

Full Stack Web Development

Supplementary Material - Introduction to Test In NextJS

TDD in Next.js

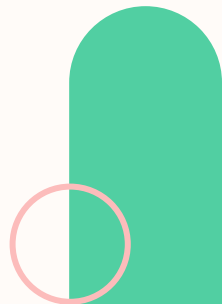
Test-Driven Development (TDD) is an approach to automated software testing that involves writing a failing test before writing the production code to make it pass. TDD helps you develop a robust test suite to catch bugs, as well as guiding you to more modular, flexible code.



TDD in Next.js

There is multiple common testing tools you can use with Next.js

- [Cypress](#)
- [Playwright](#)
- [Jest](#)
- [Vitest](#)

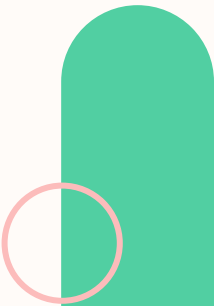


Cypress

Cypress is a next generation front end testing tool built for the modern web. Cypress address the key pain points developers and QA engineers face when testing modern applications.

Cypress make it possible to:

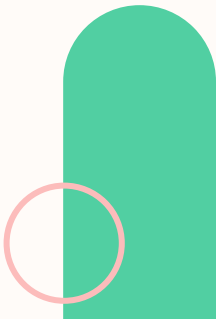
- Set up tests
- Write tests
- Run tests
- Debug tests



Cypress

Cypress offers two options for us to use what kind of type of test to create:

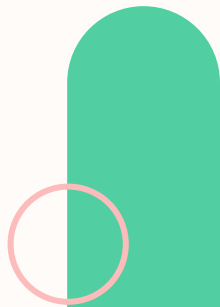
- End-to-End tests
- Component tests



End-to-End tests

E2E Testing is a technique that tests your app from the web browser through to the back end of your application, as well as testing integrations with third-party APIs and services. These types of tests are great at making sure your entire app is functioning as a cohesive whole.

End-to-end tests are great at verifying your app runs as intended, from the front end to the back end. However, end-to-end tests can be more difficult to set up, run, and maintain. There are often infrastructure needs in setting up a backend for testing purposes. Your team will need to develop a strategy on how to handle this complexity.



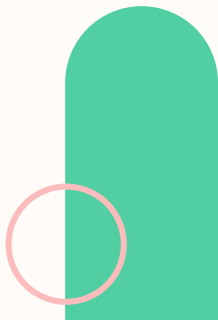
End-to-End tests

Benefits of end-to-end tests:

- Ensure your app is functioning as a cohesive whole
- Tests match the user experience
- Can be written by developers or QA Teams
- Can be used for integration testing as well

Considerations for end-to-end tests:

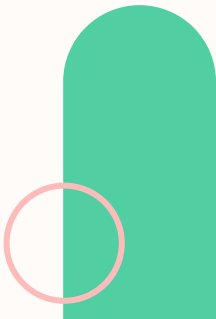
- More difficult to set up, run, and maintain
- Provision testing infrastructure in CI
- Testing certain scenarios require more setup



End-to-End tests

Common scenarios for end-to-end tests:

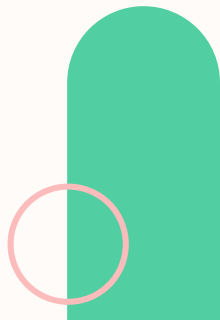
- Validating critical workflows like authentication and purchasing
- Ensuring data is persisted and displayed through multiple screens
- Running Smoke Tests and System Checks before deployment



Component tests

Component tests differ from end-to-end tests in that instead of visiting a URL to pull up an entire app, a component can be "mounted" and tested on its own. This allows you to focus on testing only the component's functionality and not worrying about other nuances with testing a component as part of the larger application.

One thing to consider, though, is even if all your component tests pass, it does not mean your app is functioning properly. Component tests do nothing to ensure that all the layers of your app are working well together. Therefore, a well-tested app has a combination of end-to-end and component tests, with each set of tests specializing in what they do best.



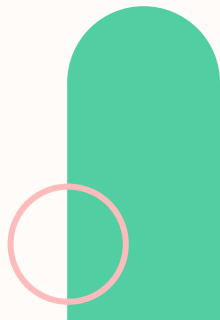
Component tests

Benefits of component tests:

- Easier to test components in isolation
- Fast and reliable
- Easy to set up specific scenarios in tests
- Don't rely on any external system to run

Considerations for component tests:

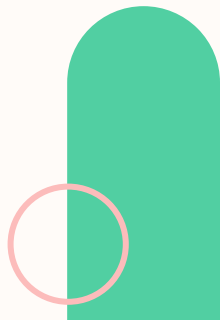
- Do not ensure overall app quality
- Do not call into external APIs/Services
- Usually written by developers working on the component



Component tests


Common scenarios for component tests:

- Testing a date picker works properly for a variety of scenarios
- That a form shows and hides specific sections based on input
- Testing components coming out of a design system
- Testing logic not tied to a component (like unit tests!)



TDD Example - Setup

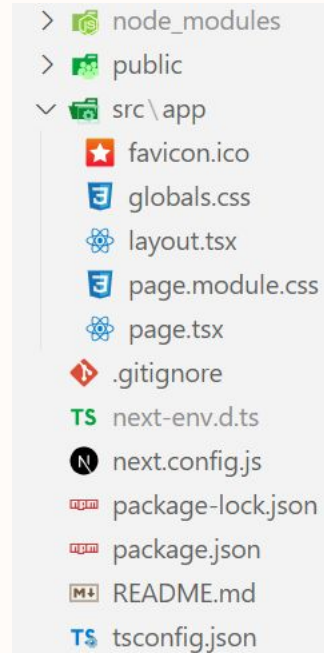
Lets setup your project!



```
mkdir next-with-cypress  
cd next-with-cypress  
npx create-next-app@latest .  
npm i cypress --save-dev
```

TDD Example - Setup

After running all the command above you should have a project structure that look like this.



A screenshot of a file explorer window showing the project structure. The tree view on the left shows a collapsed 'src' folder, which is expanded to show the 'app' subfolder. The 'app' folder contains several files: 'favicon.ico', 'globals.css', 'layout.tsx', 'page.module.css', and 'page.tsx'. Below the 'app' folder, the following files are listed in the main pane: '.gitignore', 'next-env.d.ts', 'next.config.js', 'package-lock.json', 'package.json', 'README.md', and 'tsconfig.json'. Each file is preceded by a small icon representing its type (e.g., a star for favicon, a document for CSS/JS/JSON, a book for README, and a TypeScript logo for .ts files).

- > node_modules
- > public
- ▼ src\app
 - ★ favicon.ico
 - 📄 globals.css
 - 📄 layout.tsx
 - 📄 page.module.css
 - 📄 page.tsx
- 📄 .gitignore
- 📄 next-env.d.ts
- 📄 next.config.js
- 📄 package-lock.json
- 📄 package.json
- 📄 README.md
- 📄 tsconfig.json

TDD Example - Setup

Go into our package.json and add the following command to our scripts.

Now in our tsconfig.json change our module resolution into “node”.

```
"moduleResolution": "node",
```



```
package.json

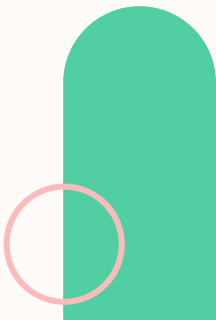
{
  "scripts": {
    "cy:open": "cypress open",
    "cy:run": "cypress run"
  }
}
```

TDD Example - Setup

Run Cypress for the first time to generate examples that use their recommended folder structure:



```
npm run cy:open
```



TDD Example - Setup

Cypress open command will open a new windows like the picture below. Lets choose component testing and configure our front-end framework with Next.js.

cy next-with-cypress (master) v13.5.0 Docs Log in

Welcome to Cypress!

[Review the differences between each testing type →](#)



E2E Testing

Build and test the entire experience of your application from end-to-end to ensure each flow matches your expectations.

Not Configured



Component Testing

Build and test your components from your design system in isolation in order to ensure each state matches your expectations.

Not Configured

Project setup

Confirm the front-end framework and bundler used in your project.

Front-end framework

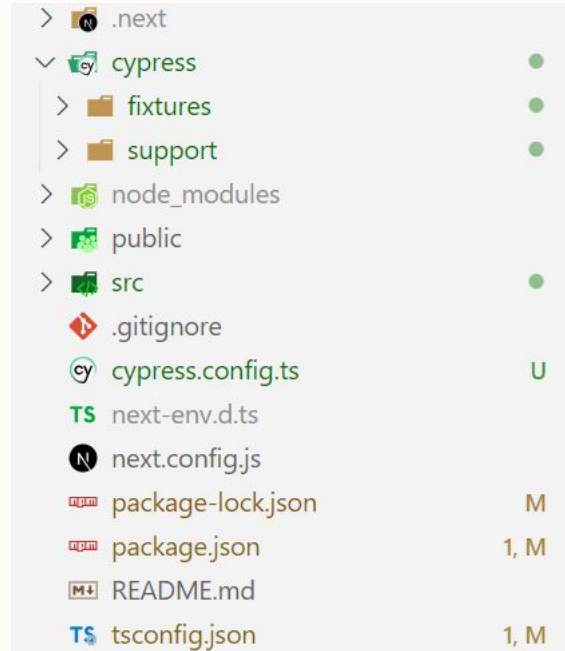
Next.js

Next step

Back

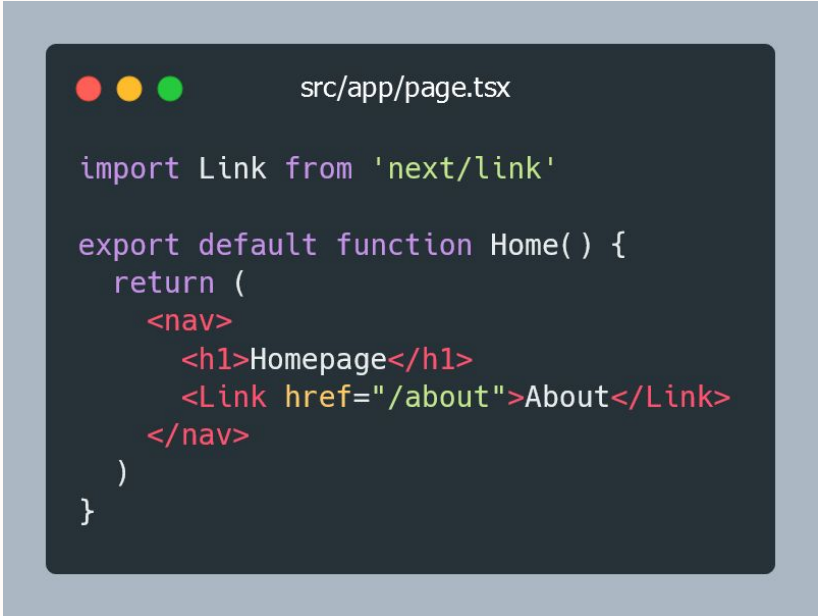
TDD Example - Setup

After the config finish running your folder structure should look like this.



TDD Example – Creating Components

Now let's modify the content of our page.tsx inside src/app.



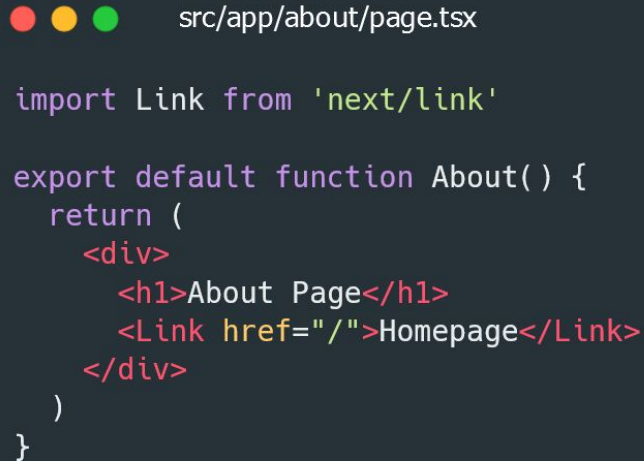
```
src/app/page.tsx

import Link from 'next/link'

export default function Home() {
  return (
    <nav>
      <h1>Homepage</h1>
      <Link href="/about">About</Link>
    </nav>
  )
}
```

TDD Example – Creating Components

Still inside src/app lets create a folder name about and create a file name page.tsx



```
src/app/about/page.tsx

import Link from 'next/link'

export default function About() {
  return (
    <div>
      <h1>About Page</h1>
      <Link href="/">Homepage</Link>
    </div>
  )
}
```

TDD Example – Creating Test

Inside our src/app create a file name pageHome.cy.tsx

src/app/pageHome.cy.tsx

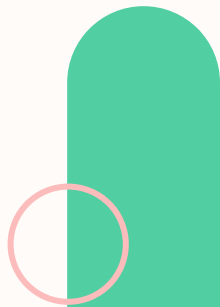
```
import Home from './page'

describe('<Home />', () => {
  it('renders', () => {
    cy.mount(<Home />)
  })
})
```

TDD Example – Creating Test

First, we import the Home component. Then, we organize our tests using the functions `describe` and `it`, which allows us to group tests into sections by using method blocks. These are global functions provided by Cypress, which means you don't have to import them directly to use them. The top-level `describe` block will be the container for all our tests in a file, and each it represents an individual test. The `describe` function takes two parameters, the first of which is the name of the test suite, and the second is a function that will execute the tests.

We defined a test using the `it` function inside `describe`. The first parameter to it is a brief description of the spec, and the second parameter is a function that contains the test code



TDD Example – Creating Test

Now let's go into our src/app/about folder and create a file name pageAbout.cy.tsx

```
src/app/about/page.tsx

import AboutPage from "../page"

describe('<AboutPage />', () => {
  it('should render and display expected content', () => {
    cy.mount(<AboutPage />)

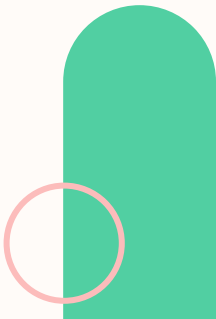
    // The new page should contain an h1 with "About page"
    cy.get('h1').contains('About Page')

    // Validate that a link with the expected URL is present
    // *Following* the link is better suited to an E2E test
    cy.get('a[href="/"]').should('be.visible')
  })
})
```

TDD Example – Commands

Cypress commands don't do anything at the moment they are invoked, but rather enqueue themselves to be run later. Commands can be chained together because Cypress manages a Promise chain on your behalf, with each command yielding a 'subject' to the next command, until the chain ends or an error is encountered.

References: <https://docs.cypress.io/api/table-of-contents>



TDD Example – Running Test

Now let's try running the test using this command

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The command 'npm run cy:run -- --component' is displayed in white text.

```
npm run cy:run -- --component
```

References: <https://docs.cypress.io/guides/guides/command-line>

TDD Example - Testing Express.js Routes

After running the command above the result should look like the picture below.

(Run Starting)

```
Cypress:      13.5.0
Browser:      Electron 114 (headless)
Node Version: v18.13.0 (C:\Program Files\nodejs\node.exe)
Specs:        2 found (pageHome.cy.tsx, about/pageAbout.cy.tsx)
Searched:     **/*.cy.{js,jsx,ts,tsx}
```

Running: pageHome.cy.tsx

(1 of 2)

```
<Home />
  ✓ renders (81ms)
```

1 passing (109ms)

(Results)

```
Tests:      1
Passing:    1
Failing:    0
Pending:    0
Skipped:    0
Screenshots: 0
Video:      false
Duration:   0 seconds
Spec Ran:   pageHome.cy.tsx
```

Running: about/pageAbout.cy.tsx

(2 of 2)

```
<AboutPage />
  ✓ should render and display expected content (142ms)
```

1 passing (168ms)

(Results)

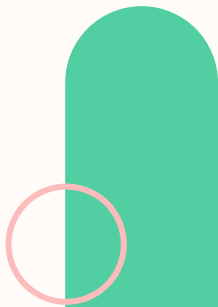
```
Tests:      1
Passing:    1
Failing:    0
Pending:    0
Skipped:    0
Screenshots: 0
Video:      false
Duration:   0 seconds
Spec Ran:   about/pageAbout.cy.tsx
```

(Run Finished)

Spec	Tests	Passing	Failing	Pending	Skipped
✓ pageHome.cy.tsx	110ms	1	1	-	-
✓ about/pageAbout.cy.tsx	173ms	1	1	-	-
✓ All specs passed!	283ms	2	2	-	-

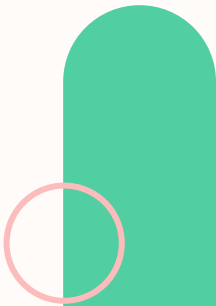
Pre-commit git hooks

While working on an enterprise development team, it is important that all typescript code linting and unit tests are passing before committing code, especially if you are using some form of continuous integration. Git hooks allow custom scripts to be ran on your repository.



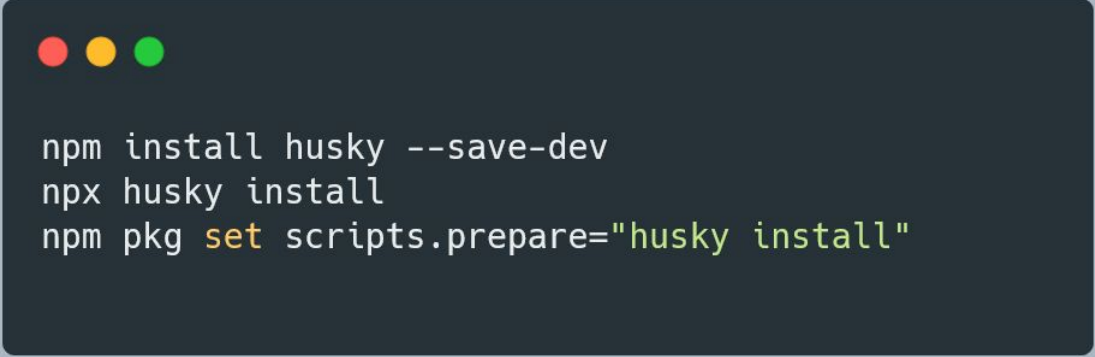
Pre-commit git hooks - Husky

Husky is a very popular (1 million downloads a month) npm package that allows custom scripts to be ran against your repository. Husky works with any project that uses a package.json file. It also works out of the box with SourceTree!



Husky - Setup

Run the command below to install husky. The last command will add a script called prepare into our package.json



```
npm install husky --save-dev  
npx husky install  
npm pkg set scripts.prepare="husky install"
```

Husky - Create a Hook

Run the command below to add a command to a hook or create a new one. The hook will get triggered whenever you make a commit.



```
npx husky add .husky/pre-commit "npm.cmd run cy:run -- --component"  
git add .husky/pre-commit
```

Husky

Now every time you commit it will run a test before you can commit.

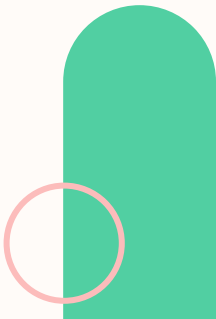


```
git commit -m "Keep calm and commit"
```

Exercise

Based on exercise slides 15

- Create a component test
- Create a hook that will run component test before commit



Thank You!

