps:
  ...
  - uses: actions/upload-artifact@v3
    with:
      name: e2e-videos
      path: cypress/videos/
```

- If we commit the updated workflow and push it to GitHub, it will store the Cypress videos for us to view later. We can find the videos (and any other stored artifacts) by navigating to the "Actions" tab for our repo on GitHub, then clicking the name of the workflow in the sidebar on the left, followed by the workflow run whose artifacts we want to see:



Build and test
build-and-test.yml

8 workflow runs

Save Cypress videos as an artifact of the build-and-... main
Build and test #8: Commit 39abfc3 pushed by robwhess

- Artifacts generated by the selected workflow run will be listed at the bottom of the page detailing that workflow run, and we can click on any artifact to download it.

- As we'll soon see, artifacts stored in this way can also be downloaded and used by other jobs and workflows.

Setting up a reusable workflow

- We want to work towards deploying our application using a GitHub Actions workflow. In order to deploy our application, we'll first need to build it, and we'll also want to test it to make sure we're not deploying broken code.

- In other words, the steps we'll want to execute to deploy the application will begin with the same steps we specified in our "Build and test" workflow above.

- With this in mind, there are a few things we need to consider as we move towards a deployment workflow:
  - We don't want to deploy directly from our "Build and test" workflow, since we eventually want to run this workflow in response to different events (e.g. to run tests on pull requests), but we only want to deploy to production when new code is pushed to the `main` branch.
  - For all the reasons why we typically want to avoid duplicated code, it'd be nice not to just copy/paste all the steps from the "Build and test" workflow to the beginning of a deployment workflow.

- For situations like this, GitHub makes it possible to reuse workflows. Let's explore how to make our "Build and test" workflow reusable so we can run it prior to deployment whenever code is pushed to the `main` branch in addition to continuing to use it to run our tests every time new code is pushed to *any* branch.

- In order to make our "Build and test" workflow reusable, we'll change the event that triggers it from push to workflow_call:

```
name: Build and test
on:
  workflow_call:
jobs:
  ...
```

- This will enable us to "call" this workflow from another. Of course, changing the event that triggers this workflow means it will no longer run when new code is pushed. To do this, let's create a new workflow that calls our "Build and test" workflow.

- Specifically, let's create a new workflow file .github/workflows/test-on-push.yml. We'll set this workflow up to run on the push event, just like our "Build and test" workflow used to do. The new workflow will will have just a single job, which will use the uses configuration to run our "Build and test" workflow:

```
name: Test pushed code
on: [ push ]
jobs:
  build-and-test:
    uses: ./.github/workflows/build-and-test.yml
```

- If we commit and push these changes, we'll see our "Build and test" workflow run just as before. However, it will now be triggered through the "Test pushed code" workflow.

Setting up a deployment workflow

- Now that our "Build and test" workflow is set up to be reusable, let's create another workflow that deploys our application when new code is pushed to the main branch. We'll specifically assume the built app is suitable for deployment to GitHub Pages (a simple hosting platform for static web applications) and create a workflow to deploy the app there.

- To be able to deploy the app that's built by our "Build and test" workflow, we'll have to modify that workflow slightly. Specifically, the built version of the application created by the "Build and test" workflow is not currently accessible outside that workflow. In order to access and deploy that build from a separate workflow, we'll need to store it as a workflow artifact, similar to the way we stored the videos recorded by Cypress.

- We'll do this in a specific way. In particular, we'll set the "Build and test" workflow up to take a boolean input that is used to control whether or not a build artifact is stored, since we may not always want to store the build (e.g. if we're just running the workflow for testing purposes).

- Inputs to reusable workflows can be specified by adding the `inputs` configuration to the `workflow_call` event. We'll create a boolean input named `upload-pages-artifact`, since the artifact we're creating will be specially crafted for deployment to GitHub Pages:

  ```
  name: Build and test
  on:
    workflow_call:
      inputs:
        upload-pages-artifact:
          type: boolean
          required: false
  jobs:
    ...
  ```

- Once the input is specified, we can use its value within the workflow, much like a function argument. We'll specifically add a final step to the `build-and-test` job that uses the input value in conjunction with the `if` conditional to create a step that runs only when the input value is `true`.

- In this case, we'll use a specialized action from the GitHub Marketplace that packages and uploads a directory to be deployed to GitHub pages. We'll specifically tell that action to package and upload the `build/` directory that's created by `npm run build`:

```
name: Build and test
...
jobs:
  build-and-test:
    ...
    steps:
      ...
      - if: ${{ inputs.upload-pages-artifact }}
        uses: actions/upload-pages-artifact@v1
        with:
          path: build/
```

- With that modification made to the “Build and test” workflow, we can move on to create a deployment workflow in the file `.github/workflows/deploy-to-pages.yml`.

- We’ll configure the workflow to run on the `push` event, but *only* when code is pushed to the `main` branch (achieved by specifying the `branches` configuration for the `push` event):

```
name: Deploy to GH Pages
on:
  push:
    branches: [ main ]
```

- This workflow will have two jobs. The first will be a job that runs our reusable “Build and test” workflow, passing `true` to the `upload-pages-artifact` input to make sure the build is stored correctly:

```
name: Deploy to GH Pages
...
jobs:
  build-and-test:
    uses: ./.github/workflows/build-and-test.yml
    with:
        upload-pages-artifact: true
```

- The second job in this workflow will perform the deployment:

```
jobs:
  ...
  deploy-to-pages:
    runs-on: ubuntu-latest
```

- Importantly, because this job will need the build artifact to be successfully created and stored by the `build-and-test` job, we'll need to make sure that this job waits to run until the `build-and-test` job has successfully finished. In other words, the `build-and-test` job is a dependency of this job. We can specify dependencies using the needs configuration:

```
jobs:
  ...
  deploy-to-pages:
    needs: build-and-test
```

- Importantly, `build-and-test` here refers to the job with that name *in the same workflow*.

- Next, we'll need to set the `deploy-to-pages` job to run with special permissions. In particular, we'll need to explicitly give this job permission to write to GitHub pages and to write to a service that's used to verify the source of the deployment:

```
jobs:
  ...
  deploy-to-pages:
    ...
    permissions:
      pages: write
      id-token: write
```

- Now we can list the steps of this job. The first step will use an action from the GitHub Marketplace to configure the workflow to deploy to GitHub Pages:

```
jobs:
  ...
  deploy-to-pages:
    ...
    steps:
      - uses: actions/configure-pages@v3
```

- The second and final step in the workflow will use another action from the GitHub Marketplace to actually perform the deployment to GitHub Pages:

```
jobs:
  ...
  deploy-to-pages:
    ...
    steps:
      ...
      - uses: actions/deploy-pages@v2
        id: deployment
```

- **Importantly, this step assumes that the application to be deployed to GitHub Pages was already stored successfully using the `actions/upload-pages-artifact` action we set up in the "Build and test" workflow.**

- The reason we set an `id` configuration on this step is because we need to add one more configuration to the `deploy-to-pages` job itself. In particular, the `actions/deploy-pages` action expects the job it runs in to target a specific deployment environment named `github-pages`, so we'll need to do this.

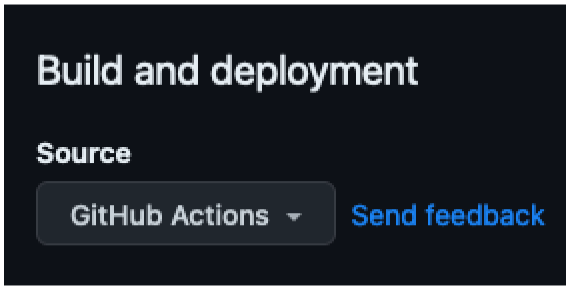
- One of the parameters of the environment we'll specify will use an output from the `actions/deploy-pages` action. It will be possible to grab that output based on the `id` we specified.

- Here's the specific configuration we need to add to the `deploy-to-pages` job:

```
jobs:
  ...
  deploy-to-pages:
    ...
    environment:
      name: github-pages
      url: ${{ steps.deployment.outputs.page_url }}
    steps:
      ...
```

- The `environment` configuration we're setting here simply allows us to set up additional protection rules that must pass before the job is allowed to run.

- This completes our workflow. Before we commit and push it, we'll need to navigate to our repo's GitHub Pages settings under "Settings → Pages" and update the "source" setting to "GitHub Actions" to allow our application to be published to GitHub Pages via GitHub Actions:


Build and deployment

Source

GitHub Actions

- With that setting updated, we can commit and push our new workflow, and when it completes, we should be able to see our application published on GitHub Pages.

- There's one more small change we'll want to make here. Specifically, you may notice that when we push to the `main` branch, our "Build and test" workflow executes twice, once through the "Test pushed code" workflow and once through the "Deploy to GH Pages" workflow.

- At this point, since the tests are being executed on the `main` branch through the deployment workflow, we can now modify the "Test pushed code" workflow to run *only* on pull requests:

```
name: Test pull requests
on: [ pull_request ]
jobs:
  build-and-test:
    uses: ./.github/workflows/build-and-test.yml
```

- Now, this job will only run when a pull request is opened (or reopened) or when new commits are made to an existing pull request.

- If we wanted to, we could now also update a branch protection rule on the `main` branch to require the "Test pull requests" workflow to succeed before a pull request can be merged.

Local continuous integration using Git hooks

-