Let’s do this one brick at a time. I’ll keep it beginner-friendly and practical.

We’ll cover:

1. What “Cypress in CI” actually means
2. Running Cypress tests in **headless mode** (common to all CI tools)
3. How to set up Cypress in **GitHub Actions**
4. How to set up Cypress in **GitLab CI**
5. How to set up Cypress in **Jenkins**
6. What are **reports** and **test artifacts**, and how to save them

---

1. What does “Cypress in CI” mean?

- **Cypress** = testing tool that runs in your dev machine.
- **CI (Continuous Integration)** = a server that runs your tests automatically whenever:
    - You push code
    - You open a merge/pull request

So **“Cypress in CI”** = running Cypress tests **automatically on a CI server** instead of running them manually on your laptop.

Benefits:

- Every commit is tested.
- You catch bugs early.
- Your team can see test results in one place (CI UI).

---

2. Running Cypress in headless mode

In CI, there is **no browser window** like when you run `npx cypress open`.
So we run Cypress in **headless mode**:

Step 1: Basic headless command

From your project root:

npx cypress run

By default, this:

- Runs in **Electron** browser.
- Runs all spec files.
- Headless by default (no UI) in CI.

Step 2: Example with Chrome/Firefox (optional)

npx cypress run --browser chrome

Step 3: Save screenshots & videos (CI loves this)

In `cypress.config.js` (or `cypress.config.mjs`):

```
const { defineConfig } = require('cypress')

module.exports = defineConfig({
  e2e: {
    baseUrl: 'http://localhost:3000',
    video: true,          // record videos
    screenshotOnRunFailure: true, // screenshots when tests fail
  },
})
```

These screenshots and videos become **artifacts** later in CI.

---

3. Cypress in GitHub Actions – step by step

Assume your project is on GitHub and you already have:

- `package.json` with Cypress devDependency
- Tests under `cypress/e2e`

Step 1: Add a GitHub Actions workflow file

Create folder and file:

```
mkdir -p .github/workflows
touch .github/workflows/cypress-tests.yml
```

Step 2: Basic workflow configuration

```
name: Cypress Tests

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  cypress-run:
    runs-on: ubuntu-latest

    steps:
      # 1. Checkout code
      - name: Checkout repository
        uses: actions/checkout@v4

      # 2. Setup Node
      - name: Use Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'

      # 3. Install dependencies
      - name: Install dependencies
```

```
      run: npm install

    # 4. Start your app (if you test UI)
    - name: Start app
      run: npm start &

    # 5. Wait for app to be ready (example: http://localhost:3000)
    - name: Wait for app
      run: npx wait-on http://localhost:3000

    # 6. Run Cypress tests headless
    - name: Run Cypress
      run: npx cypress run

    # 7. Upload artifacts (screenshots & videos)
    - name: Upload Cypress artifacts
      if: always() # run even if tests fail
      uses: actions/upload-artifact@v4
      with:
        name: cypress-artifacts
        path: |
          cypress/screenshots
          cypress/videos
```

What's happening here:

- **on:** – when to trigger (push / PR).
- **checkout** – clone your repo into the CI machine.
- **setup-node** – install Node.
- **npm install** – install Cypress + app deps.
- **npm start &** – start your app in background.
- **wait-on** – waits until your app is reachable.
- **npx cypress run** – headless tests.
- **upload-artifact** – saves screenshots/videos.

Where to see results:

- GitHub → Actions → the workflow → job → logs + artifacts.

---

4. Cypress in GitLab CI – step by step

In GitLab, you use `.gitlab-ci.yml`.

Step 1: Create `.gitlab-ci.yml` in project root

```
stages:
  - test

cypress-tests:
  stage: test
  image: cypress/included:13.15.0   # example version
  services:
    - name: node:20
  script:
    # 1. Install dependencies
    - npm install
    # 2. Start app
    - npm start &
    # 3. Wait for app to be ready
    - npx wait-on http://host.docker.internal:3000 || npx wait-on http://localhost:3000
    # 4. Run Cypress
    - npx cypress run
  artifacts:
    when: always
    paths:
      - cypress/screenshots
      - cypress/videos
    expire_in: 7 days
```

Key points:

- `image: cypress/included:…` – Docker image with Cypress + browsers ready to use.

- `script` – commands executed in CI:

    - install deps
    - start app
    - wait for app
    - run Cypress headless
- `artifacts` – GitLab will store screenshots/videos, visible in the pipeline's UI.

Where to see results:

- GitLab → CI/CD → Pipelines → Job → artifacts.

---

5. Cypress in Jenkins – step by step

Jenkins is a bit more manual but same idea.

Assume:

- Jenkins node has Node.js installed (or you use a Node Docker image).
- Your project is in Git.

Option A: Declarative Jenkinsfile (recommended)

Create `Jenkinsfile` in project root:

```
pipeline {
  agent any

  stages {
    stage('Checkout') {
      steps {
        checkout scm
      }
    }

    stage('Install Dependencies') {
```

```
      steps {
        sh 'npm install'
      }
    }

    stage('Start App') {
      steps {
        sh 'npm start &'
      }
    }

    stage('Wait for App') {
      steps {
        sh 'npx wait-on http://localhost:3000'
      }
    }

    stage('Run Cypress Tests') {
      steps {
        sh 'npx cypress run'
      }
    }
  }

  post {
    always {
      // Archive artifacts: screenshots & videos
      archiveArtifacts artifacts: 'cypress/screenshots/**, cypress/videos/**', fingerprint: true
      junit 'cypress/results/*.xml' // if you configure JUnit reporter
    }
  }
}
```

Notes:

- `checkout scm` – checks out the repo.
- `sh` – run shell commands.
- `archiveArtifacts` – Jenkins stores files as **artifacts**.
- `junit` – parses JUnit XML reports so Jenkins UI can show test results.

To use `cypress/results/*.xml`, you need a **JUnit reporter** (see next section).

---

6. Reporting and test artifacts (what & why)

6.1 What are test artifacts?

Artifacts = files CI saves after the job so you can download/view them later.

For Cypress, usual artifacts:

- **Screenshots** – images of failed tests.
- **Videos** – recordings of each spec run.
- **JUnit XML report** – machine-readable report of test results.

Why they matter:

- When tests fail in CI (no UI), artifacts help you **see what happened**.

Examples:

- In GitHub Actions, we used `actions/upload-artifact` to store them.
- In GitLab, we used `artifacts:` with `paths`.
- In Jenkins, we used `archiveArtifacts`.

6.2 What is “reporting”?

Reporting = generating structured **test result files** that CI tools understand:

- How many tests passed/failed
- Which one failed
- Error messages

- Durations

Cypress can output **JUnit-style XML** by using a reporter.

Step 1: Install reporter
npm install --save-dev mocha-junit-reporter

Step 2: Configure reporter in Cypress

In `cypress.config.js`:

```
const { defineConfig } = require('cypress')

module.exports = defineConfig({
  e2e: {
    baseUrl: 'http://localhost:3000',
    video: true,
    screenshotOnRunFailure: true,
    specPattern: 'cypress/e2e/**/*.cy.js',
  },
  reporter: 'mocha-junit-reporter',
  reporterOptions: {
    mochaFile: 'cypress/results/results-[hash].xml',
    toConsole: false,
  },
})
```

Now when you run:

npx cypress run

It will also create files like:

- `cypress/results/results-1234abcd.xml`

Step 3: Tell CI to parse reports

- **GitHub Actions**: Use a JUnit parser action (optional – mainly logs + artifacts are enough).

GitLab CI: you can add:

```
artifacts:
 when: always
 paths:
   - cypress/screenshots
   - cypress/videos
   - cypress/results
 reports:
   junit: cypress/results/*.xml
```

-

Jenkins: we already added:

```
junit 'cypress/results/*.xml'
```

-

Then Jenkins/GitLab can show a **nice tests tab** with pass/fail history.

Quick mental picture (summary)

- **Headless mode** = `npx cypress run`.
- **CI pipeline** steps (regardless of Jenkins/GitHub/GitLab):
    1. Checkout code
    2. Install dependencies (`npm install`)
    3. Start app (`npm start &`)
    4. Wait for app (`npx wait-on http://localhost:3000`)
    5. Run Cypress (`npx cypress run`)
    6. Save artifacts (screenshots, videos, JUnit XML)

If you want, next we can:

- Take **one** CI (e.g., GitHub Actions or GitLab)
- Use a **very simple sample app** (like your existing Node/React app)
- Write the full `package.json`, `cypress.config`, and pipeline file so you can literally copy–paste and run.