Sometimes, you need to create a site with gated content, restricted to only authenticated users. Using Gatsby, you may achieve this using the concept of client-only routes, to define which pages a user can view only after logging in.

Prerequisites

You should have already configured your environment to be able to use the `gatsby-cli`. A good starting point is the main tutorial.

Security notice

In production, you should use a tested and robust solution to handle the authentication. Auth0, Firebase, and Passport.js are good examples. This tutorial will only cover the authentication workflow, but you should take the security of your app as seriously as possible.

Building your Gatsby app

Start by creating a new Gatsby project using the barebones `hello-world` starter:

```
gatsby new gatsby-auth gatsbyjs/gatsby-starter-hello-world
cd gatsby-auth
```

Create a new component to hold the links. For now, it will act as a placeholder:

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

export default function NavBar() {
  return (
    <div
      style={{
        display: "flex",
        flex: "1",
        justifyContent: "space-between",
        borderBottom: "1px solid #d1c1e0",
      }}
    >
      <span>You are not logged in</span>

      <nav>
        <Link to="/">Home</Link>
        {` `}
        <Link to="/">Profile</Link>
        {` `}
        <Link to="/">Logout</Link>
      </nav>
    </div>
  )
}
```

And create the layout component that will wrap all pages and display navigation bar:

```
import React from "react"

import NavBar from "./nav-bar"

const Layout = ({ children }) => (
  <>
    <NavBar />
    {children}
  </>
)

export default Layout
```

Lastly, change the index page to use layout component:

```
import React from "react"

import Layout from "../components/layout" // highlight-line

// highlight-start
export default function Home() {
  return (
    <Layout>
      <h1>Hello world!</h1>
    </Layout>
  )
}
// highlight-end
```

Authentication service

For this tutorial you will use a hardcoded user/password. Create the folder `src/services` and add the following content to the file `auth.js`:

```
export const isBrowser = () => typeof window !== "undefined"

export const getUser = () =>
  isBrowser() && window.localStorage.getItem("gatsbyUser")
    ? JSON.parse(window.localStorage.getItem("gatsbyUser"))
    : {}

const setUser = user =>
  window.localStorage.setItem("gatsbyUser", JSON.stringify(user))

export const handleLogin = ({ username, password }) => {
  if (username === `john` && password === `pass`) {
    return setUser({
      username: `john`,
      name: `Johnny`,
```

```
      email: `johnny@example.org`,
    })
  }

  return false
}

export const isLoggedIn = () => {
  const user = getUser()

  return !!user.username
}

export const logout = callback => {
  setUser({})
  callback()
}
```

The guide on adding authentication contains more information about the flow for connecting Gatsby to an external service.

Creating client-only routes

At the beginning of this tutorial, you created a "hello world" Gatsby site, which includes the `@reach/router` library. Now, using the @reach/router library, you can create routes available only to logged-in users. This library is used by Gatsby under the hood, so you don't even have to install it.

First, create `gatsby-node.js` in root directory of your project. You will define that any route that starts with `/app/` is part of your restricted content and the page will be created on demand:

```
// Implement the Gatsby API “onCreatePage”. This is
// called after every page is created.
exports.onCreatePage = async ({ page, actions }) => {
  const { createPage } = actions

  // page.matchPath is a special key that's used for matching pages
  // only on the client.
  if (page.path.match(/^\/app/)) {
    page.matchPath = "/app/*"

    // Update the page.
    createPage(page)
  }
}
```

> *Note: You can also use the File System Route API to create client-only routes.*

Now, you must create a generic page that will have the task to generate the restricted content:

```
import React from "react"
import { Router } from "@reach/router"
```

```
import Layout from "../components/layout"
import Profile from "../components/profile"
import Login from "../components/login"

const App = () => (
  <Layout>
    <Router>
      <Profile path="/app/profile" />
      <Login path="/app/login" />
    </Router>
  </Layout>
)

export default App
```

Next, add the components regarding those new routes. The profile component to show the user data:

```
import React from "react"

const Profile = () => (
  <>
    <h1>Your profile</h1>
    <ul>
      <li>Name: Your name will appear here</li>
      <li>E-mail: And here goes the mail</li>
    </ul>
  </>
)

export default Profile
```

The login component will handle - as you may have guessed - the login process:

```
import React from "react"
import { navigate } from "gatsby"
import { handleLogin, isLoggedIn } from "../services/auth"

class Login extends React.Component {
  state = {
    username: ``,
    password: ``,
  }

  handleUpdate = event => {
    this.setState({
      [event.target.name]: event.target.value,
    })
  }

  handleSubmit = event => {
    event.preventDefault()
```

```
    handleLogin(this.state)
  }

  render() {
    if (isLoggedIn()) {
      navigate(`/app/profile`)
    }

    return (
      <>
        <h1>Log in</h1>
        <form
          method="post"
          onSubmit={event => {
            this.handleSubmit(event)
            navigate(`/app/profile`)
          }}
        >
          <label>
            Username
            <input type="text" name="username" onChange={this.handleUpdate} />
          </label>
          <label>
            Password
            <input
              type="password"
              name="password"
              onChange={this.handleUpdate}
            />
          </label>
          <input type="submit" value="Log In" />
        </form>
      </>
    )
  }
}

export default Login
```

Though the routing is working now, you still can access all routes without restriction.

Controlling private routes

To check if a user can access the content, you can wrap the restricted content inside a PrivateRoute component:

```
import React from "react"
import { navigate } from "gatsby"
import { isLoggedIn } from "../services/auth"

const PrivateRoute = ({ component: Component, location, ...rest }) => {
  if (!isLoggedIn() && location.pathname !== `/app/login`) {
```

```
    navigate("/app/login")
    return null
  }

  return <Component {...rest} />
}

export default PrivateRoute
```

And now you can edit your Router to use the PrivateRoute component:

```
import React from "react"
import { Router } from "@reach/router"
import Layout from "../components/layout"
import PrivateRoute from "../components/privateRoute" // highlight-line
import Profile from "../components/profile"
import Login from "../components/login"

const App = () => (
  <Layout>
    <Router>
      {/* highlight-next-line */}
      <PrivateRoute path="/app/profile" component={Profile} />
      <Login path="/app/login" />
    </Router>
  </Layout>
)

export default App
```

Refactoring to use new routes and user data

With the client-only routes in place, you must now refactor some files to account for the user data available.

The navigation bar will show the username and logout option to registered users:

```
import React from "react"
import { Link, navigate } from "gatsby" // highlight-line
import { getUser, isLoggedIn, logout } from "../services/auth" // highlight-line

// highlight-start
export default function NavBar() {
  let greetingMessage = ""
  if (isLoggedIn()) {
    greetingMessage = `Hello ${getUser().name}`
  } else {
    greetingMessage = "You are not logged in"
  }

  return (
    // highlight-end
```

```
    <div
      style={{
        display: "flex",
        flex: "1",
        justifyContent: "space-between",
        borderBottom: "1px solid #d1c1e0",
      }}
    >
      <span>{greetingMessage}</span> {/* highlight-line */}
      <nav>
        <Link to="/">Home</Link>
        {` `}
        <Link to="/app/profile">Profile</Link> {/* highlight-line */}
        {` `}
        {/* highlight-start */}
        {isLoggedIn() ? (
          <a
            href="/"
            onClick={event => {
              event.preventDefault()
              logout(() => navigate(`/app/login`))
            }}
          >
            Logout
          </a>
        ) : null}
        {/* highlight-end */}
      </nav>
    </div>
  )
} // highlight-line
```

The index page will suggest to log in or check the profile accordingly:

```
import React from "react"
import { Link } from "gatsby" // highlight-line
import { getUser, isLoggedIn } from "../services/auth" // highlight-line

import Layout from "../components/layout"

export default function Home() {
  return (
    <Layout>
      {/* highlight-start */}
      <h1>Hello {isLoggedIn() ? getUser().name : "world"}!</h1>
      <p>
        {isLoggedIn() ? (
          <>
            You are logged in, so check your{" "}
            <Link to="/app/profile">profile</Link>
          </>
```

```
        ) : (
          <>
            You should <Link to="/app/login">log in</Link> to see restricted
            content
          </>
        )}
      </p>
      {/* highlight-end */}
    </Layout>
  )
}
```

And the profile will show the user data:

```
import React from "react"
import { getUser } from "../services/auth" // highlight-line

const Profile = () => (
  <>
    <h1>Your profile</h1>
    <ul>
      {/* highlight-start */}
      <li>Name: {getUser().name}</li>
      <li>E-mail: {getUser().email}</li>
      {/* highlight-end */}
    </ul>
  </>
)

export default Profile
```

You should now have a complete authentication workflow, functioning with both login and a user-restricted area!

Further reading

If you want to learn more about using production-ready auth solutions, these links may help:

- Gatsby repo simple auth example
- A Gatsby email *application*, using React Context API to handle authentication
- The Gatsby store for swag and other Gatsby goodies
- Building a blog with Gatsby, React and Webtask.io!
- JAMstack PWA — Let's Build a Polling App. with Gatsby.js, Firebase, and Styled-components Pt. 2
- JAMstack Hackathon Starter - Authenticated Gatsby app starter with Netlify Identity
- Learn With Jason Livestream: How to use Netlify Identity and Netlify Functions (with Shawn Wang)