If you try to run unit tests on components that use GraphQL queries, you will discover that you have no data. Jest can't run your queries, so if you are testing components that rely on GraphQL data, you will need to provide the data yourself. This is a good thing, as otherwise your tests could break if your data changes, and in the case of remote data sources it would need network access to run tests.

In general it is best practice to test the smallest components possible, so the simplest thing to do is to test the individual page components with mock data, rather than trying to test a full page. However, if you do want to test the full page you'll need to provide the equivalent data to the component. Luckily there's a way to get the data you need.

First you should make sure you have read [the unit testing guide]() and set up your project as described. This guide is based on the same blog starter project. You will be writing a snapshot test for the index page.

As Jest doesn't run or compile away your GraphQL queries you need to mock the `graphql` function to stop it throwing an error. If you set your project up with a mock for `gatsby` as described in the unit testing guide then this is already done.

Testing page queries

As this is testing a page component you will need to put your tests in another folder so that Gatsby doesn't try to turn the tests into pages.

```
import React from "react"
import renderer from "react-test-renderer"
import Index from "../index"

describe("Index", () =>
  it("renders correctly", () => {
    const tree = renderer.create(<Index />).toJSON()
    expect(tree).toMatchSnapshot()
  }))
```

If you run this test you will get an error, as the StaticQuery in the `Layout` component is not mocked. You can fix this by mocking it, like so:

```
import React from "react"
import renderer from "react-test-renderer"
import { StaticQuery } from "gatsby"
import Index from "../pages/index"

beforeEach(() => {
  StaticQuery.mockImplementationOnce(({ render }) =>
    render({
      site: {
        siteMetadata: {
          title: `Default Starter`,
        },
      },
    })
  )
})
```

```
describe("Index", () =>
  it("renders correctly", () => {
    const tree = renderer.create(<Index />).toJSON()
    expect(tree).toMatchSnapshot()
  }))
```

This should fix the `StaticQuery` error, but in a more real-world example you may also be using a page query with the `graphql` helper from Gatsby. In this case, there is no GraphQL data being passed to the component. You can pass this in too, but the structure is a little more complicated. Luckily there's a way to get some suitable data. Run `npm run develop` and go to `http://localhost:8000/___graphql` to load the GraphiQL IDE. You can now get the right data using the same query that you used on the page. If it is a simple query with no fragments you can copy it directly. That is the case here, run this query copied from the index page:

```
query IndexQuery {
  site {
    siteMetadata {
      author
    }
  }
}
```

The output panel should now give you a nice JSON object with the query result. Here it is, trimmed to one node for brevity:

```
{
  "data": {
    "site": {
      "siteMetadata": {
        "author": "Your Name Here"
      }
    }
  }
}
```

GraphiQL doesn't know about any fragments defined by Gatsby, so if your query uses them then you'll need to replace those with the content of the fragment. If you're using `gatsby-transformer-sharp` you'll find the fragments in gatsby-transformer-sharp/src/fragments.js. So, for example if your query includes:

```
image {
  childImageSharp {
    fluid(maxWidth: 1024) {
      ...GatsbyImageSharpFluid
    }
  }
}
```

...it becomes:

```
image {
  childImageSharp {
    fluid(maxWidth: 1024) {
      base64
      aspectRatio
      src
      srcSet
      sizes
    }
  }
}
```

When you have the result, copy the `data` value from the output panel. Good practice is to store your fixtures in a separate file, but for simplicity here you will be defining it directly inside your test file:

```
import React from "react"
import renderer from "react-test-renderer"
import { StaticQuery } from "gatsby"
import Index from "../index"

beforeEach(() => {
  StaticQuery.mockImplementationOnce(({ render }) =>
    render({
      site: {
        siteMetadata: {
          title: `Default Starter`,
        },
      },
    })
  )
})

describe("Index", () => {
  it("renders correctly", () => {
    const data = {
      site: {
        siteMetadata: {
          author: "Your name",
        },
      },
    }

    const tree = renderer.create(<Index data={data} />).toJSON()
    expect(tree).toMatchSnapshot()
  })
})
```

Run the tests and they should now pass. Take a look in `__snapshots__` to see the output.

Testing StaticQuery

The method above works for page queries, as you can pass the data in directly to the component. This doesn't work for components that use `StaticQuery` though, as that uses `context` rather than `props` so you need to take a slightly different approach to testing these types of components.

Using `StaticQuery` allows you to make queries in any component, not just pages. This gives a lot of flexibility, and avoid having to pass the props down to deeply-nested components. The pattern for enabling type checking described in the docs is a good starting point for making these components testable, as it separates the query from the definition of the component itself. However that example doesn't export the inner, pure component, which is what you'll need to test.

Here is the example of a header component that queries the page data itself, rather than needing it to be passed from the layout:

```
import React from "react"
import { StaticQuery } from "gatsby"

const Header = ({ data }) => (
  <header>
    <h1>{data.site.siteMetadata.title}</h1>
  </header>
)

export default function MyHeader(props) {
  return (
    <StaticQuery
      query={graphql`
        query {
          site {
            siteMetadata {
              title
            }
          }
        }
      `}
      render={data => <Header {...props} data={data} />}
    />
  )
}
```

This is almost ready: all you need to do is export the pure component that you are passing to StaticQuery. Rename it first to avoid confusion:

```
import React from "react"
import { StaticQuery, graphql } from "gatsby"

export const PureHeader = ({ data }) => (
  <header>
    <h1>{data.site.siteMetadata.title}</h1>
  </header>
)

export const Header = props => (
  <StaticQuery
    query={graphql`
      query {
        site {
          siteMetadata {
            title
          }
        }
      }
    `}
    render={data => <PureHeader {...props} data={data} />}
  />
)

export default Header
```

If you prefer to utilize `useStaticQuery` instead, you can rewrite the Header component as:

```
import React from "react"
import { useStaticQuery, graphql } from "gatsby"

export const PureHeader = ({ data }) => (
  <header>
    <h1>{data.site.siteMetadata.title}</h1>
  </header>
)

export const Header = props => {
  const data = useStaticQuery(graphql`
    query {
      site {
        siteMetadata {
          title
        }
      }
    }
  `)

  return <PureHeader {...props} data={data}></PureHeader>
}

export default Header
```

Note that because `useStaticQuery` is a React Hook, it is a function and so can be assigned to `data` which is then passed as a prop to the `PureHeader`. This is only possible since we have also made `Header` a function component.

Now you have two components exported from the file: the component that includes the StaticQuery data which is still the default export, and another component that you can test. This means you can test the component independently of the GraphQL. In addition, whether you utilize StaticQuery or useStaticQuery, your test should still function properly.

This is a good example of the benefits of keeping components "pure", meaning they always generate the same output if given the same inputs and have no side effects apart from their return value. This means you can be sure the tests are always reproducible and don't fail if, for example, the network is down or the data source changes. In this example, `Header` is impure as it makes a query, so the output depends on something apart from its props. `PureHeader` is pure because its return value is entirely dependent on the props passed to it. This means it's easier to test, and a snapshot should never change.

Here's how:

```
import React from "react"
import renderer from "react-test-renderer"

import { PureHeader as Header } from "../header"

describe("Header", () => {
  it("renders correctly", () => {
    // Created using the query from Header.js
    const data = {
      site: {
        siteMetadata: {
          title: "Gatsby Starter Blog",
        },
      },
    }
    const tree = renderer.create(<Header data={data} />).toJSON()
    expect(tree).toMatchSnapshot()
  })
})
```

Using TypeScript

If you are using TypeScript this is a lot easier to get right as the type errors will tell you exactly what you should be passing to the components. This is why it is a good idea to define type interfaces for all of your GraphQL queries.